System Register Hijacking: Compromising Kernel Integrity By Turning System Registers Against the System

Jennifer Miller*, Manas Ghandat*, Kyle Zeng*, Hongkai Chen*, Abdelouahab (Habs) Benchikh*
Tiffany Bao*, Ruoyu Wang*, Adam Doupé*, Yan Shoshitaishvili*
*Arizona State University
{jmill,mghandat,zengyhkyle,hongkai.chen,abenchik,tbao,fishw,doupe,yans}@asu.edu

Abstract

The Linux kernel has been a battleground between security researchers identifying new exploitation techniques and those developing mitigations to protect the kernel from exploitation. This is an ongoing battle: last year, Google's KernelCTF Vulnerability Research Program paid out 44 bounties for unique exploitation techniques submitted to the program, many of which targeted control flow hijacking vulnerabilities. However, the era of control flow hijacking exploits in the kernel may be coming to an end: FineIBT, now the default Control Flow Integrity measure in the Linux kernel, blocks all known control flow hijacking exploitation techniques.

In this paper, we propose System Register Hijacking, a previously overlooked frontier in the exploitation of control flow hijacking vulnerabilities in the kernel context. Our approach provides a comprehensive examination of typically overlooked system registers, leading us to propose several powerful exploitation techniques targeting different x86-64 system registers (e.g., `cr0`, `cr3`, and `gs`) and aarch64 system registers (e.g., `pan`, `elr_el1`, and `vbar_el1`) to break kernel security in different ways. While all of our techniques present new avenues for attackers, one in particular, which leverages the x86-64 `swapgs` instruction, requires neither general purpose register nor stack control, making it one of the most powerful kernel exploitation primitives currently known. Moreover, to our knowledge, this is the first exploitation primitive capable of bypassing the FineIBT mitigation, demonstrating not only the power of our technique but also the continued relevance of control flow hijacking vulnerabilities.

In addition to developing these techniques, we propose mitigations to defend against most of them. Though some of our techniques appear challenging to mitigate, our `swapgs` mitigation restores FineIBT's security posture at a performance cost of just under 1%.

1 Introduction

Linux powers an increasingly large number of systems and devices worldwide: over 85% of smartphones, 40% of servers, and 90% of cloud infrastructure run Linux [33, 39]. This popularity makes Linux an increasingly interesting and profitable target for attackers. At the same time, the complexity of the Linux kernel means that it is never free from vulnerabilities. As of September 2024, 1,250 Linux kernel vulnerabilities have been published as CVEs [18], many of which would allow attackers to hijack control flow in Linux kernels upon successful exploitation.

Control-flow hijacking techniques can convert a memory corruption vulnerability into a powerful exploit. While finding and eliminating these vulnerabilities is important, it is also crucial to secure the Linux kernel to make the exploitation of these vulnerabilities more difficult. As a response, the development of defenses in the Linux kernel against exploitation has gained momentum in the past decade, leading to many mitigations being widely deployed to hamper successful and reliable exploitation efforts. Over time, defenders have proposed and developed many mitigations to thwart control-flow hijacking techniques: Mitigations like Supervisor Mode Execution Prevention (SMEP), Supervisor Mode Access Prevention (SMAP), Kernel Address Space Layout Randomization (KASLR), CR-Pinning, and NX-Physmap have raised the bar for successful exploitation of control flow hijacking.

However, despite the introduction of such exploit mitigations in the Linux kernel, attackers continue to discover new techniques to bypass them. To name a few well-known exploitation techniques: returning to user-space code (ret2usr [44]), stack pivoting to a user-space stack (pivot2usr [20]), returning to user-space code in "physmap" (ret2dir [36]), and reusing userspace data spilled to the kernel stack (retspill [62]). Because these techniques can fundamentally change the level of threat posed by vulnerabilities (e.g., by making previously unexploitable vulnerabilities exploitable), they are of significant interest to both the research and industry communities. One result of this interest is Google's KernelCTF Vulnerability Reward Program [25], which rewards security researchers based on novel applications of exploitation techniques rather than the underlying vulnerabilities themselves (that is, one can earn a kCTF re-

ward with a new exploitation technique on a known vulnerability). Over the last year, Google's kCTF has seen 44 unique successful exploits against their "mitigations" instance of the Linux kernel, which deploys even experimental mitigations (i.e., these mitigations do not exist or are not enabled by default in mainline kernels) to defend against popular attack vectors.

Several years ago, the Google Project Zero team published a Linux exploit that overwrote the `cr4` system register [23] to disable SMAP and SMEP, allowing kernel control flow to be redirected to userspace-controlled memory regions. Though it was recognized at the time as a powerful technique, we realized that it was actually a glimpse at a different style of thought about exploitation in the Linux kernel. Most exploits in the kernel look, at an instruction level, like userspace exploits: they use general-purpose registers to index kernel memory or pass arguments to internal kernel APIs (e.g., `commit_creds` or `copy_from_user`). However, the kernel uses additional registers that userspace does not: *System Registers*. These registers influence CPU features (such as SMAP and SMEP), memory structure (e.g., page tables, thread-local storage), and critical CPU operations (e.g., interrupt handlers). Most of these registers, such as `cr0` and `cr4`, are completely inaccessible from userspace. Some, such as `EFLAGS`, exist in userspace but have special meaning in kernel mode (the `AC` flag in `EFLAGS` disables SMAP). Driven by the insight that Google's use of `cr4` was not the only exploitation implication of system registers, we hypothesized that other system registers could be leveraged for similar effects.

In this paper, we present an in-depth analysis of System Register Hijacking (SRH), a novel class of exploitation techniques that bypasses many mitigations in the Linux kernel. We systematically evaluate all system register-accessing x86-64 and aarch64 instructions in the Linux kernel, analyze potential effects that attackers can achieve by using them in control flow hijacking, demonstrate the feasibility of these techniques (and their variants), understand their prerequisites, and measure their prevalence (in terms of the number of gadgets) in the kernel. Of our resulting seven exploitation techniques on x86-64 and aarch64 (the known `cr4` hijack described above, plus six new ones), four techniques require only register control at the control flow hijack site, two require some stack control, and one requires *neither* stack nor register control, representing a "worst-case scenario" exploitation technique in the Linux kernel and thus being applicable to *any* control flow hijacking vulnerability on x86-64. Moreover, this last technique, relying on the `swapgs` instruction and hijacking the `KERNEL_GSBASE_MSR` system register, *bypasses FineIBT mitigations*. To the best of our knowledge, our `swapgs` SRH technique is the first and only general technique applicable to exploiting control flow hijacking vulnerabilities in kernels with FineIBT enabled.

