Kermes: Systematically Unveiling Kernel Memory Region Exploitation Potential

Xiang Mei†, Jennifer Miller†, Tiffany Bao†, Ruoyu Wang†,
Adam Doupé†, Kyle Zeng†, and Yan Shoshitaishvili†
†Arizona State University
†{n132,jmill,tbao,fishw,doupe,zengyhkyle,yans}@asu.edu

***Abstract*—Linux kernel exploitation begins with vulnerability triage to form an initial primitive, then proceeds to primitive escalation, and ends in achieving privilege escalation. The effort to protect Linux against attackers seeking to escalate their privileges has been an active battle between new mitigations being proposed and new ways of bypassing those mitigations being discovered. Though this battle is still ongoing, each new mitigation raises the bar for attackers, prompting the discovery of increasingly novel techniques to bypass them.**

In this paper, we propose a methodology to systematically study kernel memory regions and analyze their capabilities to be misused during exploitation, leading us to discover ten novel exploitation techniques. Through our analysis we uncovered techniques capable of bypassing both upstream Control Flow Integrity mitigations supported by the x86_64 Linux kernel, kCFI and FineIBT (including its paranoid mode). These techniques improve on existing BPF JIT spray techniques to craft kCFI hashes out of JIT instructions and to create a valid FineIBT target. We also discover a novel stack pivoting technique *ret2enter* which enables a stable stack pivot to a nearby virtual memory page without any general purpose register control, removing a prerequisite that existing pivots require. Evaluating the techniques on all current longterm kernel releases and inside complete exploits for real-world CVEs, we find that most of them carry an exploit to root with the reliability of the original. We propose and evaluate mitigations against the most impactful techniques and discuss potential mitigations against the remaining techniques which are difficult to fully mitigate.

I. Introduction

Linux runs at the core of servers, mobile phones, and embedded systems, which makes local privilege escalation on it a high-impact goal for attackers. Such an exploit typically starts from a memory-corruption vulnerability and manipulates it into progressively stronger primitives until the attacker gains administrative privileges.

On kernelCTF, Google's kernel bug bounty program, the use of ret2jit [1] (jump into user-controlled Just-In-Time compilation code) in exploits has increased. Alongside this trend, ret2jit itself has also been evolving in exploits: becoming more stable, and independent of the patched pivot2cea technique [2] (pivot stack to the ROP chain on CPU entry area). Much prior work on the Linux kernel memory concentrates on the kernel heap area, which is the closest target for the majority of today's kernel vulnerabilities: 98 of the 99 most recent kernelCTF submissions [3] target kernel heap bugs such as heap object 'use-after-free' and 'out-of-bounds'.

Linux memory management differs across architectures. In this paper, we present a comprehensive study of x86-64 Linux kernel memory regions from an exploitation perspective. We started using the Linux kernel's documentation on memory layout [4] as a guideline to explore the primitives attackers can gain on each memory region. During our analysis, we noticed the existence of some undocumented memory regions. To make our study comprehensive, we studied both documented memory regions and the undocumented ones by inspecting page tables during runtime. For each memory region, we dynamically analyze its memory-related metadata (e.g., page tables and vmemmap structs) to drill into the internal representation of each memory region. For each region, we first analyze its static properties (such as permissions, user-facing interfaces, and base-address leakability), and then examine whether the region enables two classes of exploitation primitives: *spray*, which places attacker-controlled content at a predictable address within the region, and *primitive escalation*, which upgrades a weak primitive into a more powerful one using the region's properties. Based on this analysis, we identify novel techniques in both classes and propose mitigations for each. Applying the analysis to other architectures and determining which techniques carry over remain future work.

Through our systematic study, we investigate all kernel memory regions and, considering non-privileged user reachability, evaluate 7 of them in depth, from which we discover 10 novel primitives across 6 regions. A headline finding is a one-gadget style stack pivot that escalates a control-flow hijacking primitive directly into arbitrary code execution, without requiring any additional register control at the hijack point. More strikingly, while kernel CFI is widely believed to make control-flow hijacking primitives extremely hard to exploit, we show that any unprivileged user can bypass both x86_64 kernel CFI implementations, kCFI and FineIBT, by leveraging JITed user data in the kernel, including FineIBT's paranoid mode, the strongest configuration it offers. Although these techniques do not exploit a system on their own, they substantially lower the bar for exploitation by supplying reliable primitives that attackers can compose with the initial vulnerability. For

Network and Distributed System Security (NDSS) Symposium 2027
22–26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2027.232942
www.ndss-symposium.org

this reason, we have proposed mitigations for the discovered techniques and are actively working on deliverable patches. We note, however, that some primitives (most notably the spray techniques enabled by side-channel attacks) originate at the hardware level and cannot be fully closed in software; we present them to surface the residual risk and to call for broader attention to side-channel defenses in the community.

In summary, this paper makes the following contributions:

- **Systematic study.** We present a systematic study of x86-64 Linux kernel memory regions from an exploitation perspective.
- **Primary techniques.** Four techniques supply a capability no public technique offers: *jit4cfi* and *jit4cfi+*, to our knowledge the first generic kCFI bypass and the second generic FineIBT bypass on x86_64 Linux, the latter effective even in FineIBT's paranoid mode; *ret2enter*, a one-gadget-style stack pivot requiring no register control; and *pivot2init*, which pivots without a region base leak.
- **Alternative paths.** Six further primitives reach previously unstudied regions: *pivot2dma*, *pivot2cea+*, *pivot2init+*, and *pivot2vmalloc* offer new routes to an escalation existing techniques already reach, while *vmemmap bitflip* and *pivot2espfix* are of a different kind.
- **Evaluation.**[1] We evaluate the discovered techniques on all current longterm kernel releases and inside complete exploits for real-world kernelCTF CVEs, confirming the security risk they pose against both Default and CFI-Hardened Linux kernels.
- **Mitigations.** We propose and verify potential patches that mitigate the risks brought by the discovered techniques.

II. Background

A. Linux Kernel Exploitation

An attacker reaches the kernel through the syscall interface, triggers a memory-corruption bug there, and builds the corruption into progressively stronger primitives: an arbitrary read/write or function-pointer overwrite, then control-flow hijacking and arbitrary code execution. Not every exploit needs every stage; some are skipped depending on the bug.

B. Kernel Memory Regions

The Linux kernel splits its virtual address space into distinct regions. The *direct mapping* (physmap) maps all physical memory into kernel virtual space, while the remaining regions are purpose-specific mappings. Kernel memory regions occupy either fixed or randomized locations [4]; randomized regions receive a fresh offset at each boot, as shown in Figure 1. User space does not directly allocate from these regions; instead, it uses syscalls as interfaces to trigger allocations in different regions.

Among these regions, three allocation interfaces give user space more control over what data is placed in the kernel. The **SLAB/SLUB allocator** (`kmalloc`) serves small objects from fixed-size caches backed by physmap pages, holding user-triggered allocations such as `sendmsg` buffers; the **page allocator** (buddy system) handles page-granularity requests such as ring buffers and anonymous page faults; and the **vmalloc allocator** maps non-contiguous physical pages into a contiguous virtual range, where kernel stacks (under `CONFIG_VMAP_STACK`), XDP buffers, and other large allocations reside. User activity also influences regions it cannot allocate from directly: loading a BPF program places JITed code in the module/BPF JIT region, and entering the kernel writes user register values into the CPU entry area's exception stacks.

Prior research concentrates on the heap, hunting exploitable objects [5]–[7] to escalate heap memory-corruption primitives such as UAF, while other regions remain less explored.

C. Memory Region Base Address Leaking

A commonly deployed mitigation called Kernel Address Space Layout Randomization (KASLR) randomizes the base addresses of several critical kernel memory regions at boot. An exploit that needs a predictable address in one of these regions must recover its base. While traditionally this has been bypassed by leaking a pointer to a given memory area and subtracting from it to get the base address of the region, microarchitectural attacks have been discovered which are capable of generically derandomizing KASLR when an attacker has local code execution.

In particular, prefetch attacks [8] have seen use in modern kernel exploitation to break KASLR without needing a kernel address disclosure. A variant of a prefetch attack, called EntryBleed [9], demonstrated that it is possible to perform a prefetch attack even with the Kernel Page Table Isolation (KPTI) mitigation [10], [11] enabled.

D. Control-flow Hijacking

In the Linux kernel, control flow hijacking is the most common method used to perform privilege escalation: the attacker leverages the initial memory corruption bug to modify the control flow metadata, e.g., return address of a function call and function pointers stored in the memory. Among the 99 most recent exploits submitted to Google's kernelCTF bug bounty program, 93 (93.9%) rely on control flow hijacking, confirming that it remains the dominant kernel exploitation technique in practice.

1) Related Kernel Mitigations: Setting PC is not by itself sufficient for arbitrary code execution, because mitigations on the user/kernel boundary restrict what the hijacked kernel may reach: SMEP forbids executing instructions on user pages, and SMAP forbids the kernel from directly accessing user memory, which blocks a pivot to a fake stack in user space. The ret2dir paper [12] showed both can be circumvented through the kernel's direct-mapped physical memory, where a user page is aliased at a kernel virtual address that SMEP and SMAP consider legitimate; Linux answered by marking the direct map non-executable (*NX-physmap*), so data reached this way can still be read and written but not executed.

[1]Evaluation: https://github.com/sefcom/Kermes.

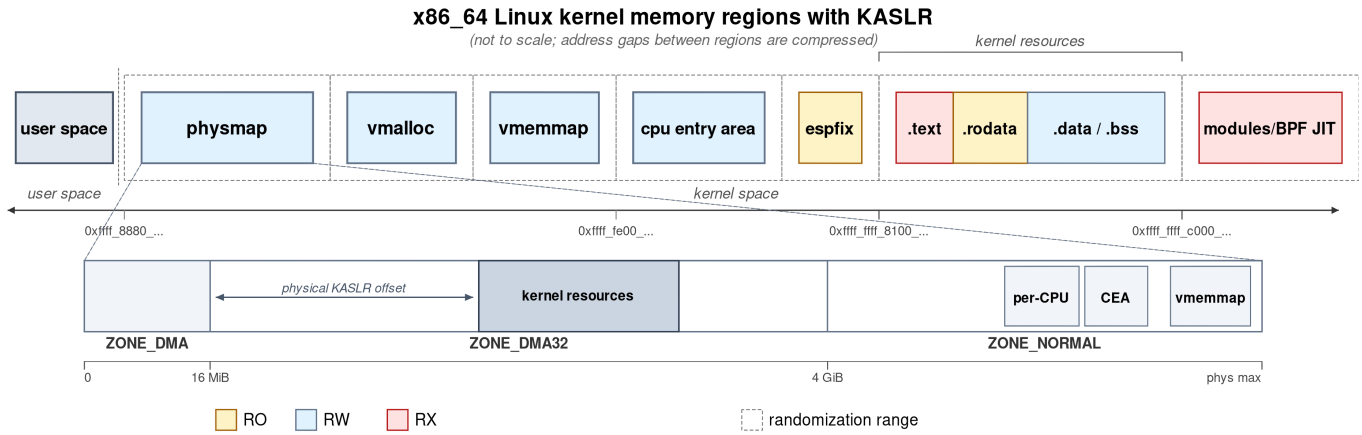


Fig. 1. x86_64 Linux kernel memory regions, colored by permission. Dashed outlines show the KASLR-randomized range for each region (the solid colored block inside is the actual mapping at one boot). The zoom illustrates the internal layout of the direct mapping (physmap): `ZONE_DMA` at the low end, `ZONE_DMA32` containing the kernel image at a physical-KASLR randomized offset, and `ZONE_NORMAL` covering RAM above 4 GiB. We show only the three most commonly used zones. For clarity, the `.data/.bss` block is shown as a single region; we omit the `idt_table` struct inside them that was set as read-only.