Contributions. In summary, our paper makes the following contributions:

- We propose System Register Hijacking, a category of exploitation techniques that leverage kernel-only system registers to achieve attacker goals. We also measure the prevalence of gadgets that can enable these techniques in many recent Linux kernel images.
- Furthermore, we demonstrate the viability of SRH techniques by developing PoCs based on vulnerable kernel modules and showcasing the effectiveness of the `swapgs` technique by exploiting real-world vulnerabilities, presenting the first-ever generic forward-edge bypass for the recent FineIBT mitigation.
- We propose mitigations for most of our SRH techniques and discuss the challenges inherent in mitigating these weaknesses.

2 Background

Before discussing the details of our attack, we provide a brief overview of the Linux kernel's security mechanisms and the architectural features that are relevant to our proposed SRH attacks.

2.1 Control Flow Hijacking

A common exploitation technique used to achieve Local Privilege Escalation in the Linux kernel is to use a memory corruption vulnerability to corrupt a function pointer or a pointer to a function table on the kernel heap. This enables the attacker to hijack the control flow of an indirect jump or call from its intended target to an attacker-controlled target, thus corrupting a forward control flow edge.

However, even having hijacked control flow, it can be difficult to maintain control for long enough to escalate privileges. The traditional approach for transferring forward control flow hijacking to a ROP chain in the Linux kernel involves pivoting the stack pointer to a controlled address on the kernel heap. This requires a stack-pivoting gadget such as a `pop rsp; ret;` gadget or a `mov rsp, <attacker-controlled-reg>; ret;` gadget. The *pivot-to-heap* technique can be difficult to implement in practice because the viable pivoting gadgets are dependent on the system state at the site of control flow hijacking. Therefore, we consider pivot-to-heap to have significantly complex preconditions.

For example, using the gadget `mov rsp, rax; ret;` has the preconditions that (1) `rax` holds the address of the attacker-controlled stack, and (2) the attacker controls enough memory at `rax` to implement the ROP payload.

ret2dir [36] found, at the time it was published, that the kernel's physical memory map (physmap) region was located at a fixed address and was entirely read-write-execute memory. At the time, the SMEP mitigation (which prevents redirecting

kernel control flow to user space) was still fairly new, and this technique presented a clear bypass of the mitigation by returning to a *synonym* address in the kernel's physmap corresponding to a userspace page containing a payload rather than the userspace address.

KEPLER [57] proposed a two-part attack that used control flow hijacking to induce a stack overflow. The technique involves first inducing a disclosure of the contents of the kernel stack by targeting a `copy_to_user` call. The disclosure would provide the value of the kernel's stack canary, which could then be used in a `copy_from_user`-based gadget to overflow the stack.

RetSpill [62] improved upon pivot-to-heap by relaxing the preconditions to successfully pivot control flow. The idea is that pivoting to controlled memory on the kernel stack has considerably easier-to-satisfy preconditions than pivoting to the heap. This technique is possible because of the existence of ROP gadgets that perform stack adjustments, such as `add rsp, x; ret;` and where attacker-controlled data is *spilled* to the kernel stack at `rsp+x`. Furthermore, this can be used as a short ROP chain to perform privilege escalation or to pivot to a longer ROP chain elsewhere.

2.2 Control Flow Integrity

The Linux kernel supports a software-based Control Flow Integrity (CFI) solution, called kCFI [45], which is intended to protect against forward control flow hijacking. kCFI assigns tags at compile time to functions and sites where indirect control flow occurs, then compares the tag of the call site with the tag of the target to ensure it is a valid target. Previous work [62] found that some indirect control flows under kCFI do not validate their targets, making kCFI an incomplete mitigation in that regard. Currently, kCFI is not enabled on major Linux desktop and server distributions, but it is enabled by Android.

Modern Intel x86-64 processors support hardware-based CFI protections: forward edge protection via *IBT* and backward edge protection via *Shadow Stack*. IBT works by ensuring that the target of forward edge control flow begins with an `endbr64` instruction and faults otherwise. The Linux kernel currently supports in-kernel IBT [3], providing sparse but forward edge CFI, and a finer-grained forward edge CFI called FineIBT [21, 63], which adds software checks to indirect call sites in addition to enabling IBT.

Shadow Stack works by maintaining a second stack of return addresses in memory that cannot be modified by normal memory accesses, and a fault occurs if the return address popped off the stack does not match the one popped off the shadow stack. Currently, there is no support for hardware-accelerated shadow stacks in the kernel. While recent work proposed a potential implementation [43], it has not been open-sourced.

There is an implementation of software-only shadow stacks for aarch64 in Clang [6] and GCC [2]. Previously, there was a version of Clang's *SoftwareCallStacks* for x86-64, but the Clang documentation clarifies that it was removed due to performance and security issues [6]. The aarch64 software call stacks feature is enabled by major distributions such as Ubuntu.

2.3 KASLR Bypasses

Similar to the userspace security feature ASLR, which randomizes the virtual addresses of memory regions, the Linux kernel also randomizes the virtual addresses of its various memory regions. KASLR has been shown many times to be extremely vulnerable to microarchitectural side channels on both x86-64 [17, 27, 55] and AArch64 [34]. Thus, for this work, we consider it irrelevant in our attack scenario of Local Privilege Escalation.

The Function Granular KASLR (FG-KASLR) mitigation [42] attempts to add more entropy such that side-channeling the base address of the kernel is not enough to fully bypass KASLR for the kernel text area. However, this mitigation has not been merged into the upstream kernel. Regardless, the latest FG-KASLR also fails to mitigate our proposed attacks because it does not randomize the entry point code's position relative to the base address of the kernel.

2.4 SMAP and PAN Bypasses

Modern x86-64 processors support a protection mechanism called SMAP (Supervisor Mode Access Prevention), and aarch64 processors support Privileged Access Never (PAN), which enforces a policy that the kernel is not allowed to access user space memory unless explicitly permitted. On x86-64, the kernel can temporarily access user memory by setting the Alignment Check (AC) bit in the EFLAGS register or disabling SMAP via a control register. On AArch64, the kernel can temporarily allow user accesses by writing zero to the PAN MSR.

The goal of these mechanisms is to prevent exploits that attempt to fool the kernel into using a forged data structure or stack that exists in user-controlled memory. Unfortunately, this mitigation can be bypassed using the Linux kernel's identity-mapped memory region, referred to as physmap, which is a mapping of all physical memory in the virtual address space. To bypass SMAP or PAN and have the kernel use user-controlled memory, an attacker must probe memory in physmap that corresponds to memory they control in user space. Thus, we consider this mitigation irrelevant insofar as the ability to locate user-controlled memory in physmap is a given in the context of Local Privilege Escalation.