2) Why Stack Pivoting: These mitigations leave code reuse as the practical path: attackers chain `ret`-terminated gadgets already present in the kernel image, which requires attacker-controlled stack contents for the chain to consume. Stack buffer overflows once supplied both the hijack and the chain, but `STACKPROTECTOR` has effectively closed that path, and exploitation has shifted to the heap: none of the 99 recent kCTF submissions [3] targets a stack vulnerability, while 98 of 99 target heap-allocated objects. After corrupting a function pointer, however, the attacker typically controls little of the stack or registers at the moment of the hijack, so *stack pivoting* is required: redirecting the kernel stack pointer into an attacker-controlled region where a ROP chain has been pre-staged.

3) Stack Pivoting Techniques: RetSpill [13] pivots into user-controlled data already spilled into kernel memory, such as the registers saved on the kernel stack at every entry to kernel mode, and shows that this on-stack data alone suffices to turn one PC-control into a full ROP chain.

E. Control Flow Integrity

Control Flow Integrity (CFI) [14] restricts the possible targets of indirect branches, and the Linux kernel implements it upstream in several forms. kCFI [15] is a software scheme that instruments each indirect call site to check a hash of the call's prototype against a hash stored in a stub before the target function, which sharply restricts the targets reachable from a hijacked indirect branch. `KERNEL_IBT` is a coarse-grained hardware scheme built on Intel Indirect Branch Tracking: every indirect branch must land on an `endbr64` instruction, which the kernel places at the start of every function, so control flow cannot enter mid-function or at a misaligned instruction. FineIBT [16], [17] combines the two, relying on `KERNEL_IBT` to force entry at a function start and adding a stub that checks a hash the caller moves into a register. Prior work, System Register Hijacking [18], discovered a `KERNEL_IBT` and FineIBT bypass that reused entrypoint code. As a response to this attack, a more strict mode of FineIBT, FineIBT-Paranoid [19], was introduced which performs a hash check before an indirect call in addition to the hash check in the FineIBT stub before targets of indirect control flow.

III. Threat Model

In our threat model, the attacker is an unprivileged local user. The attacker can run arbitrary code in user space and invoke the kernel through the standard syscall interface, but holds no root capabilities (`CAP_SYS_ADMIN`, `CAP_SYS_MODULE`, `CAP_BPF`). The attacker can, however, create a new user namespace and gain capabilities within it.

We consider two kernel hardening levels:

- **Default**: the baseline configuration deployed on distribution kernels (e.g., Ubuntu). It has SMEP, SMAP, NX-physmap, CR Pinning, `STATIC_USERMODE_HELPER`, `RANDKSTACK`, `STACKPROTECTOR`, and KASLR.
- **kCFI-Hardened** and **FineIBT-Hardened**: the default configuration plus kCFI or FineIBT, respectively. Kernel CFI is only deployed in Android currently, but it is expected to be adopted in more distributions in the future.

IV. Methodology

We start from the official Linux kernel memory-layout documentation [4] as a guideline. The documentation describes each region's purpose and address range, but not the state of the memory inside it, which is what determines whether a region is useful to an attacker. We therefore supplement it with dynamic inspection of a running kernel, walking the kernel page tables to recover the permissions and present-bit state actually installed for the pages of a region, and reading their `struct page` descriptors through vmemmap to recover allocation state, reference count, and owner. Comparing the two exposes what neither gives alone: which pages are mapped but

no longer in use, which are writable, and which hold long-lived metadata rather than transient allocations. This distinguishes a region's metadata, sub-sections, and free pages, and is the input to §IV-B. Throughout this paper, unless explicitly stated otherwise, *memory* refers to virtual memory.

A. Technique Categories

Our study focuses on generic exploitation techniques that arise from the design of kernel memory regions, independent of any specific vulnerability; we therefore exclude bug-specific techniques such as race-condition primitive transformation and heap-specific primitive escalation such as cross-cache techniques. The remaining techniques fall into two categories.

A **spray** is a technique that places attacker-controlled data at a predictable kernel virtual address, giving the attacker a known location to later reference from a corrupted pointer or pivoted stack.

A **primitive escalation** promotes an existing primitive to one with stronger capability along one or both of two dimensions: **scope**, the breadth of memory or control the primitive grants (e.g., from a 16-byte write to an arbitrary write); and **semantic level**, the abstraction at which the primitive operates (e.g., from a CFHP to ROP chain execution).

B. Memory Regions Analysis

Our analytical procedure evaluates each region in three steps under our threat model (§III). Each step asks different questions about a region, and its answers feed into the next step.

(S1) Region analysis asks what the region is, in the terms an attacker cares about. We look at four properties: which of the three allocation interfaces (§II-B) land userspace content there without privilege; how its pages are permissioned, including any mapped but unused; how many independent sources of randomization separate the attacker from a known address inside it; and which security-critical metadata it holds. The first three determine whether attacker-controlled data can be placed at a *predictable* address; the last identifies potential targets for primitive escalation.

(S2) Spray primitive discovery asks how attacker data reaches the region. For each allocation path S1 found reachable, we identify the primitive that drives it and whether its backing pages can also be mapped into userspace, granting further access. A path carries into S3 when an unprivileged process can place a chosen byte sequence at an address computable from at most the base-address leak S1 established.

(S3) Primitive escalation asks what the region yields once the attacker is there, along two paths: corrupting the metadata S1 identified, and pivoting the kernel stack into a spray S2 established while holding a CFHP. We consider an escalation an improvement when it advances the primitive along one of the two dimensions of §IV-A, scope or semantic level, under prerequisites no stronger than those of a published technique reaching the same capability.

Every region in our scope goes through all three steps, and Table I records the outcome for each. The walkthrough below reports the S1 results that motivate our novel primitives, omitting for space the routine finds that only confirm what is already well understood; S2 and S3 are developed in §V, where their results combine into the techniques we contribute.

1) Direct mapping (physmap): The physmap is a contiguous, linear mapping of the entire physical memory into the kernel's virtual address space: every physical page has a corresponding virtual address at `page_offset_base` + `physical_address`. Because of this one-to-one correspondence, all kernel data, regardless of which virtual region it belongs to, is also accessible through the physmap.

Three independent factors introduce randomness into this region. KASLR randomizes the virtual base `page_offset_base` at every boot. Physical KASLR randomizes where the kernel image lands inside physical memory, which in turn shifts its virtual position within the physmap. Independently of both, runtime heap allocations are dynamic, so the physical pages they occupy are not predictable in advance. Within the region, physical memory is partitioned into zones (ZONE_DMA, ZONE_DMA32, and ZONE_NORMAL on common configurations, with ranges shown in Figure 1), and the page allocator serves requests in the order ZONE_NORMAL, ZONE_DMA32, ZONE_DMA. As Figure 1 shows, the kernel image lands inside ZONE_DMA32 at a Physical-KASLR-randomized offset, while during boot the kernel allocates its metadata starting from the end of physical memory downwards, so per-CPU data, vmemmap, and the CPU entry area sit near the end of ZONE_NORMAL.

Any unprivileged user can allocate from the physmap through the page allocator or the heap object allocator, and although the base of the physmap can be leaked via side-channel attacks, the attacker still cannot predict where their data lands, because allocation is dynamic and the state of memory at allocation time is unknown to the attacker. Looking deeper into the region, we observe that two classes of sub-regions escape this unpredictability and enable stable techniques. The first is free pages in zones that the page allocator rarely touches: ZONE_DMA is essentially untouched at runtime because no common path uses `GFP_DMA`, and the parts of ZONE_DMA32 outside the kernel image's Physical-KASLR window remain similarly stable. The second is metadata whose address is determined by only one randomization source: during boot, the kernel allocates long-lived metadata downward from the end of physical memory, so per-CPU data, pages taken by CEA, and vmemmap backing pages all sit near the end of ZONE_NORMAL at fixed offsets from the region boundary; once `page_offset_base` is leaked, they are reachable at known addresses. We develop techniques against both classes in §V-D.

2) vmalloc/ioremap: The vmalloc region is a dedicated virtual range that serves allocations requiring a contiguous virtual address space whose backing physical pages may be discrete, and assigns these allocations sequentially within the region.

An unprivileged user can spray into this region through syscalls that maintain large data buffers, such as io_uring or XDP, and its base address can be leaked via a side-channel attack (§II-C). With `CONFIG_VMAP_STACK` enabled by default, kernel stacks are also allocated in this region; combined with linear allocation, an attacker can ensure that the kernel stack at the moment of CFH is surrounded by user-controlled data.

Despite these properties, the region has not been treated as a valuable spray target, because the sprayed data lies several pages away from the stack and no general technique exists to pivot across that distance, leaving prior sprays such as ret2dir without a clear advantage here. We show in §V-B that this gap can be closed: a one-gadget-style stack pivot turns the surrounding spray into a strong primitive.

ioremap shares this virtual range but maps hardware-defined physical addresses (PCI BARs, ACPI tables) on behalf of drivers; we found no unprivileged-user path that places sufficient attacker-controlled content in an ioremap mapping under a generic configuration.

3) Kernel text, rodata, and bss: The kernel image occupies a contiguous region whose sub-sections carry different permissions: `.text` is mapped RX, `.rodata` is mapped RO, and `.data`/`.bss` are mapped RW. This region holds many of the critical structures targeted by existing privilege-escalation primitives. Our systematic study of the region surfaces a less-known property: an unprivileged user can also spray data inside it.

By design, the kernel-resources region is not meant to store large blobs of user data: its writable sections hold kernel metadata in small fixed-size pieces, and any code that requires bulk storage requests it from dynamic allocators instead. Our page-status inspection nonetheless surfaced an anomaly here. Comparing the region's page tables against the corresponding `struct page` descriptors after boot, we found contiguous runs of pages that the page tables still mapped into the kernel's address space yet whose descriptors reported them as free, with a zero reference count and no owner. A page that is simultaneously mapped and free is what interests an attacker: a valid kernel virtual address whose contents the kernel no longer considers live. Tracing these pages back through the boot path, they belong to sub-regions the kernel releases to the buddy allocator once boot completes, without tearing down the kernel-virtual mappings that cover them. Their virtual addresses therefore remain valid in the kernel-resources range while their physical backing returns to the allocator and becomes reclaimable through ordinary user allocations. §V-C identifies the sub-regions that behave this way. We build a spray primitive on this behavior.

4) Module/BPF JIT region: The Module/BPF JIT region is mapped with execute permission, used by loaded kernel loadable modules and JIT-emitted BPF programs. They are well-gated by the permission checks and verifiers.

Module loading is gated by `CAP_SYS_MODULE`, so in our threat model the attack surface in this region reduces to the BPF JIT. BPF JIT code is shaped by userspace: the JIT emits native instructions that depend on the data provided by the user. Of the two BPF interfaces, **eBPF** requires `CAP_BPF` or `unprivileged_bpf_disabled=0`, which recent distributions disable by default, leaving **cBPF** as the only path here. cBPF is the original packet-filtering interface behind `seccomp` and socket filters, reachable without privilege, and the user chooses both which allowed instructions the kernel JITs into native code and which immediates they embed.

Despite this narrow surface, prior exploitations have shown that the cBPF JIT alone is sufficient to escalate a CFHP into arbitrary code execution: ret2jit [1] embeds shellcode bytes inside cBPF immediates and redirects control flow into these malicious immediates. As discussed in §I, the recent rise of ret2jit in kernelCTF exploitations indicates that the cBPF JIT can serve as a one-gadget-style privilege-escalation primitive independently. Building on this surface, we analyze how existing techniques exploit the user-controlled `imm32` field of cBPF operands and push the technique one step further: develop two novel techniques (§V-A) bypassing x86 CFI implementations (kCFI and IBT) that ret2jit cannot defeat.

5) Other regions: The remaining regions in our scope each serve a single, well-defined purpose; we consolidate them here.

a) Virtual memory mapping (vmemmap).: vmemmap stores an array of `struct page` descriptors covering all physical memory. It is RW and is used by the memory-management subsystem; its backing pages live at a fixed offset from the end of the physmap. Userspace cannot write to it directly, but it can still be leveraged to build privilege-escalation primitives.

b) CPU entry area.: The CPU entry area is a per-CPU region holding the stacks and data structures the CPU uses when switching from user mode into the kernel, including the stacks dedicated to hardware exception handlers (e.g., divide-by-zero). A prior exploitation technique, pivot2cea [2], abused its fixed virtual address to store a ROP chain in its stacks. This path was later blocked by a patch [20] (CVE-2023-0597) that randomized the region's virtual base, but the randomization was later defeated by the side-channel attack. Independently of concurrent work [21], our methodology arrives at the same observation: the patch can be bypassed through the physmap, since the virtual base is randomized but the underlying physical pages remain at a fixed offset (§IV-B1), making the same payload reachable through physmap.

c) espfix.: espfix is a trampoline region used to safely return to 16-bit user segments on `iret`. Its base is randomized (12 bits) and side-channel leakable (§II-C), with per-CPU slots at fixed offsets. Each slot holds only 0x18 user-controllable bytes; on modern many-core systems, however, these slots aggregate into enough contiguous storage to store a ROP chain, which we develop as pivot2espfix (§V-D).

V. Findings

This section presents the novel primitives our analysis discovered. For each region, we develop S2 (spray primitive discovery) and S3 (primitive escalation) on top of the S1

TABLE I
PER-REGION CHARACTERIZATION OF THE X86_64 LINUX KERNEL ADDRESS SPACE AND THE CFHP ESCALATION TECHNIQUES THAT TARGET EACH REGION, WITH THEIR PREREQUISITES.

Accessibility: the difficulty for an unprivileged user to leave large blobs of data in the region, directly or indirectly. **Prefetch Stability**: stability of a prefetch attack across 100 runs for each memory area, attack based on Google Project Zero's implementation [22]. **Prefetch Time**: average runtime of the prefetch attack in seconds across 100 runs. Prefetch times vary by orders of magnitude because regions differ in randomization entropy and in how well-optimized the available leak method is; the kernel image has lower entropy and a well-optimized probe. ◐ Spray and ◑ Primitive escalation are the two technique categories defined in §IV-A. ● Both: technique performs spray and primitive escalation. † ret2dir's code-execution path (shellcode on physmap) is patched by NX-physmap; its spray path (placing ROP data on physmap) remains applicable. ‡ EFI runtime region and Fixmap are omitted: the EFI region is not accessible from normal kernel execution context (Ring 0), and unprivileged users have little influence over Fixmap contents. § CR4 hijacking's original path (returning to `native_write_cr4`) was mitigated by CR-Pinning; Miller et al. [18] showed the technique remains applicable through alternative `mov cr4` gadgets outside the pinned function.

Memory Region‡	Perm	Access.	Prefetch Stability	Prefetch Time	Patched Techniques	Applicable Techniques	Novel Techniques
User space	RWX	High	—	—	◐ ret2usr [23]	◑ popf [18]	
Direct mapping	RW	High	90%	7.0765	◑ ret2dir† [12]	◐ ret2dir† [12] ◑ KEPLER [24] ◑ pivot2heap	◐ pivot2dma ◐ pivot2cea+ ◐ pivot2init+
vmalloc/ioremap	RW	High	77%	7.5848		◑ RetSpill [13]	◐ pivot2vmalloc ◑ ret2enter
Virtual memory map	RW	Low	76%	7.6918			◑ vmemmap bitflip
CPU entry area	RW	Medium	60%	10.8001	◐ pivot2cea [2]		
Kernel text, rodata, bss	RWX	Low	99%	0.0006	◑ run_cmd	◑ CR0 hijacking [18] ◑ CR4 hijacking§ ◑ swapgs [18] ◑ IDT hijack [18]	◐ pivot2init
Module/BPF JIT	RX	Medium	86%	5.0512		● ret2jit [1]	● jit4cfi ● jit4cfi+
espfix	RW	Medium	76%	0.0363			◐ pivot2espfix

properties established in §IV-B. Table I organizes the findings by region, listing the techniques that target each region together with the relevant prior work. Table II compares prior techniques and ours side by side, contrasting their primitive-escalation paths and prerequisites.

A. Bypassing kCFI and FineIBT via the cBPF JIT

Building on our analysis of the BPF JIT region (§IV-B4), we present two techniques that bypass kCFI and FineIBT from an unprivileged user context: *jit4cfi* and *jit4cfi+*.

Both techniques build on ret2jit [1], which exploits the fact that cBPF JITs user-controlled `imm32` operands directly into the kernel's RX text. For example, the cBPF immediate-return instruction is JITed to `MOV EAX, imm32; RET`, leaving the four user-chosen `imm32` bytes as executable shellcode in the BPF JIT region. Prior work hijacks PC into those bytes to obtain code execution. We noticed this technique can be upgraded to bypass kCFI and FineIBT, because the same `imm32` channel lets an unprivileged user place any byte sequence in RX memory, which is exactly what is needed to satisfy a CFI hash check.

cBPF supports only `imm32` operands, and we found no other way for an unprivileged user to control more than 32 bits in the JITed code. Our budget is therefore 4 bytes, which must accomplish two things at once: (i) forge the 4-byte CFI hash, and (ii) redirect control flow into our shellcode. Since many one-byte opcodes can redirect control flow (for example, `JMP .+imm8` uses one attacker-chosen opcode byte and the following JIT byte as its operand), we use at most 3 bytes of `imm32` for the forged hash and offload the remaining byte onto adjacent opcode bytes (Figure 2).

The opcode is fixed by the cBPF instruction; only `imm32` is under our control. The forgeable hashes are therefore limited by which cBPF instruction we use. We checked all allowed cBPF instructions and grouped the usable ones into two match types (Table VI). The 4-byte hash window can be placed at any byte offset in the JITed stream relative to the opcode, so each slot is filled either by a fixed opcode byte, which must already equal the corresponding hash byte, or by an attacker-chosen `imm32` byte, which can be set to it. An alignment is usable when every slot covered by the opcode already matches and the byte at PC after the check falls on `imm32`, leaving room for the one-byte gadget. We refer to the $L = 1$ special case as *Any-match* (the opcode byte may land on any of the four hash slots) and the $L > 1$ general case as *Substring-match* (a contiguous substring of the opcode aligns with the hash window).

We performed a comprehensive study over the allowed cBPF instructions and list all the instructions that can be used for CFI bypassing in Table VI; the evaluation shows that, regardless of which control-flow hijacking primitive the attacker holds, 14% of kCFI hashes can be bypassed without

TABLE II

SIDE-BY-SIDE COMPARISON OF PRIOR PRIMITIVE-ESCALATION TECHNIQUES AND THE TECHNIQUES INTRODUCED IN THIS PAPER, CONTRASTING THEIR PRIMITIVE-ESCALATION PATHS, ADDITIONAL PREREQUISITES, AND APPLICABILITY UNDER CFI. **KERNEL IMAGE** REFERS TO THE KERNEL'S TEXT, RODATA, AND BSS SECTIONS. **CFI BYPASS**: THE TECHNIQUE REMAINS APPLICABLE WHEN KCFI OR FINEIBT IS ENABLED. † SWAPGS ORIGINALLY BYPASSED FINEIBT, BUT THE FINEIBT PARANOID MODE WAS INTRODUCED TO MITIGATE THIS PATH. ‡ AAW IN THE ESCALATION COLUMN DENOTES A LENGTH-LIMITED (≤ 0x10 BYTES), SINGLE-SHOT ARBITRARY WRITE; AN UNLIMITED, REPEATABLE AAW ALREADY ADMITS MANY KNOWN PATHS TO ACE. ¶ BY TARGETING THE VMEMMAP'S PHYSMAP-BACKING PAGE, VMEMMAP BITFLIP AVOIDS THE ADDITIONAL LEAK OF THE VMEMMAP BASE ADDRESS.

Technique	Region	Escalation‡	Additional Prerequisites	CFI Bypass
Prior work				
ret2dir [12]	physmap	CFHP → ACE	kernel image base leak, register control for pivot, physmap base leak	✗
KEPLER [24]	physmap	CFHP → ACE	kernel image base leak, register control for pivot, physmap base leak	✗
pivot2heap	physmap	CFHP → ACE	kernel image base leak, register control for pivot, heap address leak	✗
RetSpill [13]	vmalloc	CFHP → ACE	kernel image base leak, panic_on_oops=0	✗
ret2jit [1]	BPF JIT	CFHP → ACE	BPF base leak	✗
popf [18]	User space	CFHP → ACE	kernel image base leak, stack control	✗
CR0 hijacking [18]	kernel image	CFHP, AAW → ACE	kernel image base leak, register control	✗
CR4 hijacking [18]	kernel image	CFHP → ACE	kernel image base leak, register control	✗
IDT hijack [18]	kernel image	CFHP → ACE	kernel image base leak, physmap base leak, register control	✗
swapgs [18]	kernel image	CFHP → ACE	kernel image base leak	✓†
Kermes Techniques				
pivot2dma	physmap	CFHP → ACE	kernel image base leak, register control for pivot, physmap base leak	✗
pivot2cea+	physmap	CFHP → ACE	kernel image base leak, register control for pivot, physmap base leak	✗
pivot2init+	physmap	CFHP → ACE	kernel image base leak, register control for pivot, physmap base leak	✗
pivot2espfix	espfix	CFHP → ACE	kernel image base leak, register control for pivot, espfix base leak	✗
pivot2init	kernel image	CFHP → ACE	kernel image base leak, register control for pivot	✗
vmemmap bitflip	vmemmap	bit flip → page UAF	physmap base leak¶, victim page address leak	✗
pivot2vmalloc & ret2enter	vmalloc	CFHP → ACE	kernel image base leak	✗
jit4cfi	BPF JIT	CFHP → ACE	BPF base leak	✗
jit4cfi+	BPF JIT	CFHP → ACE	BPF base leak, RBP or `[RBP+8]` user-controlled	✓

any primitive transformation. To our knowledge, this is the first generic kCFI bypass; we evaluate it on 5 real-world CVEs in §VI-B.