2.5 Kernel Page Table Isolation

Kernel Page Table Isolation (KPTI) is a mitigation originally proposed to prevent Prefetch and Fault Timing side-channels [26]. KPTI attempts to increase the isolation between user space and kernel space by allocating separate page tables to be used in kernel and user mode. It was ultimately upstreamed in the Linux kernel as a software mitigation for Meltdown [40]. By default, the Linux kernel only enables KPTI on systems affected by Meltdown: systems with an Intel CPU from before 2018. Many of the recent transient execution-based side-channels, such as Retbleed [53], Phantom [54], and BHI [11], have also been unaffected by KPTI.

2.6 x86-64 FSGSBASE Extension

Modern x86-64 CPUs support an extension called FSGSBASE [31], which adds instructions for directly reading and modifying the `FSBase` and `GSBase` registers. Notably, this extension also provides instructions for unprivileged use in user applications. Prior to FSGSBASE, the `arch_prctl` syscall, which performs integrity checks, was the primary method of writing the `GSBase` and `FSBase` registers. With the FSGSBASE extension enabled, as it is on modern Linux systems, there are no integrity checks on what value is written to either `GSBase` or `FSBase` via the `wrgsbase` or `wrfsbase` instructions, respectively. The capability to arbitrarily set the `GSBase` register has led to security issues in x86-64-based operating systems in the past [32].

2.7 x86-64 Per-CPU Variables

The Linux kernel uses the x86-64 `gs` segment for accessing per-cpu variables, using instructions such as `mov rax, gs:0x28`. Accesses that use offsets from the `gs` segment register are calculated based on the value of the `GSBase` register. Important values and structures are stored in per-cpu variables, such as the current thread's stack address, stack canary, and task struct (which stores the current thread's credentials and other security-relevant data). As mentioned in Section 2.6, the `GSBase` register is also available for use in user applications via `wrgsbase`. When entering and exiting the kernel, the `swapgs` instruction is used to switch between the user `gsbase` and kernel `gsbase`.

2.8 Privileged Instructions for ROP

The use of privileged instructions in ROP is not itself novel; however, in modern kernel exploitation, only a few privileged instructions are ever targeted. In the past, Google's Project Zero proposed an attack that would disable SMEP and SMAP by returning to `native_write_cr4` [23]; however, this technique was later mitigated via CR-Pinning. With this gadget mitigated, new techniques involving other privileged instructions have not been observed. The instructions `swapgs`, `popf`, `iret`, and `sysret` are the only system instructions commonly seen in modern ROP chains. Their use is entirely focused on gracefully terminating a ROP chain by switching back to user mode rather than expanding the capabilities of the exploit.

2.9 System Registers

General-purpose registers (GPRs) and system registers are integral components of a CPU's architecture, serving distinct but complementary roles. General-purpose registers are versatile, high-speed storage locations within the CPU that hold data temporarily during the execution of instructions. They are used for a wide range of operations, such as arithmetic, logic, and data manipulation. Typically, these registers are directly accessible by the CPU's instruction set, allowing for efficient processing of data. System registers, on the other hand, are specialized registers that control or monitor the CPU's internal operations, such as managing system state, configuration, and control of hardware resources. They often include status registers, control registers, and other special-purpose registers that are crucial for system management tasks, such as interrupt handling, memory management, and operating mode configuration.

3 Threat Model

In our threat model, we consider three different attack scenarios:

Base. The Base scenario is targeting a Linux kernel with common mitigations deployed, such as: SMEP, SMAP, KPTI, NX-physmap, CR Pinning, STATIC_USERMODE_HELPER, RANDKSTACK, and STACK CANARY. This set of mitigations is modeled after stock desktop kernel configurations, such as Ubuntu's kernel configuration.

Due to the rapid and active development of control-flow integrity (CFI) schemes in the Linux kernel, described in Section 2.2, the deployment of CFI schemes to stock kernels is imminent. In this research, besides stock kernels, we also explore the security impact of system registers on CFI-enabled kernels.

FineIBT-Protected. The kernel has all mitigations enabled in the Base scenario in addition to FineIBT.

kCFI-Protected. The kernel has all mitigations enabled in the Base scenario in addition to kCFI.

4 System Register Hijacking

We propose a class of techniques, called *System Register Hijacking*, in which a control flow hijacking primitive is used to execute an instruction that modifies a security-sensitive system register. In studying the x86-64 instruction set, we found that system registers are typically security-sensitive for one of two reasons: either they control a security feature's status,

or they control the address of a security-critical structure in memory.

Hijacking registers that control security features may temporarily or permanently disable that feature on a given CPU thread, making future steps in an exploit easier. An example is the `cr4` register, which can permanently disable several security features, including SMEP and SMAP, on the CPU thread.

Compromising registers that control the addresses of structures can be used to redirect their address to a forged structure, leading to capabilities that depend on the structure in question. These structures may contain either code or data addresses that the system uses during execution. An example of a structure register is the IDTR (Interrupt Descriptor Table Register), which contains the address of the IDT, a structure that specifies what code addresses to use when various interrupts, traps, and exceptions occur. If an attacker can modify the address stored in the IDTR, they can control the descriptors and point them to any code address of their choosing.

To identify the instructions capable of modifying security-sensitive system registers on x86-64, we first identified the existing system registers by referencing the Intel SDM Volume 3A, Section 2.1.6, "System Registers" [30]. We then narrowed down which of those registers relate to security features or structure addresses. Finally, we identified which instructions can be used to modify the security-relevant portions of those registers. For example, the `lmsw` instruction can only modify the lower 16 bits of `cr0`, none of which affect security-related features. From this, we identified 15 instructions that modify security-sensitive registers, which can be seen in the first column of Table 1.

For aarch64, we had to take a different approach. There are many different versions of aarch64; rather than choosing one implementation to support, we support the set of system registers used by the Linux kernel. We gathered this set of registers by scanning several kernel images for occurrences of the `MSR` instruction, which is used when writing system registers in aarch64. From the set of system registers written to by the kernel, we consulted ARM architecture manuals [8] to identify which of those registers are security-sensitive and can be written from EL1 (Kernel Mode). We identified eight instructions that modify security-sensitive registers, which can be seen in the first column of Table 1.

5 Measurement

To evaluate the prevalence of potential gadgets that modify system registers in a typical Linux kernel, we wrote a script on top of the Rust bindings for the Capstone Engine disassembly library to collect all occurrences of instructions that modify Security-Sensitive System Registers for a given kernel image. The script accepts memory dumps of the `.text` section from a running kernel and scans through them, attempting to disassemble the data at every offset of the section, then performing a string comparison on each disassembled instruction to determine if it matches an instruction capable of modifying a Security-Sensitive System Register. We purposefully made this analysis simple so as not to miss gadgets that existing ROP gadget finders may exclude due to their lengths or conditional control flow.

Measuring the number of potential gadgets in a kernel image is non-trivial due to the complexity of the kernel's self-patching mechanisms. The most impactful self-patching feature with respect to our analysis is "alternatives," which allows for CPU feature-dependent instructions to be used only when the necessary features are available. This means that the `.text` sections of kernel images do not contain instructions like `stac` and `clac` since those are dependent on the SMAP CPU feature being available; instead, they get patched in at runtime. Statically applying alternatives is possible by using the `.altinstructions` section of the kernel image but requires source code parsing and reimplementing the kernel's logic for applying alternatives [48]. Rather than trying to accurately apply alternatives and other self-patching mechanisms statically in our analysis, we opted to use memory dumps of the `.text` section, which already have the alternative instructions applied. We used QEMU (version 6.2.0) with the `-cpu max` architecture for a consistent set of CPU features.

We performed a large-scale evaluation of the presence of these gadgets in four kernel builds: Ubuntu Generic, Ubuntu AWS, Fedora Core, and Fedora Enterprise Linux Next, for the five most recent major versions of each.

5.1 Measurement on x86-64

The results of our measurement of potential x86-64 gadgets are located in Table 5. For this analysis, we include instructions that can modify `EFLAGS.AC`, which controls whether or not the SMAP feature is enforced. We find an extremely high number of occurrences of the `popf` instruction due to its single-byte opcode. However, many gadgets containing `popf` are invalid due to being misaligned, resulting in an invalid opcode exception when executed.

We observed a disproportionate number of `wrmsr` gadgets in Fedora Enterprise Linux Next (ELN) builds. Upon inspection, unlike the other kernel builds, ELN builds contain many instances of the same functions. For example, the `native_write_msr` function, which occurs once in other builds, has ten distinct occurrences in the 6.11 ELN build. This code duplication is likely the cause of the increased number of `wrmsr` gadgets.

5.2 Measurement on aarch64

The results of our measurement of potential aarch64 gadgets are located in Table 7. We find significantly fewer potential gadgets on aarch64 overall due to the 4-byte instruction alignment requirement of the architecture.

Instruction/Gadget	**Register**	**Feature/Structure**	**Precond**
x86-64			
mov cr0	cr0	WP	
mov cr0, rax; mov rax, rbp; popf; pop r15; ret			RC + SC
mov cr4	cr4	SMEP/SMAP/UMIP/CET/MPK	
mov cr4, rbx; je (taken); jmp r8			RC
popf	EFLAGS	SMAP*	
popf; ret			SC
lidt	idtr	IDT	
lidt [rdi]; xor edi, edi; ret			RC
lgdt	gdtr	GDT	
lgdt [rdi]; xor edi, edi; ret			RC
swapgs	MSR_GSBASE	Per-cpu Variables	
swapgs; lfence; ret;			None
mov cr3	cr3	Page Tables	
mov cr3, r9; push r8; ret;			RC
wrmsr	MSR_EFER + MSR_GSBASE + MSR_FSBASE	Per-cpu Variables	
wrmsr; ret			RC
iretq	EFLAGS	SMAP*	
iretq			SC
wrgsbase	MSR_GSBASE	Per-cpu Variables	
wrgsbase rax; jmp; jmp; jmp; ret			RC
wrfsbase	MSR_FSBASE	Per-cpu Variables	
stac	EFLAGS	SMAP*	
clac	EFLAGS	SMAP*	
ltr	tr	TSS	
ltr word ptr [rbx + 0x5d]; ret			RC
lldt	ldtr	LDT	
lldt ax; ret			RC
aarch64			
msr elr_el1	elr_el1	elr_el1	
msr elr_el1, x2; msr spsr_el1, x1; ret			RC
msr pan	pan	PAN	
msr pan, #0; ret			RC
msr sctlr_el1	sctlr_el1	MTE/EPAN	
msr elr_el2, x0; ret			RC
msr spsr_el1	spsr_el1	TCO/PAN/UAO	
msr spsr_el1, x1; nop; nop; ret			RC
msr tcr_el1	tcr_el1	MTE	
msr spsr_el1, x1; nop; nop; ret			RC
msr ttbr0_el1	ttbr0_el1	ttbr0_el1	
msr ttbr0_el1, x0; isb ; msr daif, x1; ret			RC
msr ttbr1_el1	ttbr1_el1	ttbr1_el1	
msr ttbr1_el1, x0; isb ; msr daif, x1; ret			RC
msr vbar_el1	vbar_el1	vbar_el1	
msr vbar_el1, x0; ret			RC

Table 1: Security-Sensitive System Register modifying instructions identified on x86-64 and aarch64, which mitigations or structures they control, and the preconditions necessary (SC is Stack Control, RC is Register Control). Representative gadgets for each instruction where valid gadgets that are short enough to display were present are listed beneath the associated instruction. *The `iret`, `popf`, `stac`, and `clac` instructions are able to modify the `AC` bit in the `EFLAGS` register which controls the status of SMAP.

The most common gadgets are those related to `ttbr0_el1` and `ttbr1_el1`, which are the page table registers for kernel mode. Interestingly, there are very few of these gadgets present in the Fedora ELN builds compared to the other builds. Reviewing the sources of these gadgets, it appears that, unlike ELN, other kernel builds are inlining calls to user-memory accessor functions such as `copy_from_user`, which contain instructions that write to `ttbr0_el1` and `ttbr1_el1`.

5.3 Gadget Verification

To understand how many of these potential gadgets are valid for exploitation purposes, we ran angrop [7], a symbolic execution-based ROP gadget analyzer, on all of the potential gadgets. We added validation passes that run after angrop to remove cases that are valid ROP gadgets, in that they return, but do not result in control of the system register being targeted. This accounts for cases where gadgets contain multiple writes of different values into the same system register, nullifying the first write to the register. In such cases, there would be a valid gadget starting at the last write to the register, but we would consider the gadget containing both writes to be invalid.

Since angrop [7] is based on angr [51], which does not have accurate modeling of system instructions, it is likely that this analysis has false negatives. For example, the `stac` and `clac` instructions are not currently supported in angr, so symbolic execution halts at those instructions. To work around this, we patched angr's intermediate representation to support those instructions, but there may be other cases we did not account for. Unsurprisingly, angrop did not recognize syscall entry points as valid gadgets, but as we will discuss in Section 6, they are very strong gadgets. We added pattern matching to detect code locations that perform a `swapgs` followed by a write of the stack pointer from a `gs`-relative address. These gadgets are included in the `swapgs` gadgets in the gadget validation numbers.