However, jit4cfi has two limitations. First, it bypasses only the 14% of kCFI hashes that satisfy this matching condition; the remaining 86% require additional primitive transformation. Second, it cannot bypass FineIBT: FineIBT requires the target byte sequence to contain the `ENDBR64` marker, and Substring-match would need some cBPF opcode to appear as a substring of the `ENDBR64` byte sequence so that the remaining hash slots can be filled from `imm32`. None of the JIT-emitting cBPF opcodes (Table VI) is a substring of `ENDBR64`, so the marker is unforgeable under jit4cfi. We address both limitations in jit4cfi+.

jit4cfi+ uses the entire 4-byte `imm32` for the forged CFI hash, leaving no byte for a control-flow hijacking gadget; control flow instead rides the cBPF JITed code itself. The JITed cBPF function follows the standard x86-64 calling convention and ends with `leave; ret`: `leave` loads RSP from RBP, and `ret` pops the return address from the new stack. If the attacker hijacks control flow into the middle of the JITed body so that the forged `imm32` satisfies the CFI check, the kernel then executes the rest of the JITed function as ordinary code; when it reaches `leave; ret`, the stack pivots to wherever RBP points. With RBP (or the data at RBP) under attacker control, the pivot lands on a user-sprayed ROP chain, yielding ACE under both kCFI and FineIBT. The concrete implementation of FineIBT changes across kernel versions. For clarity, we explain the technique using hardware-enforced FineIBT without paranoid mode: the attacker sets the 4-byte `imm32` to `\xF3 \x0F \x1E \xFA`, placing a valid `ENDBR64` at the landing address. We evaluate its applicability across FineIBT implementations, including a successful FineIBT-Paranoid bypass, in §VI-B.

Unlike jit4cfi, which works directly from a generic CFHP, jit4cfi+ requires a CFHP whose call site has RBP or the value at `[RBP+8]` fully attacker-controlled, since these are exactly the values the `leave; ret` epilogue consumes to pivot the stack and pop the next return address (Figure 2). This is not a common case. Here we demonstrate how a stronger primitive bypasses CFI by overwriting a kernel function pointer with the address of the forged `imm32` via an 8-byte arbitrary write.

Finding a usable CFHP call site requires two properties: (i) the function pointer lives in a writable kernel region; and (ii) at invocation time, RBP or `[RBP+8]` is userland controlled. We satisfy (i) by enumerating function pointers in the kernel-resources region. There are other regions that could contain valid candidates, but we prioritize the kernel-resources region since it requires no additional base-address leak. For (ii), we drive each candidate from userspace using two complementary harness sources: probe code extracted from a kernel fuzzing corpus (e.g., syzbot), and LLM-generated probes for functions not covered by the corpus. For each candidate that we successfully invoke, we record the values of RBP and `[RBP+8]` at the call site and rank candidates by how much of those values an unprivileged user can directly control.

Among the ranked candidates, we identified one that fully bypasses kCFI and FineIBT. To our knowledge, this is the first

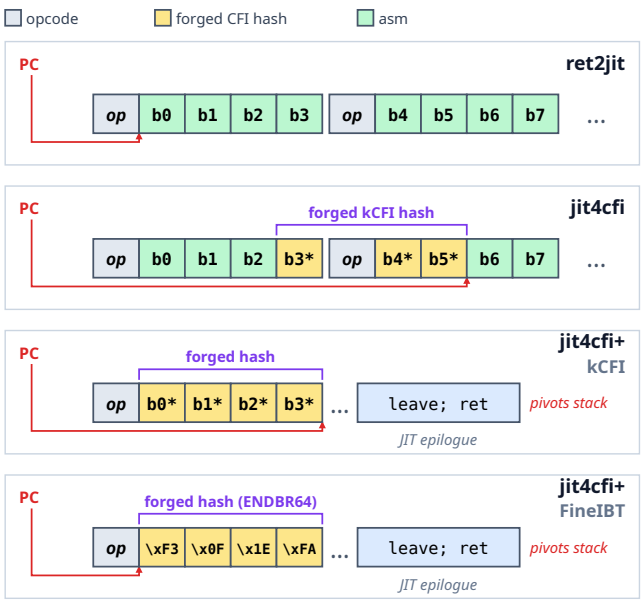


Fig. 2. How cBPF JIT-based control-flow hijack techniques operate over the JITed byte stream. In *ret2jit*, PC lands inside the imm bytes of one JITed instruction and executes them as shellcode. In *jit4cfi*, the attacker chooses three `imm32` bytes so that, together with the preceding opcode, they form a valid 4-byte kCFI hash; the last `imm32` byte holds a 1-byte control-flow hijacking gadget. In *jit4cfi+*, all four `imm32` bytes form the forged hash, and after passing the CFI check PC continues into the cBPF body and reaches the JIT `leave; ret` epilogue, triggering stack pivoting. The kCFI variant chooses any 4-byte hash matching the target, while the FineIBT variant requires the four bytes to equal `ENDBR64`.

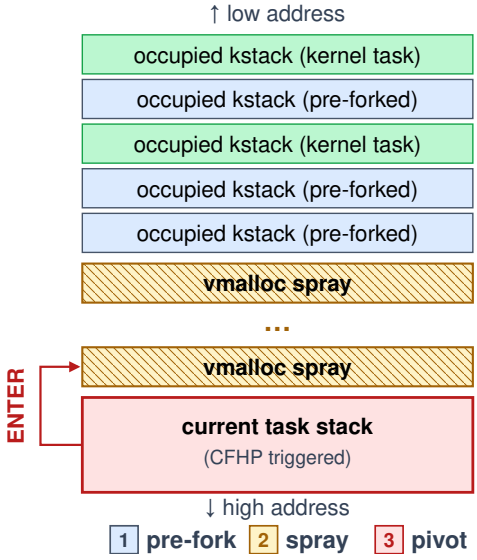


Fig. 3. vmalloc layout at CFHP time. The *ret2enter* has three steps: (1) pre-fork sleeping children to fill kstack fragments, (2) spray `0x10000`-byte vmalloc blobs, which contain the exploitation ROP chain, and (3) pivot RSP from the current task stack into the sprayed vmalloc blob with `ENTER imm16, imm8` where `imm16 > 0x5000`. The values given are safe numbers, not strict lower bounds.

generic technique that bypasses both kCFI and FineIBT. We evaluate it against the same 5 CVEs in §VI-B.

B. One-gadget-style primitive in vmalloc (pivot2vmalloc + ret2enter)

Building on the vmalloc region (§IV-B2), we escalate any CFHP into ROP chain execution without register control at the hijack site, combining *pivot2vmalloc*, which sprays a ROP chain into vmalloc allocations adjacent to the kernel stack, with *ret2enter*, which pivots there using the x86 `ENTER` instruction.

Prior work on the vmalloc region falls into two categories. The first exploits bugs in vmalloc-allocated objects (kernel stacks, large buffers) and uses vmalloc's linear allocation to corrupt adjacent objects [25]–[27]. The second uses side-channel techniques to leak kernel-stack addresses within the vmalloc region [28]. Neither category uses the vmalloc region as a stack-pivot target, which is what *ret2enter* enables: pivot to an adjacent page with only one gadget.

As established in §IV-B2, an unprivileged user can surround the kernel stack with attacker-controlled data inside the vmalloc region, which makes vmalloc a tempting target for stack-pivot exploitation. Yet anyone who tries to exploit this lands on the same wall: finding a generic gadget to pivot RSP into the surrounding sprayed pages is hard in either direction. Increasing RSP only needs to cross one guard page, but finding a gadget of the shape `add/adc RSP, imm32` with `imm32 > 0x1000` (a safe margin given the random offset introduced by `RANDKSTACK`) is hard in practice. Decreasing RSP is even harder: the displacement must cross the guard page *and* the unused frames between the current frame and the bottom of the stack. *ret2enter* closes this gap: we discovered that the x86 `ENTER` instruction can serve as a pivot gadget reaching the surrounding pages. *ret2enter* is one of the strongest primitives uncovered by our regions analysis: the `ENTER`-based stack pivot has been reachable from any CFHP since `VMAP_STACK` landed in v4.9 (2016), yet to our knowledge no prior work has discovered or exploited it.

The x86 `ENTER` instruction sets up a function's stack frame in a single instruction: `ENTER imm16, imm8` takes a nesting level `L = imm8 & 0x1f` (Intel masks the operand to five bits), pushes `L+1` qwords (the saved RBP, then `L-1` outer-frame pointers copied from `[RBP-8*i]`, then a fresh RBP), sets RBP to the new frame, and finally subtracts the unsigned 16-bit `imm16` from RSP. The net effect is that RSP decreases by `imm16 + 8*(L+1)` bytes, enough to clear both the guard page and the unused stack frames below the current frame. `ENTER` is encoded as a single-byte opcode `\xC8` followed by its two operands, so the instruction appears broadly across the kernel binary as both intended and unintended sequences; we found more than 1,000 usable gadgets in a base-config LTS kernel, and even more in distribution kernels.

As Figure 3 shows, a CFHP-to-ACE exploit chains the two pieces in three steps: pre-fork, spray, and pivot. First, since the vmalloc region contains many fragment slots that a newly-spawned process may take, the attacker spawns many sleeping

child processes to defragment vmalloc, making the layout of later allocations more controllable. Second, a vmalloc spray primitive (e.g., XDP UMEM registrations) lays multiple `0x10000`-byte blobs adjacent to where the next kernel stack will land, each holding a RET slide followed by the ROP chain, so that any sufficiently large pivot lands inside the slide and falls through into the chain. Third, because the pivot must cross the current kernel stack (`0x4000` bytes) and its guard page (`0x1000` bytes), the attacker selects an `ENTER` gadget whose `imm16` decreases RSP by more than `0x5000` bytes, then triggers the CFHP to redirect PC there; the `ENTER` carries RSP across the stack boundary into one of the sprayed blobs, where the `RET` following it begins executing the ROP chain.

Together, *pivot2vmalloc* and *ret2enter* turn an arbitrary CFHP into a full ROP-chain execution without requiring register control, providing a one-gadget-style technique. We evaluate it on 5 real-world kernelCTF CVEs in §VI-A (Table III). Our public kernelCTF exploit for CVE-2026-31419 also demonstrates the practical use of *ret2enter* [29].

C. Spray via `__init` reclaim (pivot2init/pivot2init+)

Building on our analysis of the kernel-resources region (§IV-B3), we present *pivot2init*, a spray technique that places attacker-controlled payload inside the kernel-resources region. The address of the sprayed payload can be computed from a kernel-text base leak, which any ROP build already requires, so the technique adds no leak prerequisite.

The technique exploits the kernel's `__init` reclaim behavior. The kernel frees the physical pages of `__init` sections to the buddy allocator after boot but does not unmap their kernel-virtual addresses from the kernel-resources region (§IV-B3). The freed physical pages are reclaimable from userspace through ordinary allocations such as `mmap`, while the kernel-virtual mapping continues to translate to the same physical page. An attacker can therefore allocate a userspace page, fill it with prepared payload, and access this payload through a kernel-virtual address that lies at a fixed offset from the kernel-text base.