On x86-64, we consider all `iret` gadgets to be valid because the instruction both pops the EFLAGS register off the stack and returns. Additionally, we consider all `clac` gadgets invalid because, while they can modify the `EFLAGS.AC` bit, they can only clear it. This means `clac` gadgets can never disable SMAP enforcement, only re-enable it. Similarly, on aarch64, we remove gadgets that write one to the `PAN` MSR because they cannot be used to disable the security feature.

This verification analysis is intended to represent a lower bound on the number of valid gadgets, while the measurement results represent the upper bound. Regardless of false negatives, we find that many valid gadgets are present across all the kernel builds we analyzed. The results of gadget verification on x86-64 and aarch64 can be found in Table 6 and Table 8, respectively. Concerningly, this includes gadgets for modifying `cr4` and `cr0` on x86-64, which are meant to be protected by the CR-Pinning mitigation, as well as gadgets capable of disabling PAN on aarch64.

```
entry_SYSCALL_64:
<+0>:     endbr64
<+4>:     swapgs
<+7>:     mov    QWORD PTR gs:0x6014,rsp
<+16>:    jmp    <entry_SYSCALL_64+36>
<+18>:    mov    rsp,cr3
<+21>:    nop
<+26>:    and    rsp,0xffffffffffffe7ff
<+33>:    mov    cr3,rsp
<+36>:    mov    rsp,QWORD PTR gs:0x32c98
```

Figure 1: A snippet of the disassembly of the syscall entry point.

6 System Register Hijacking Techniques

Based on our analysis of system registers and gadgets that allow their modification, we derive new forward control flow hijacking techniques and find new gadgets for the known `cr4` hijacking technique. The preconditions of the techniques we propose vary depending on the instruction being targeted and the specific instances of the gadget. We only propose techniques for which there are gadgets with few register dependencies that may be met by controlled arguments at the control flow hijacking site or satisfied in combination with a stack pivot to a ROP chain.

6.1 x86-64: swapgs Stack Pivoting

The `swapgs` instruction is commonly found in entry, exit, and other interrupt handling code in the kernel. It swaps the values of two system registers: the `GSBase` register and the `KernelGSBase` register. By setting the `GSBase` register in user mode through the `wrgsbase` instruction (which is accessible to user mode code), the `KernelGSBase` will be set to the userspace value on kernel entry when the `swapgs` instruction, seen in Figure 1, executes.

The result of executing a `swapgs` gadget is that the memory backing per-cpu variables can be redirected to an attacker-controlled address. Some `swapgs` gadgets, such as `swapgs; ret`, are impractical to use because the stack canary is stored in a per-cpu variable, so if the caller of the gadget returns, it will likely fail the stack check and panic the kernel.

We found that in many gadgets, the `swapgs` instruction was followed by a write to the stack pointer register of a value read from a per-cpu variable. This is often done on kernel entry from user mode to switch to the kernel stack. Hijacking execution to one of these locations will result in a stack pivot when the stack pointer register is set based on the controlled per-cpu variables.

Notably, these gadgets require no general-purpose register control, relying only on the malicious `GSBase` register, and

will function for any control flow hijacking primitive that executes in the context of the thread where the `wrgsbase` was performed.

For this technique to work, the attacker must be able to forge a per-cpu structure in the kernel address space for `GSBase` to point to. We satisfy this requirement either by leaking the address of a controlled page or by Transparent Huge Page (THP) spraying. THPs are 2 MB aligned in physical memory, allowing a user to fill a large range of the kernel's physical memory map with controlled pages. The ability to make a stable guess of the address of a controlled page in the physical memory map is also reliant on knowing the base address of the kernel's physical memory map, which is obtainable by either leaks or side-channels [17, 28] (as mentioned in Section 2).

For the technique to be successful, the forged `GSBase` structure must contain at least the kernel stack pointer and a pointer at the offset associated with the task struct to avoid potential double faults in the page fault handler. We find that a stable method for this is to purposely cause the code to trigger a page fault on unmapped memory and overwrite the stack in the page fault handler near the interrupt return. The stack will be overwritten as the page fault handler executes, leading to the return to the handler being hijacked into executing a user-specified ROP chain. The ROP chain will have to switch back to the original `GSBase` value using a `swapgs; ret` gadget, then perform the necessary steps for privilege escalation, e.g., `commit_creds(init_cred)`, and finally return control flow to userspace using `iret` or `sysret`.

6.2 x86-64: CR0 Hijacking

The `cr0` system register contains important bit fields that are used to enable paging, write protection, and other important kernel features. A `cr0` gadget can be used to bypass the Write Protect (WP) mitigation, permitting writes to any memory region that is accessible to the kernel. This includes inherently read-only regions such as the kernel text section. The CR-Pinning mitigation, which mitigated `cr4` hijacking, was eventually extended to protect the `native_write_cr0` function as well [23], intending to prevent an attacker from disabling the mitigation. As far as we can tell, no exploits have ever made use of CR0 hijacking due to the preemptive mitigation of `native_write_cr0`. However, we found that there are still gadgets that can be used to control `cr0` outside the mitigated function. An example is shown in Figure 2.

6.3 x86-64: CR4 Hijacking

In our analysis, we found several occurrences of the `mov cr4` instruction outside the mitigated `native_write_cr4` functions.

An example gadget that can control `cr4` in existing kernels is found in the assembly function `sev_verify_cbit`, as seen in Figure 3. With this gadget, an attacker can permanently disable SMEP, SMAP, and all other security features controlled by `cr4` on the executing CPU thread. This type of gadget enables a two-step exploitation process, where the attacker first disables security features and then returns to user space to execute a ROP chain, as demonstrated by Google Project Zero [23].

```
virtual_mapped:
<+35>:  mov    cr0,rax
<+38>:  mov    rax,rbp
<+41>:  popf
<+42>:  pop    r15
<+44>:  pop    r14
<+46>:  pop    r13
<+48>:  pop    r12
<+50>:  pop    rbp
<+51>:  pop    rbx
<+52>:  ret
```

Figure 2: The disassembly of a `mov cr0` gadget found in an Ubuntu 22.04 kernel image we analyzed.

```
sev_verify_cbit:
<+69>:  mov    cr4,rsi
<+72>:  je     sev_verify_cbit+87
<+74>:  xor    rsp,rsp
<+77>:  sub    rsp,0x1000
<+84>:  hlt
<+85>:  jmp    sev_verify_cbit+84
<+87>:  mov    rax,rdi
<+90>:  jmp    __x86_return_thunk
```

Figure 3: The disassembly of a `mov cr4` gadget found in an Ubuntu 22.04 kernel image we analyzed.