A deeper analysis of the kernel-resources region revealed that this issue extends beyond `__init`: any pages that are freed without unmapping their kernel-virtual addresses can host the spray. We refer to such pages collectively as *nperm* regions. Our analysis on x86_64 identified four main nperm regions:

- **`__init` section.** Released after boot setup; contains code and data needed only during initialization.
- **Rodata/data gap.** Released when rodata is marked read-only; the unused space between the rodata and data sections.
- **SMP locks.** Released when SMP patching is unneeded; used only on multi-CPU boots.
- **Boot-time brk region.** Released after boot; backs early `extend_brk` allocations.

Each region adds pages to the buddy-allocator freelist while leaving the kernel-virtual mapping in place, so any of them can serve as the spray target for *pivot2init*.

The same nperm pages also live at fixed offsets inside the physmap, since the kernel allocates them at boot inside the kernel-image-backed area of the physmap. Once the attacker has leaked `page_offset_base` (§II-C), the sprayed payload becomes reachable through the physmap as well. We extend *pivot2init* to this path as *pivot2init+*, which provides an alternative path to reach these pages. *pivot2init+* requires an additional base-address leak (`page_offset_base`) but is harder to patch in practice, as it is a variant of the ret2dir spray technique [12], which has remained unpatched for over a decade.

We evaluate *pivot2init* and *pivot2init+* in §VI-A; the practical use of *pivot2init* is demonstrated in our public kernelCTF exploit for CVE-2025-38477 [30].

D. More spray techniques (pivot2dma, pivot2cea+, pivot2espfix)

Beyond *pivot2init*, we found three more spray primitives in our analysis. Each carries its own advantage (*pivot2dma* achieves more stable spraying than the original ret2dir spray, and *pivot2cea+* and *pivot2espfix* are independent of the kernel's runtime memory state), but none reduces the leak prerequisites like *pivot2init*. For space, we introduce them briefly here; collectively, they reveal that placing user-controlled data at a predictable kernel address is not a hard problem in Linux.

pivot2dma sprays free pages in `ZONE_DMA` and `ZONE_DMA32`, inside physmap. The page allocator's default order is `ZONE_NORMAL` → `ZONE_DMA32` → `ZONE_DMA`, so unless `ZONE_NORMAL` is exhausted, the lower-prioritized zones remain untouched. Since both zones contain free pages that sit before the kernel-resources block lands at its Physical-KASLR-randomized offset, their reclaim is more reliable than generic ret2dir: a randomly chosen page in ret2dir's spray could be occupied by kernel resources (because of the Physical-KASLR offset) or by heap objects in `ZONE_NORMAL`, while *pivot2dma* avoids the former entirely and reduces the effect of the latter.

pivot2cea+ revives the patched pivot2cea spray [2]: Linux closed the original technique by virtually randomizing the CPU entry area (CVE-2023-0597), but our analysis revealed that the patch only randomized the virtual mapping; the underlying physical pages still sit at a fixed offset inside physmap. Once `page_offset_base` is leaked, the attacker reaches CEA's backing pages through physmap at that fixed offset and reuses the original pivot2cea spray. Unlike ret2dir or *pivot2dma*, *pivot2cea+* targets boot-time-allocated pages whose physmap addresses are fixed once the kernel has booted, so its reliability does not depend on the runtime memory state. We note that this technique was independently discovered and published earlier in a public CTF write-up [21].

pivot2espfix sprays each CPU's espfix slot via the kernel's user-to-kernel transition path: by setting up 16-bit LDT segments and triggering an iret from a 16-bit segment, the attacker forces the kernel to copy 24 bytes (0x18) of attacker-chosen register state into that CPU's espfix slot. A single 0x18-byte slot is too small to store a ROP chain. However, by pinning

one thread per CPU and triggering the spray concurrently on each, the attacker plants the payload across N CPUs and gains $0x18 \times N$ bytes in the espfix region, enough for a ROP chain. *pivot2espfix* is independent of runtime memory state; unlike the other techniques, however, it requires racing the spray with the CFHP, since the slots are continually rewritten on every iret-from-16-bit-segment event.

E. Metadata-based primitive escalations (vmemmap bitflip)

During our region analysis, we also found primitive escalations based on corruption of kernel metadata. Even though existing techniques achieve similar privilege escalations, the techniques in this subsection reveal alternative paths through the exploitation metadata our regions analysis identified.

Building on our analysis of the vmemmap region (§IV-B5a), we present *vmemmap bitflip*, a primitive escalation that turns a single-bit flip into a page-level use-after-free by corrupting the per-page metadata vmemmap stores. Prior exploitations have studied a closely related kind of kernel metadata, the page tables: prior work [31], [32] has shown that corrupting a page-table entry, including via a single bit flip, is sufficient to achieve privilege escalation. Our study, from an attack perspective, reveals that a single-bit flip in the `struct page` descriptors stored in vmemmap is similarly sufficient to escalate primitives, yielding a page UAF. The attacker flips one bit in the target page's `_refcount` field to decrease the count by one, so that when the legitimate owner subsequently drops its reference, the page is freed while a live pointer to it still exists. The resulting page UAF can be transformed into arbitrary read/write and ultimately into arbitrary code execution; we evaluate the technique in §VI-A.

VI. Evaluation

Building on the techniques presented in §V, we evaluate them through three research questions.

- **RQ1: Feasibility and stability.** Do the techniques work, and do they keep working when the host is under load?
- **RQ2: Generality.** Do they apply across kernel versions, and do they survive inside complete exploits for real vulnerabilities?
- **RQ3: Mitigation bypassing.** Can our CFI bypasses (*jit4cfi* and *jit4cfi+*, §V-A) defeat the deployed x86_64 kernel CFI implementations?

We answer them in two evaluations that share the same structure. §VI-A covers our 8 non-CFI-bypassing techniques under the Default configuration (§III), first measuring success rate and time cost on an idle and a stress-loaded host, then repeating the measurement on all current LTS branches, and finally substituting each technique into complete exploits for 5 real-world kernelCTF CVEs. §VI-B applies the same cross-version and real-world CVE evaluations to *jit4cfi* and *jit4cfi+*, this time under the kCFI-Hardened, FineIBT-Hardened, and FineIBT-Paranoid configurations (§III). Every numerical success-rate cell reports 100 iterations.

A. Feasibility, Stability, and Cross-Version Applicability

We evaluate each of the 8 non-CFI-bypassing techniques in a QEMU guest with 2 virtual CPUs and 4 GB of RAM, launched inside a Docker container on a 13th Gen Intel Core i7-1360P host. The guest runs under KVM, so its virtual CPUs are ordinary host threads sharing the host's physical cores rather than cores reserved for the guest. Table III reports three measurements per technique: **Idle** and **Stressed** fix the kernel at one 6.12 LTS release and vary the host load, the latter with `stress-ng` consuming half of the host's CPU and RAM; the **LTS releases** columns instead hold the host idle and run each technique on the latest release of all six longterm branches. For the **Idle**, **Stressed**, and **LTS releases** measurements, the prerequisites of §III are supplied by a kernel module that triggers the control-flow hijacking primitive and discloses the relevant region bases, so those rates measure each technique in isolation; the **CVE evaluation** columns instead replace the spray primitive of a real exploit with each of our techniques, testing whether they apply once the prerequisites must come from an actual vulnerability rather than from our module. Two do not: *vmemmap bitflip* needs a single-bit write and the victim page's address, which no CVE in our set provides, and *pivot2espfix* needs an espfix in-region leak on top of the usual kernel image leak and register control. The remaining six are evaluated below.

Feasibility. Seven of the 8 techniques succeed on an idle host, completing in seconds. *ret2enter*, *vmemmap bitflip*, *pivot2dma*, and *pivot2init+* succeed in all 100 iterations, and *pivot2vmalloc*, *pivot2init*, and *pivot2cea+* fall no lower than 96%. Techniques that only read and pivot finish in about 5.7 s and those that also groom the page allocator in about 7 s, but both figures are dominated by container startup and kernel boot, so they bound the per-iteration cost rather than measure it.

Stability under load. Techniques whose target does not depend on the kernel's runtime memory or CPU state are unaffected by host load. *vmemmap bitflip* and *pivot2dma* hold at 100%, and *pivot2vmalloc* and *ret2enter* are likewise stable, because `stress-ng` mainly exercises the page and slab allocators, neither of which reclaims the vmalloc mappings they rely on. Two techniques degrade, for distinct reasons. *pivot2cea+* falls from 99% to 55% because the CPU entry area's exception-handler stack is shared with other handlers and is overwritten far more often once the CPU is busy. *pivot2init* falls from 99% to 58% because it targets pages the slab or buddy allocator is likely to claim under memory pressure, whereas *pivot2dma* targets ZONE_DMA32 pages, which the allocator reaches only after ZONE_NORMAL is exhausted. *pivot2init+* shows the same effect far more mildly, 100% to 97%, as it grooms a larger set of candidate pages and tolerates losing some.

Cross-version applicability. All seven work on all current LTS branches, 5.10 through 6.18, a span of roughly five years crossing the 5.x-to-6.x boundary, and none falls below 93% on any branch. Variation is small: *pivot2init* is the most version-

TABLE III

SUCCESS RATE, MEAN PER-ITERATION TIME COST, CROSS-VERSION SUCCESS RATE, AND CVE APPLICABILITY FOR EACH TECHNIQUE. ALL CELLS REPORT 100 ITERATIONS. **IDLE** REUSES THE 6.12 **LTS RELEASES** MEASUREMENT; **STRESSED** REPEATS IT UNDER `STRESS-NG` LOAD. **LTS RELEASES** REPORTS THE LATEST RELEASE OF EACH LTS BRANCH AS OF AUGUST 2026, ON AN IDLE HOST. **CVE EVALUATION** REPORTS EACH TECHNIQUE CARRYING A REAL-WORLD KERNELCTF EXPLOIT. ⋆ *Original exploit* IS THE STOCK EXPLOIT'S OWN SUCCESS RATE, THE BASELINE EVERY TECHNIQUE ROW IS MEASURED AGAINST; A CELL AT OR ABOVE IT PRESERVES THE EXPLOIT'S RELIABILITY. † *ret2enter* MEASURES THE COMBINED *pivot2vmalloc* + *ret2enter* CHAIN. ‡ *pivot2espfix* **IDLE/STRESSED** CELLS GIVE THE DERATED RATE WITH THE RAW RATE IN PARENTHESES: ESPFIX IS LEAKED ONLY TO ITS 4 KB PAGE, AND ONLY 1 OF ITS 64 SLOTS IS CORRECT (§VI-A). ALL OTHER *pivot2espfix* CELLS ARE DERATED (<1%). ¶ *not available*: NO CVE HERE SUPPLIES *vmemmap bitflip*'S WRITE PRIMITIVE. A ROP CHAIN SPENDING THE HIJACK ON ONE 8-BYTE WRITE REACHES ROOT ON ALL FIVE, BUT IS NOT COMPARABLE TO THE OTHER ROWS, SO IT IS NOT SCORED. § TIME COST INCLUDES DOCKER CONTAINER STARTUP AND KERNEL BOOT.

	Idle (LTS 6.12)		**Stressed (LTS 6.12)**		**Success rate on latest LTS releases (Aug 2026)**						**CVE evaluation**				
Technique	**Succ.**	**Time (s)**	**Succ.**	**Time (s)**	**5.10**	**5.15**	**6.1**	**6.6**	**6.12**	**6.18**	**C1**	**C2**	**C3**	**C4**	**C5**
Original exploit⋆	–	–	–	–	–	–	–	–	–	–	100%	100%	99%	100%	98%
pivot2espfix‡	<1% (42%)	5.67	<1% (45%)	5.78	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
vmemmap bitflip	100%	5.63	100%	5.74	99%	100%	100%	100%	100%	100%	*not available*¶				
pivot2vmalloc	96%	5.68	96%	5.88	100%	100%	100%	100%	96%	96%	100%	100%	96%	97%	92%
pivot2init	99%	6.93	58%	7.71	100%	94%	93%	93%	99%	95%	99%	100%	94%	98%	93%
pivot2cea+	99%	5.72	55%	5.82	96%	96%	99%	98%	99%	99%	97%	100%	87%	81%	88%
pivot2dma	100%	6.94	100%	7.98	99%	100%	98%	100%	100%	100%	100%	100%	99%	97%	93%
pivot2init+	100%	7.06	97%	7.78	100%	100%	100%	100%	100%	100%	100%	100%	98%	95%	94%
ret2enter†	100%	5.68	100%	5.73	100%	100%	100%	100%	100%	100%	100%	100%	96%	94%	97%

C1: CVE-2023-4623 C2: CVE-2023-6111 C3: CVE-2023-6817 C4: CVE-2024-1085 C5: CVE-2024-26925

sensitive at 93–100%, while *pivot2vmalloc* holds 100% on the four older branches and 96% on 6.12 and 6.18. This durability follows from what the techniques depend on: each rests on a structural property of a memory region, such as its fixed placement, its permissions, or the fact that it stays mapped after being freed. Such properties are established by the region's design rather than by any particular data structure, and so they survive the refactoring that occurs across major versions.

What does change is the primitive used to reach a region, and these are interchangeable. *ret2enter* sprays vmalloc through io_uring provided-buffer rings on 6.x, but that interface is absent from the older branches, so we substitute AF_XDP UMEM registrations on 5.10 and 5.15, where *pivot2vmalloc* likewise uses `PACKET_TX_RING`. *vmemmap bitflip* accounts for the target page's reference count differing after the folio rework, three on 6.x and four on 5.10 and 5.15, and selects the bit to flip accordingly. Although allocation interfaces and page-reference counts change across kernel versions, we can adapt the exploit setup without redesigning the underlying techniques.

The exception: *pivot2espfix*. *pivot2espfix* succeeds in under 1% of iterations on every configuration and branch. Our prefetch-based method leaks the 4 KB page espfix lands on but not which of its 64 slots the executing CPU takes; the table reports both the raw rate of locating the page (42% idle, 45% stressed) and the derated rate after that one-in-64 choice. A slot is also too small to hold a chain, so the payload must be reached through a further pivot (§V-D), which needs many CPUs to contribute slots; our two virtual CPUs populate too few, and a wrong-slot pivot double-faults fatally even with `panic_on_oops` disabled.

Real-world CVEs. The measurements so far isolate each technique behind a synthetic primitive. To test them inside complete exploits, we take 5 CVEs from the kernelCTF dataset, as in prior work [18], and for each one replace the technique the original exploit uses after control-flow hijacking with ours, leaving the rest of the chain untouched. Table III's **CVE evaluation** columns give the result, with the *Original exploit* row as the baseline each technique is measured against.

We focus on *ret2enter*, the only substitution besides *pivot2init* that makes an exploit simpler rather than merely equivalent: it needs only a kernel image base leak, dropping the register control and second address leak the other techniques share with existing pivots (Table II). Integrating it reduces to copying one portable function performing the pre-fork and spray (§V-B), filling in the per-binary `ENTER` gadget address, and pointing the hijacked PC at it. Reliability is preserved: *ret2enter* matches the baseline on two CVEs and stays within six points on the rest, and the others behave comparably, with *pivot2cea+* weakest at 81–100% because the entry-area stack it targets is the most contended. The small drops come from porting rather than the techniques: the substituted code is not tuned per vulnerability, and its spray may run too early, leaving later allocations room to disturb the layout.

Summary. Six of our 8 techniques are practical: each one carries a real kernelCTF exploit to root, and five of the six stay at or above 92% on every CVE we tested, with *ret2enter* and *pivot2dma* at or near 100%. Substituting them costs the original exploits almost nothing in reliability, and *ret2enter* additionally removes a prerequisite the original chains needed. The two exceptions are *vmemmap bitflip* and *pivot2espfix*, which succeed against our synthetic primitive but demand a bit-flip write or an espfix in-region leak that real vulnerabilities rarely supply. We did not tune the implementations for individual kernel versions, vulnerabilities, or configurations, so adapting them to a specific target may improve these success rates.

B. *jit4cfi/jit4cfi+: Bypassing kCFI and FineIBT*

Having covered the non-CFI-bypassing techniques under the Default configuration, we now evaluate *jit4cfi* and *jit4cfi+* against the x86_64 kernel CFI implementations. *jit4cfi* bypasses kCFI, including FineIBT's kCFI fallback when hardware IBT is unavailable, while *jit4cfi+* additionally bypasses hardware-enforced FineIBT and FineIBT-Paranoid. We conduct two evaluations to study their cross-version and real-world applicability.

Cross-version evaluation. We run each technique on the latest point release of every LTS branch supporting the evaluated CFI configuration; Table IV reports the success rates, with shaded rows giving FineIBT's kCFI fallback, the mode Linux uses when hardware IBT is unavailable, rather than hardware-enforced FineIBT.

TABLE IV
CFI-BYPASS APPLICABILITY PER (TECHNIQUE, CFI IMPLEMENTATION, LTS RELEASE). SHADED : MEASURED ON FINEIBT-COMPILED KERNELS WITHOUT HARDWARE IBT SUPPORT, WHICH FALL BACK TO REHASHED KCFI AT RUNTIME. *n/i*: NOT IN THAT RELEASE. †: NEEDS A RUNTIME HASH LEAK (FINEIBT RANDOMIZES HASHES PER BOOT FROM 6.6). SCOPE: TABLE VII.

		LTS release			
Technique	**CFI**	**6.1**	**6.6**	**6.12**	**6.18**
jit4cfi	kCFI	100%	100%	100%	100%
	FineIBT†	*n/i*	100%	100%	100%
	FineIBT-Paranoid†	*n/i*	*n/i*	*n/i*	100%
jit4cfi+	kCFI	100%	100%	100%	100%
	FineIBT	*n/i*	100%	100%	100%
	FineIBT-Paranoid†	*n/i*	*n/i*	*n/i*	100%

A single fixed JIT landing guess succeeds roughly 20% of the time. With `panic_on_oops=0`, the usual non-panic Linux setting, a miss kills only the faulting task, allowing another offset to be tried. Each evaluation run makes these retries, and both techniques succeed in all 100 runs for every applicable version and CFI configuration. *jit4cfi* remains effective against kCFI and FineIBT's kCFI fallback, while *jit4cfi+* also remains effective against hardware-enforced FineIBT and FineIBT-Paranoid. These results show that the techniques are not limited to a single kernel version.

Real-world CVE evaluation. We integrate both techniques into five kernelCTF CVEs spanning Linux 6.1 through 6.17 and evaluate each applicable combination of technique and CFI configuration. Unlike the cross-version experiments, these evaluations execute complete exploit chains from the vulnerability trigger through privilege escalation. Table V summarizes the results under kCFI, FineIBT, and FineIBT-Paranoid.

Both techniques complete the end-to-end exploit chain under kCFI on all five CVEs. Although *jit4cfi* cannot bypass hardware-enforced FineIBT, *jit4cfi+* succeeds under FineIBT on all five CVEs, including FineIBT-Paranoid on CVE-2025-40019.

For the kCFI evaluation, the 14% of kCFI hashes *jit4cfi* can forge (§V-A) restricts the target of the hijacked function pointer, not which vulnerabilities are exploitable. CVE-2023-4623 provides an 8-byte arbitrary write, allowing us to select `tcp_prot->setsockopt`, whose hash (`0x1ab87fdb`) is forgeable. After spraying the JIT code, we overwrite the function pointer and trigger the call, bypassing kCFI and gaining control-flow hijacking through the code following the forged hash.

TABLE V
SUCCESS OF *jit4cfi* AND *jit4cfi+* ACROSS KERNELCTF CVES. A CHECK MARK INDICATES THAT INTEGRATING THE TECHNIQUE INTO AN EXPLOIT FOR THE LISTED CVE ACHIEVED PRIVILEGE ESCALATION; IT DOES NOT REPORT A SUCCESS RATE. *jit4cfi* CANNOT BYPASS FINEIBT ON A CPU WITH HARDWARE IBT SUPPORT; THAT COLUMN IS THEREFORE *n/a*. ‡ FINEIBT-PARANOID.

		jit4cfi		**jit4cfi+**	
CVE	**Kernel**	**kCFI**	**FineIBT**	**kCFI**	**FineIBT**
CVE-2023-4623	6.1, 6.12	✓	*n/a*	✓	✓
CVE-2024-50164	6.6	✓	*n/a*	✓	✓
CVE-2023-52926	6.6	✓	*n/a*	✓	✓
CVE-2024-53125	6.6	✓	*n/a*	✓	✓
CVE-2025-40019	6.17	✓	*n/a*	✓	✓‡

For the FineIBT evaluation, *jit4cfi+*'s call-site prerequisite (§V-A) is what the evaluation must satisfy. Using the methodology in §V-A, we identify `inet_af_ops->fill_link_af` as a usable target whose call site leaves RBP pointing into the reply skb. After the attacker grooms this buffer and overwrites the function pointer, the JITed body reaches `leave; ret` and pivots to the controlled ROP chain.

Among the five CVEs, CVE-2025-40019 runs natively on Linux 6.17, which supports fully functional FineIBT-Paranoid and therefore provides the strongest CFI configuration evaluated against our techniques. We evaluate it in the same bare-metal setting on an Intel i7-1185G7 processor with hardware IBT support. FineIBT-Paranoid adds a caller-side hash check to the hardware `ENDBR64` check, so *jit4cfi+* places the leaked per-boot hash and `ENDBR64` in separate operands at their required offsets. Starting from the vulnerability's arbitrary read/write primitive, the exploit overwrites the target function pointer, passes both checks, and completes the end-to-end privilege escalation.

Overall, *jit4cfi* provides a generic kCFI bypass, while *jit4cfi+* extends it to hardware-enforced FineIBT and FineIBT-Paranoid.

VII. MITIGATION

To mitigate *jit4cfi* and *jit4cfi+*, we propose to enable BPF JIT constant blinding by default whenever x86_64 CFI is enabled. Currently, `bpf_jit_harden` defaults to 0, leaving user-provided BPF constants unchanged during JIT compilation. Both techniques use this property to place chosen operand bytes in JITed code and forge CFI hashes or `ENDBR64`. Setting `bpf_jit_harden=1` blinds constants from unprivileged users with random values while preserving their runtime semantics, preventing direct control over the operand bytes

required by the bypasses. We scope this change to x86_64 because the techniques rely on x86 instruction encoding and unaligned execution; the fixed-length, aligned instructions emitted by the ARM64 BPF JIT do not provide the same CFI-forgery primitive. We evaluated this mitigation in the same environment described in §VI-B and confirmed that both *jit4cfi* and *jit4cfi+* fail to bypass CFI when constant blinding is enabled.