6.4 x86-64: popf Extension + RetSpill

The `popf` instruction is capable of setting the `AC` bit in the `EFLAGS` register, which is the bit set by the kernel to temporarily prevent SMAP checking when it executes functions such

```
ret_to_kernel:
<+12>:  msr    elr_el1, x21
<+16>:  msr    spsr_el1, x22
<+20>:  ldp    x0, x1, [sp]
<+24>:  ldp    x2, x3, [sp, #16]
...
<+72>:  ldp    x26, x27, [sp, #208]
<+76>:  ldp    x28, x29, [sp, #224]
<+80>:  ldr    x30, [sp, #240]
<+84>:  add    sp, sp, #0x150
<+88>:  nop
<+92>:  eret
```

Figure 4: The disassembly of the `ret_to_kernel` gadget found in an Ubuntu 22.04 kernel image we analyzed.

as `copy_from_user`. Despite this powerful capability, the instruction has not seen use in exploits outside of Capture The Flag competitions [60].

A recent technique, RetSpill [62], proposed using data spilled to the kernel stack as ROP gadgets, leveraging a stack adjustment gadget to pivot the stack to those registers. A downside of this technique is that the amount of controlled data on the stack to pivot to is fairly constrained: the paper claims only 11 gadgets on average. This is often enough to execute a ROP chain for typical privilege escalation; however, when the exploit also needs to escape a container or namespace sandbox, a longer ROP chain that does not fit in spilled stack registers is required. In this case, a `popf; ret` gadget can be used to extend the ROP chain by temporarily disabling SMAP checking, and a subsequent gadget can pivot the stack to a much longer chain in user memory, requiring only 0x18 bytes of controlled data to extend the ROP chain as needed.

6.5 x86-64: IDT Hijacking

Past exploit techniques [46] were able to overwrite Interrupt Descriptor Table (IDT) entries to gain shellcode execution, however, this approach no longer functions in modern kernels because the IDT is read-only memory. There has been some exploration of relocating the IDT via `lidt` in the context of a Xen hypervisor vulnerability [14], however, this technique was previously unexplored in the context of forward control flow hijacking. No past work has discussed hijacking the IDT with both SMEP and SMAP enabled.

We found that several gadgets, including `native_load_idt`, can be used to relocate the address of the IDT. Once the IDT has been hijacked to point to a user-controlled page, the kernel enters a weird mode where the IDT is user-controlled while control flow continues in the kernel. Writing a ROP gadget to the IDT and triggering the respective interrupt allows the attacker to execute arbitrary code with almost all registers controlled by the attacker. In this mode, exceptions or faults that might normally kill the exploit or panic the kernel can be redirected to attacker-chosen addresses, potentially nullifying their effects. For example, the IDT entry for Page Fault exceptions could be redirected to an attacker-chosen gadget that returns from the exception immediately, ignoring null-pointer dereferences and other memory access violations rather than allowing the kernel to handle them.

6.6 aarch64: PAN Hijacking

We found that some aarch64 kernels contain gadgets like `msr pan, #0`, which can be used to disable the Privileged Access Never (PAN) mitigation. This is similar to techniques involving `popf` or `mov cr4` to disable SMAP on x86-64. The technique can be used to perform a pivot2usr [20] attack by first disabling PAN and then pivoting the stack to userspace memory. The case we evaluated on an Ubuntu kernel required chaining with another gadget to set the Link Register before it could be used.

Architecture	Technique	Success Rate
x86-64	swapgs Stack Pivoting	82%
x86-64	CR0 Hijacking	100%
x86-64	CR4 Hijacking	100%
x86-64	popf Extension + RetSpill	100%
x86-64	IDT Hijacking	100%
aarch64	PAN Hijacking	98%
aarch64	SPSR_EL1 Hijacking	100%

Table 2: Success rate of the Proof of Concept exploits for each technique across 50 runs.

6.7 aarch64: SPSR_EL1 Hijacking

This technique uses a specific gadget that sets `SPSR_EL1` along with all of the general-purpose registers using data on the stack and ends with an `eret` instruction. The gadget is located in the `ret_to_kernel` function shown in Figure 4. This gadget can be used in a Sigreturn Oriented Programming (SROP) [13] style attack, setting all general-purpose registers to control the arguments to a desired method. The `SPSR_EL1` register is used when the `eret` instruction executes to restore the Processor State (PSTATE).

The PSTATE includes the PAN MSR, allowing this gadget to control not only the general-purpose registers but also the status of the PAN mitigation. With `SPSR_EL1` set to disable PAN on `eret`, the `elr_el1`, which controls what the instruction pointer is set to on `eret`, can be set to a gadget that pivots the stack to userspace. This allows the attacker to continue ROPing or chaining additional `ret_to_kernel` executions.

7 Technique Validation

Now that we have discovered several new exploitation techniques using System Register Hijacking, we attempt to validate that each technique can be used by an attacker.

7.1 Proof of Concepts

We evaluated the potential of each SRH technique by developing a Proof of Concept (PoC) exploit. This involved creating an intentionally vulnerable kernel module and exploiting the kernel using the vulnerable module while leveraging the technique. Some of the techniques require Stack Control as a precondition, most commonly obtained via ROP, as a starting point. For example, `popf` requires stack control because it sets EFLAGS based on the value at the top of the stack. The vulnerabilities inserted included primitives to provide ROP, control flow hijacking, and kernel address disclosures. An

```
static ssize_t proc_ioctl(
  struct file* filep, u
  nsigned int cmd,
  unsigned long arg
)
{
...
  size_t entries = 0;
  struct ropchain_req *req = arg;
  get_user(entries, &req->entries);
  uint64_t *ropchain = memdup_user(
    &req->chain,
    entries * sizeof(req->chain[0])
  );
  asm volatile (
      ".intel_syntax noprefix;"
      "mov rsp, %0;"
      "ret;"
      ".att_syntax prefix;"
      : : "r" (ropchain) :
  );
...
}
```

Figure 5: A snippet of an inserted vulnerability which provides ROP to the attacker.

example of one such inserted vulnerability, intended to provide ROP, can be seen in Figure 5. We successfully created a PoC for every technique presented in Section 6. The PoCs for each technique take advantage of one or more artificial vulnerabilities in the kernel module to escalate privileges. For PoCs related to techniques affecting Feature Registers, the PoC disables a mitigation and then demonstrates that it is disabled by performing an exploitation step that would otherwise be prevented by the feature (e.g., ret2usr [44] for CR4). For PoCs related to techniques affecting system data structures, the PoC hijacks the address of the structure to a controlled address and uses the forged structure to continue exploitation (e.g., hijacking the IDT and then causing a divide-by-zero exception to invoke the malicious Division Error descriptor). The stability of each of these Proofs of Concept can be found in Table 2.