To mitigate ret2enter we recommend extending the number of stack guard pages before stacks allocated in the vmalloc area. The 16-bit immediate can decrease RSP by at most `0xffff` bytes; the values pushed by `ENTER` can decrease it by up to another `0x100` bytes. So to fully mitigate ret2enter we propose adding an additional 0x10000 bytes to the existing vmalloc guard page, extending it from a single page to 0x11 pages. We confirmed that our proof of concept exploit no longer worked with the mitigation applied and that we were unable to discover any `ENTER` gadgets with a large enough frame size to pivot past the guard pages.

To mitigate pivot2init, we unmap the kernel-virtual addresses of the four regions whose physical pages are released to the buddy allocator at boot: the `__init` section, the rodata/data gap, the SMP-locks section, and the boot-time `brk` region. We confirmed that our pivot2init proof of concept no longer succeeds on the patched kernel.

The remaining techniques, pivot2init+, pivot2dma, pivot2vmalloc, pivot2espfix, pivot2cea+, and vmemmap bit flip, are challenging to fully mitigate. Mitigating them poses a similar challenge to mitigating ret2dir, a technique which still has not been fully mitigated nearly twelve years later. Even randomizing the locations of the DMA zones or CPU entry area data in physmap would only be a stopgap, since an attacker may still obtain their locations through side channels or pointer leaks. The core problem of these kinds of techniques is that userspace data as well as security critical data remains accessible in various memory regions from all execution contexts. To comprehensively mitigate these techniques some form of kernel memory isolation is necessary, such as using hardware features like x86_64 Memory Protection Keys to efficiently restrict access to pages containing user data and data that if corrupted may lead to privilege escalation.

VIII. Future Work

Applying our regions analysis to ARM64 and RISC-V is the natural next step, and §IX sets out what we anticipate it would and would not carry over. A second direction is the reverse of our analysis: the same per-region view that locates exploitation potential also indicates where a mitigation would be most effective, which is how we arrived at the patches of §VII.

IX. Limitations

We acknowledge four limitations of our study.

Scope of the study. Our regions analysis is comprehensive but not complete: we focus on the documented kernel memory regions and do not dive into their sub-sections or finer-grained divisions, and we restrict the threat model to unprivileged users under the configurations defined in §III.

Side-channel assumption. Our spray techniques rely on a side-channel attack to leak the base addresses of target memory regions. Such attacks are widely applicable on current machines, but the underlying side-channel vulnerabilities may be patched in future kernels or microarchitectures.

Architecture scope. We evaluate on x86_64 only, and the two halves of our work generalize differently. The regions analysis is methodology-driven, asking which allocation interfaces reach a region, whether its base is leakable, and what metadata it holds; those questions are equally well posed elsewhere, and ARM64 Linux keeps the same structures our spray techniques rest on, a linear map, a vmalloc area with `VMAP_STACK` stacks, and a vmemmap of `struct page`, so the region properties *pivot2vmalloc*, *pivot2dma*, and *vmemmap bitflip* exploit are present there as well, given a suitable leak. The escalations do not carry over. *ret2enter* needs `ENTER imm16, imm8`, which decrements the stack pointer by a 16-bit immediate in one instruction, and fixed-length RISC encodings offer no equivalent; *jit4cfi* and *jit4cfi+* further require entering JIT-emitted immediates at an unaligned offset, which such encodings forbid, and ARM64 enforces CFI through pointer authentication and BTI rather than kCFI hashes and `ENDBR64`. We therefore claim our escalation techniques for x86_64 alone. The methodology itself is not so restricted: applied to ARM64 or RISC-V it would examine that architecture's own regions and metadata, a surface our x86_64 study never encounters (§VIII).

CFI bypass stability. A randomization offset applied to JITed code keeps the success rate of *jit4cfi* and *jit4cfi+* around 20% per attempt. They also require the caller's call site to satisfy specific conditions (a forgeable kCFI hash for *jit4cfi*, attacker-controlled RBP or `[RBP+8]` for *jit4cfi+*), which limits the set of vulnerable call sites in practice.

X. Related Work

A. Kernel privilege escalation

Although generic methods for converting Control Flow Hijacking (CFH) into Local Privilege Escalation (LPE) are not common, prior work has discovered several, often subverting a mitigation in the process. ret2usr [23] executed shellcode directly from userspace memory until SMEP closed that path, after which ret2dir [12] reached the same data through its physmap alias, which at the time was neither randomized nor non-executable (§II-D1). Kepler [24] automates the chain from CFH by combining gadget discovery with usercopy-based canary leakage, while RetSpill [13] removes the need for register control by pivoting to user data already spilled on the kernel stack. System Register Hijacking [18] instead uses CFH to modify system registers, disabling mitigations or widening the attacker's capabilities.

B. Kernel heap exploitation

Converting a kernel heap corruption into a stronger primitive is itself an area of study. Zeng et al. [33] characterize what makes heap exploitation stable. PSpray [34] uses allocator timing side channels to predict heap layouts, while SLUBStick [35] uses them to make cross-cache reuse reliable. DirtyCred [36] and DirtyFree [37] convert small corruptions into privilege escalation by swapping credential objects, the latter also defeating `SLAB_VIRTUAL`, and ELOISE [5] discovers dynamically sized objects whose length fields leak out-of-bounds heap data. These techniques are complementary to ours: they stabilize the heap primitive that our region-based escalations then consume.

C. Memory Region Derandomization

Our spray techniques assume the region base is leakable (§II-C). Beyond the prefetch attack [8], KASLR has been broken by timing side channels including cache probing and double page faults [38], faulting TSX transactions [39], and contention on the processor's page-table walks via 4k-aliasing [40].

XI. Conclusion

In this paper, we performed a systematic study of the Linux kernel's memory regions from an exploitation perspective, discovered 10 novel exploitation techniques, showed that most of them carry real-world CVE exploits to root on every current longterm kernel release, against both Default and CFI-Hardened Linux kernels, revealed that both x86_64 kernel CFI implementations, kCFI and FineIBT up to its paranoid mode, can be generically bypassed by an unprivileged user, and proposed mitigations against the most impactful techniques. Beyond these specific findings, our regions analysis provides a systematic methodology that generalizes to future kernel security research.

XII. Ethics Considerations

Disclosure process. Publishing exploitation techniques can help attackers, and we accept that this work carries that risk. We therefore treated the techniques as sensitive from the outset and consulted the Linux kernel security team before publication. We privately reported the affected configurations, the techniques, our proof-of-concept results, and candidate mitigations we had implemented and tested to the Linux kernel security team, and asked it to choose the scope of each fix. The team approved public disclosure and recommended continuing on the kernel hardening mailing list; we published only afterwards, and are now working there with the community on mitigation patches. This paper discloses no new vulnerability and no unpatched CVE: every CVE in our evaluation was public and patched in all supported LTS releases before our study.

Redaction. We agree that implementation details enabling immediate exploitation should be withheld[2] until mitigations exist, and we apply that policy selectively, since what an attacker gains differs sharply across our techniques. *jit4cfi*, *jit4cfi+*, *ret2enter*, and *pivot2init* each supply a capability no public technique offers: the first two are, to our knowledge, the only generic bypasses of the kernel's CFI implementations, and the latter two escalate a hijacking primitive without the register control or extra address leak prior stack pivots require. For these we mask the implementation details wherever the mitigation has not yet landed, publishing them once a fix is available, and we continue working to land those mitigations. The remaining spray techniques exploit regions not previously studied for this purpose, but we do not believe they give an attacker a meaningful advantage over what is already public: *pivot2dma*, *pivot2cea+*, and *pivot2init+* share the prerequisites of *ret2dir* [12], namely a kernel image base leak, register control, and a physmap base leak, and reach the same escalation, while *pivot2espfix* and *vmemmap bitflip* demand more (Table II). *pivot2init* is the exception, needing no physmap leak, which is why it belongs to the masked group above. Disclosing the rest therefore seems unlikely to supply a capability the literature does not already offer, while withholding them would leave defenders without the region-level understanding needed to reason about the class. This matches how the kernel handles such issues: the security team treats hardening problems that are not tied to a specific vulnerability as public matters, directing them to the hardening mailing list precisely because open discussion is what gets them reviewed and patched, whereas an embargo on a defect with no fix in sight tends to delay the fix. Our report followed that route, and we pursue mitigations for this group as well, but we do not redact it.

Residual risk. Our techniques are primitive escalations rather than vulnerabilities, each presuming an attacker who already holds a control-flow hijacking primitive or an arbitrary write from a separate bug, but this does not exempt them from care: they make future memory-corruption bugs substantially easier to exploit. The spray techniques that rely on hardware side channels (§IX) are the sharpest case, since their root cause lies below the kernel and no patch we propose eliminates them. We recognize that disclosing them leaves a risk we cannot close, and we weighed it against the alternative: defenders cannot reason about a class of attack whose mechanism is undisclosed, while attackers who find it independently lose nothing. We therefore describe the mechanism and the residual risk.

[2]We have hashed the exploitation code used in the evaluation of the withheld techniques and will release the code after the corresponding mitigation patches land.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful feedback.

This material is based upon work supported by the Air Force Office of Scientific Research (award FA9550-24-1-0204); the Advanced Research Projects Agency for Health (ARPA-H) (contract SP4701-23-C-0074); and the National Science Foundation (awards 2146568 and 2232915). Additional support came from the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare

Center Pacific (NIWC Pacific) (contract N66001-22-C-4026), and from DARPA (contract 140D0424C0044). We also acknowledge the generous support of the US Department of Defense. This work relates to Department of Navy award N00014-24-1-2193 issued by the Office of Naval Research.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Air Force, ARPA-H, DARPA, NIWC Pacific, or the Office of Naval Research.

REFERENCES

[1] "ret2jit in CVE-2025-21700 kernelCTF exploit," 2025, accessed: 2026-04-19. [Online]. Available: https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2025-21700_lts_cos_mitigation/docs/novel-techniques.md

[2] "pivot2cea in CVE-2023-3776 kernelCTF exploit," 2023, accessed: 2026-04-23. [Online]. Available: https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2023-3776_lts/docs/exploit.md#put-payload-in-fixed-kernel-address-cve-2023-0597

[3] Google, "kernelCTF exploit proofs of concept," 2026, accessed: 2026-04-11. [Online]. Available: https://github.com/google/security-research/tree/master/pocs/linux/kernelctf

[4] Linux Kernel Developers, "Virtual memory map for x86-64 Linux," 2026, accessed: 2026-04-12. [Online]. Available: https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

[5] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1165–1184. [Online]. Available: https://doi.org/10.1145/3372297.3423353

[6] R. Wang, K. Chen, C. Zhang, Z. Pan, Q. Li, S. Qin, S. Xu, M. Zhang, and Y. Li, "AlphaEXP: An expert system for identifying Security-Sensitive kernel objects," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 4229–4246. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/wang-ruipeng

[7] E. Avllazagaj, Y. Kwon, and T. Dumitras, "SCAVY: Automated discovery of memory corruption targets in linux kernel for privilege escalation," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 7141–7158. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/avllazagaj

[8] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 368–379. [Online]. Available: https://doi.org/10.1145/2976749.2978356

[9] W. Liu, J. Ravichandran, and M. Yan, "Entrybleed: A universal kaslr bypass against kpti on linux," in *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 10–18. [Online]. Available: https://doi.org/10.1145/3623652.3623669

[10] T. Gleixner, "x86/kpti: Kernel Page Table Isolation," 2017, accessed: 2026-10-30. [Online]. Available: https://lwn.net/Articles/740608/

[11] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "Kaslr is dead: Long live kaslr," in *Engineering Secure Software and Systems*, E. Bodden, M. Payer, and E. Athanasopoulos, Eds. Cham: Springer International Publishing, 2017, pp. 161–176.