7.2 Real-World Evaluation

Next, we analyzed all KernelCTF submissions with public exploits [24] that perform Local Privilege Escalation. We then manually identified the class of attack each exploit used to achieve privilege escalation. We also analyzed whether the exploits would still work if KERNEL_IBT/FineIBT were enabled for the kernel. We found that only five of the forty exploits we analyzed utilized a Data-Only Attack rather than Control Flow Hijacking. As such, most of the submitted exploits would no longer work with these mitigations enabled, thus forcing future submissions away from control flow hijacking, which currently appears to be the path of least resistance.

To demonstrate the applicability of the `swapgs` Stack Pivoting technique (which can bypass IBT/FineIBT), we modified the KernelCTF exploits for six existing vulnerabilities, listed in Table 3, to use `swapgs` Stack Pivoting in place of their original stack pivoting techniques. All exploits were successfully adapted to use the technique, provided that an attacker could fully control the value of `GSBase`.

CVE	Version	Original	Modified
2021-4154*	5.4.120	100%	100%
2023-4623	6.1.36	100%	100%
2023-6111	6.1.60	100%	100%
2023-6817	6.1.63	98%	98%
2024-1085	6.1.70	100%	90%
2024-26925	6.1.81	100%	88%

Table 3: Stability across 50 runs of the original exploits and modified exploits which use the `swapgs` Stack Pivoting technique for six real world vulnerabilities.
*2021-4154 targeted a version of the Linux kernel prior to version 5.9, which introduced support for the FSGSBASE extension, we patched out the check in the `arch_prctl` syscall that prevents setting `GsBase` to a value outside of the userspace address range to create the same capability as on modern kernels where FSGSBASE is enabled.

As part of this evaluation, we measured the stability of each exploit before and after modifying it to support our technique, finding that in most cases, it did not degrade the exploit's stability. In the worst case, stability dropped from 100% to 88% for CVE-2024-26925.

8 Case Studies

In this section, we discuss specific examples of porting existing exploits to use the `swapgs` Stack Pivoting technique and provide an example of using the technique to bypass FineIBT.

8.1 CVE-2024-26925

We ported the existing exploit against the kCTF environment for CVE-2024-26925 to use the `swapgs` Stack Pivoting technique. The changes involved inserting 39 lines, deleting 25 lines, and including a 195-line header file. The header file contains boilerplate code specific to the technique, including functions for spraying Transparent Huge Pages, spawning threads to hijack a return address in the forged stack that is pivoted to by the technique, and setting the malicious `GSBase` value. The header file includes five constants to be updated according to the target kernel: three values corresponding to offsets of per-cpu variables, one value representing the address of a `pop rsp; ret;` stack pivot gadget, and one value indicating the offset to the target return address to overwrite with the stack pivot gadget. The stack pivot gadget is used to pivot the stack to a prewritten ROP chain. This approach

```
pipe_read:
<+407>: mov    r11,QWORD PTR [rcx+0x8]
<+411>: mov    r13,QWORD PTR [rbp-0x80]
<+415>: mov    rdi,r13
<+418>: mov    rsi,r14
<+421>: mov    r10d,0x839cb82f
<+427>: sub    r11,0x10
<+431>: nop    DWORD PTR [rax+0x0]
<+435>: call   r11

__cfi_anon_pipe_buf_release:
<+0>:   endbr64
<+4>:   sub    r10d,0x839cb82f
        je     anon_pipe_buf_release
anon_pipe_buf_release:
<+0>:   nop    WORD PTR [rax]
<+4>:   nop    DWORD PTR [rax+rax*1+0x0]
<+9>:   push   rbp
<+10>:  mov    rbp,rsp
```

Figure 6: An example of an indirect call and call target when FineIBT is enabled.

ensures that the threads used to overwrite the return address only need to write 0x10 bytes for the pivot gadget and the address of the ROP chain rather than the entire ROP chain, which is likely much longer than 0x10 bytes.

The changes to the exploit file itself primarily involved adding approximately eight lines of code invoking the boilerplate header file functions. The remaining modifications consisted of converting the ROP chain to an array format expected by the boilerplate code and removing the existing ROP chain code originally used in the exploit.

8.2 CVE-2024-1085

In porting this exploit to use the `swapgs` Stack Pivoting technique, we inserted 48 lines and deleted 27 lines in the main exploit file. We included the same header file described in Section 8.1 and modified the necessary values. In this case, neither the per-cpu offsets nor the stack offset differed from those in the CVE-2024-26925 exploit, so the only required modification in the header file was updating the address of the `pop rsp; ret;` gadget.

Again, most changes involved converting the ROP chain originally used by the exploit into a format compatible with the functions in our header file.

8.3 Bypassing [Fine]IBT

In this case study, we discuss the `swapgs` Stack Pivoting technique's ability to bypass KERNEL_IBT and FineIBT mitigations. Our study found that this technique, beyond requiring no general-purpose register control, provides a unique capability: the ability to fully bypass the existing IBT-based kernel mitigations.

Figure 6 presents an example of the assembly of an indirect control flow target when FineIBT is enabled. Each control flow target has a stub function, beginning with `__cfi_`, which starts with the `endbr64` instruction, making it a valid target when IBT is enabled. This function also includes a software check against a hash to ensure that control flow originates from a valid caller. The stub functions containing `endbr64` exist only for functions that are potential indirect call sites, severely limiting possible control flow targets, with the subsequent software check restricting them even further.

We found that when the kernel is compiled with the FineIBT mitigation enabled, the entry points begin with `endbr64` instructions but lack the software checks to verify the source of the control flow, as seen in Figure 1. This omission makes them valid targets under FineIBT.

As of this writing, all Linux kernel versions since the introduction of FineIBT support in version 6.2 are affected by this bypass. This specific bypass can be addressed in software by preventing users from storing kernel addresses in the `GSBase` register when calling into kernel code, as discussed in Section 9. However, the architectural reason this bypass works - the requirement that kernel entry points begin with `endbr64` when IBT is enabled - is not addressable in software. In our review, we did not identify any other methods of using entry points to bypass FineIBT aside from the `swapgs` Stack Pivoting technique described.

We demonstrate the ability to bypass FineIBT (and KERNEL_IBT by extension) by modifying an existing privilege escalation exploit [12] for the CVE-2024-41009 [4] vulnerability in the Linux kernel.