[12] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 957–972. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kemerlis

[13] K. Zeng, Z. Lin, K. Lu, X. Xing, R. Wang, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Retspill: Igniting user-controlled data to burn away linux kernel protections," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 3093–3107. [Online]. Available: https://doi.org/10.1145/3576915.3623220

[14] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, Nov. 2009. [Online]. Available: https://doi.org/10.1145/1609956.1609960

[15] J. Moreira, S. Rigo, M. Polychronakis, and V. P. Kemerlis, "DROP THE ROP: Fine-grained control-flow integrity for the Linux kernel," Black Hat Asia white paper, 2017. [Online]. Available: https://www.blackhat.com/docs/asia-17/materials/asia-17-Moreira-Drop-The-Rop-Fine-Grained-Control-Flow-Integrity-For-The-Linux-Kernel-wp.pdf

[16] P. Zijlstra, "x86/ibt: Implement FineIBT," 2022, accessed: 2026-10-30. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=931ab63664f0

[17] A. J. Gaidis, J. Moreira, K. Sun, A. Milburn, V. Atlidakis, and V. P. Kemerlis, "Fineibt: Fine-grain control-flow enforcement with indirect branch tracking," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 527–546. [Online]. Available: https://doi.org/10.1145/3607199.3607219

[18] J. Miller, M. Ghandat, K. Zeng, H. Chen, A. H. Benchikh, T. Bao, R. Wang, A. Doupé, and Y. Shoshitaishvili, "System register hijacking: Compromising kernel integrity by turning system registers against the system," in *34th USENIX Security Symposium (USENIX Security 25)*. Seattle, WA: USENIX Association, Aug. 2025, pp. 7427–7446. [Online]. Available: https://www.usenix.org/conference/usenixsecurity25/presentation/miller

[19] P. Zijlstra, "x86/ibt: Add paranoid FineIBT mode," 2025, accessed: 2026-10-30. [Online]. Available: https://github.com/torvalds/linux/commit/97e59672a9d2aec0c27f6cd6a6b0edfdd6e5a85c

[20] ——, "x86/mm: Randomize per-cpu entry area," 2022, Linux kernel commit 97e3d26b5e5f (fix for CVE-2023-0597); accessed: 2026-04-25. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=97e3d26b5e5f371b3ee223d94dd123e6c442ba80

[21] leave (kqx), "zenerational-aura — corCTF 2025," 2025, accessed: 2026-04-25. [Online]. Available: https://kqx.io/writeups/zenerational/#the-exploit

[22] S. Jenkins, "Exploiting CVE-2022-42703: Bringing back the stack attack," 2022, accessed: 2026-10-30. [Online]. Available: https://projectzero.google/2022/12/exploiting-CVE-2022-42703-bringing-back-the-stack-attack.html

[23] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight kernel protection against Return-to-User attacks," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 459–474. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kemerlis

[24] W. Wu, Y. Chen, X. Xing, and W. Zou, "KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1187–1204. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/wu-wei

[25] B. Azad, "An iOS hacker tries Android," 2020, Google Project Zero. Accessed: 2026-04-12. [Online]. Available: https://projectzero.google/2020/12/an-ios-hacker-tries-android.html

[26] Qualys Research Team, "Sequoia: A deep root in Linux's filesystem layer (CVE-2021-33909)," 2021, accessed: 2026-04-12. [Online]. Available: https://www.qualys.com/2021/07/20/cve-2021-33909/sequoia-local-privilege-escalation-linux.txt

[27] R. Bastide, "A kernel stack use-after-free: Exploiting NVIDIA's GPU Linux drivers," 2025, accessed: 2026-04-12. [Online]. Available: https://blog.quarkslab.com/nvidia_gpu_kernel_vmalloc_exploit.html

[28] L. Maar, L. Giner, D. Gruss, and S. Mangard, "When good kernel defenses go bad: Reliable and stable kernel exploits via Defense-Amplified TLB Side-Channel leaks," in *34th USENIX Security Symposium (USENIX Security 25)*. Seattle, WA: USENIX

Association, Aug. 2025, pp. 7447–7465. [Online]. Available: https://www.usenix.org/conference/usenixsecurity25/presentation/maar-kernel
[29] "ret2enter in CVE-2026-31419 kernelCTF exploit," 2026, accessed: 2026-10-30. [Online]. Available: https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2026-31419_cos/docs/novel-techniques.md
[30] "pivot2init in CVE-2025-38477 kernelCTF exploit," 2026, accessed: 2026-10-29. [Online]. Available: https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2025-38477_cos/docs/novel-techniques.md
[31] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1675–1689. [Online]. Available: https://doi.org/10.1145/2976749.2978406
[32] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds. Cham: Springer International Publishing, 2016, pp. 300–321.
[33] K. Zeng, Y. Chen, H. Cho, X. Xing, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Playing for K(H)eaps: Understanding and improving linux kernel exploit reliability," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 71–88. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zeng
[34] Y. Lee, J. Kwak, J. Kang, Y. Jeon, and B. Lee, "Pspray: Timing Side-Channel based linux kernel heap exploitation technique," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6825–6842. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/lee-yoochan
[35] L. Maar, S. Gast, M. Unterguggenberger, M. Oberhuber, and S. Mangard, "SLUBStick: Arbitrary memory writes through practical software Cross-Cache attacks within the linux kernel," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4051–4068. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/maar-slubstick
[36] Z. Lin, Y. Wu, and X. Xing, "Dirtycred: Escalating privilege in linux kernel," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1963–1976. [Online]. Available: https://doi.org/10.1145/3548606.3560585
[37] Y. Lee, H. Kwon, and T. Holz, "DirtyFree: Simplified data-oriented programming in the linux kernel," in *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, 2026.
[38] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 191–205.
[39] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with intel tsx," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 380–392. [Online]. Available: https://doi.org/10.1145/2976749.2978321
[40] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Binoculars: Contention-Based Side-Channel attacks exploiting the page walker," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 699–716. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhao-zirui

APPENDIX A
GENERATIVE AI TOOL USAGE

We used Anthropic Claude 4.8 to assist with evaluation data collection, prepare graphs and tables, generate C probe programs that trigger specific functions in the Linux kernel, and improve writing. All technical contributions, including the methodology, the new exploitation techniques, the analysis of kernel memory regions, the proof-of-concept exploits, and the interpretation of evaluation results, are the original work of the authors. All AI-assisted artifacts were reviewed and validated by the authors before inclusion.

TABLE VI

JITED CBPF CODE ENABLING KCFI BYPASS. OUR TECHNIQUE BYPASSES KCFI MITIGATION FOR MORE THAN 10,000 PROTECTED FUNCTIONS (APPROXIMATELY 14%), WHILE REQUIRING ONLY A SINGLE TARGET FUNCTION IN PRACTICAL EXPLOITATION SCENARIOS. [*] *Match Type*: *Any* FOR SINGLE-BYTE OPCODES ($L = 1$) AND *Substring* FOR MULTI-BYTE OPCODES ($L > 1$). BOTH TYPES ARE GOVERNED BY THE SAME MATCHING CONDITION (§V-A); *Any* IS THE $L = 1$ SPECIAL CASE. [†] STATISTICS ARE BASED ON A RECENT LINUX KERNEL LTS VERSION (6.12.63) WITH KCTF CONFIG PLUS KCFI REQUIRED CONFIGURATION. [‡] MULTIPLE CBPF INSTRUCTIONS GENERATE IDENTICAL JITED BYTE PATTERNS. ONLY ONE INSTRUCTION IS SHOWN PER UNIQUE PATTERN. SPECIFICALLY: `LD|IMM` GENERATES THE SAME PATTERN AS `RET|K`, AND `LDX|B|MSH` GENERATES THE SAME PATTERNS AS `LD|ABS`. [§] `LD|ABS, K>=0` GENERATES MULTIPLE INSTRUCTIONS WITH CONTROLLED 32-BIT OPERAND IN THE JITED CODE, NUMBERED #1-#3 TO INDICATE DISTINCT INSTRUCTIONS.

cBPF Instruction	**JITed Code**	**JITed Bytes**	**Match Type** [*]	**Matched Hashes** [†]
ALU Operations				
`ALU | OR | K`	`OR EAX, IMM32`	`\x0D`	Any	1034
`ALU | ADD | K`	`ADD EAX, IMM32`	`\x05`	Any	1008
`ALU | SUB | K`	`SUB EAX, IMM32`	`\x2D`	Any	912
`ALU | AND | K`	`AND EAX, IMM32`	`\x25`	Any	1220
`ALU | XOR | K`	`XOR EAX, IMM32`	`\x35`	Any	882
`ALU | MUL | K`	`IMUL EAX, EAX, IMM32`	`\x69\xC0`	Substring	420
`ALU | {DIV,MOD} | K`	`MOV R11, IMM32`	`\x49\xC7\xC3`	Substring	229
Load/Return Instructions [‡§]				
`RET | K`	`MOV EAX, IMM32`	`\xB8`	Any	1829
`LDX | IMM`	`MOV R13D, IMM32`	`\x41\xBD`	Substring	291
`LD | W | IND`	`ADD RCX, IMM32`	`\x48\x81\xC1`	Substring	356
`LD | ABS, K<0`	`MOV RCX, IMM32`	`\x48\xC7\xC1`	Substring	356
`LD | ABS, K>=0 #1`	`SUB RSI, IMM32`	`\x48\x81\xEE`	Substring	698
`LD | ABS, K>=0 #2`	`ADD RSI, IMM32`	`\x48\x81\xC6`	Substring	235
`LD | ABS, K>=0 #3`	`MOV ECX, IMM32`	`\xB9`	Any	927
Jump Instructions				
`JMP | {JEQ,JGE,JGT,JSET} | K`	`MOV ESI, IMM32`	`\xBE`	Any	881
Total (Deduplicated):				**10,305 (14%)**

TABLE VII

X86_64 KERNEL CFI MITIGATIONS AVAILABLE PER LTS RELEASE (LATEST POINT RELEASE AS OF AUGUST 2026); THE CFI-BYPASS EVALUATION (TABLE IV) IS SCOPED ACCORDINGLY.

Mitigation	**5.10**	**5.15**	**6.1**	**6.6**	**6.12**	**6.18**
kCFI	✗	✗	✓	✓	✓	✓
FineIBT	✗	✗	✗	✓	✓	✓
FineIBT-Paranoid	✗	✗	✗	✗	✗	✓