Since virtualization support for CET is not yet merged into KVM, nor is there emulation support for CET in QEMU, we conducted our experiment on a bare-metal system where the host kernel had FineIBT enabled. We performed the experiment on a system with an Intel i7-1185G7 processor and a self-compiled 6.6.32 kernel with FineIBT enabled. We found that our modified exploit, which targets the `entry_SYSCALL_compat` entry point to perform a `swapgs` Stack Pivot, successfully escalated privileges to root against the FineIBT-hardened kernel. The exploit succeeded in 100% of executions across 50 runs with and without FineIBT enabled. For comparison, the original exploit also succeeded 100% of the time across 50 runs in the KernelCTF environment.

9 Mitigations

In this section, we explore and propose mitigations for the discovered techniques. For systems where IBT is available (e.g., any recent Intel processor) and FineIBT is enabled (the new default CFI for mainline x86-64 kernel builds), or kCFI is enabled, these techniques are mostly mitigated by preventing attackers from hijacking control through current approaches. For systems where IBT is not available, such as on ARM

systems or those with older Intel and current AMD processors, and for kernel distributions that do not want to enable kCFI, we suggest our own mitigations.

9.1 Mitigating swapgs

To mitigate `swapgs` Stack Pivoting, we propose implementing checks on the `GSBase` value coming from userspace. This technique can be fully mitigated by preventing system calls from being executed when the `GSBase` value is outside the userspace address range. We add a check that performs an `rdmsr` instruction to read the `KERNEL_GSBASE_MSR` and ensure that it is in the user address range, returning a new error `EWRGSBASE` to userspace otherwise.

We implement our mitigation on Linux Kernel v6.11-rc7 and evaluated its performance by running the Phoronix Benchmarks [1]. The results can be found in Table 4. As shown, the mitigation introduces minimal overhead, only 0.91% on average. Alternative mitigations, such as saving, zeroing, and restoring the `KERNEL_GSBASE_MSR`, would keep syscall behavior unchanged (e.g., no additional error codes), but additional `rdmsr` and `wrmsr` calls would introduce higher overhead.

9.2 Mitigating cr0 and cr4

For the unmitigated `cr0` and `cr4` gadgets, we recommend extending CR-Pinning to include these cases.

9.3 Mitigating lidt

The `lidt` gadgets can be mitigated by enforcing that their value always be set to the constant virtual address where they are located. On modern PML4 Linux, this is always at `0xfffffe0000000000`. Performing a post-check on the value after any `lidt` instruction and resetting its value should be sufficient to prevent its misuse.

9.4 Mitigating MSR pan and MSR spsr_el1

There were very few valid `msr pan` gadgets in the kernel builds we analyzed, with several kernel builds containing zero occurrences of a PAN disable gadget. The `uaccess_enable_privileged` function was a common source of valid PAN hijacking gadgets. Every `uaccess_enable_privileged` call should be followed by a corresponding call to `uaccess_disable_privileged`. However, since these functions are not being inlined, it is possible to misuse the `uaccess_enable_privileged` function. A simple mitigation for this gadget would be to always inline these functions, ensuring that every `uaccess_enable_privileged` call is always followed by a `uaccess_disable_privileged` call.

The `spsr_el1` technique we described in Section 6.7 also controls the PAN feature's status. Depending on how the `ret_to_kernel` function is used normally, it might be possible to protect its return target with ARM Pointer Authentication. This way, an attacker would need to be able to sign a pointer in order to use the gadget.

9.5 Mitigating popf

Mitigations for the `popf` based technique are less clear. The `popf` technique for temporarily bypassing SMAP in ROP chains is effectively non-mitigatable, as there are too many occurrences of the instruction to possibly mitigate them all. Even in intended instruction locations, it is likely unknowable whether the AC bit should be set by that specific instance. It seems that the decision to make the AC bit in `EFLAGS` control SMAP's enforcement was made for ease of implementation and adoption. However, as an unintended side effect, it has weakened the feature in contexts where an attacker can achieve forward control flow hijacking and ROP.

10 Discussion

Gadget Discovery. We do not attempt to design an advanced gadget scanner for discovering SRH gadgets. Most ROP/JOP gadget scanners are effective at finding common gadgets but fail to capture complex control flows. Gadget scanners based on symbolic execution, such as `angrop` [7], can better capture control flows but lack support for handling the semantics of system instructions. We found that `angrop` was effective at measuring the existence of a significant number of valid SRH gadgets but lacked the ability to capture all the classes of gadgets we discussed on its own. Hence, we implemented our own verification passes on top of the results of `angrop`, as discussed in Section 5.3.

Unusable or Impractical Gadgets. When evaluating the exploitability of system registers, we found that some registers, which are theoretically security-sensitive, are not in practice. The `fsbase` register is unused by the kernel, so it is, of course, unusable, but other instructions such as `lldt` and `ltr` were unusable for less clear reasons. Both `lldt` and `ltr` modify segment selectors, which contain a privilege level in bits 1 and 0. Bit 2 indicates which table (LDT or GDT) the selector references, and bits 3 through 15 specify an index into the relevant descriptor table. Linux allows adding custom descriptors to the LDT and GDT via the `modify_ldt` and `set_thread_area` system calls. The offset of the selected descriptor in the specified table is calculated by taking the index times eight, which architecturally prevents any misuse by crafting unaligned descriptors through the aforementioned system calls. Additionally, the segment limits on the LDT and GDT prevent selectors from being set to values that are out of bounds of the relevant tables. Ultimately, these properties

Benchmark	Unmitigated	Mitigated	Overhead
PHP(Score)	583037	581918	0.19%
Apache(Reqs/s)	36789.52	36520.26	0.74%
OpenSSL-SHA256(byte/s)	1783513310	1789828323	0.35%
OpenSSL-SHA512(byte/s)	2019532493	2034334743	0.72%
OpenSSL-RSA4096(sign/s)	1576.4	1588.8	0.78%
OpenSSL-RSA4096(verify/s)	103155.3	102974.6	0.18%
Sysbench-RAM(MiB/sec)	4818.16	4760.00	1.22%
Sysbench-CPU(events/sec)	7073.28	7028.56	0.64%
Memcached(ops/s)	876405.11	891076.33	1.65%
PyBench(ms)	1245	1246	0.08%
Nginx(Reqs/s)	24801.42	23980.04	3.42%

Table 4: Performance overheads from proposed mitigations on Phoronix Benchmarks

result in the gadgets that modify the `ldt` and `tr` segment selectors being unexploitable.

A register we deemed impractical to hijack was the `cr3` register. Page table entries are becoming more popular as targets for kernel memory corruption [47,56]. However, hijacking the page table base register, `cr3`, has not seen the same attention. We explored the possibility of hijacking `cr3` to a forged page table, but we deemed this technique impractical due to the high amount of knowledge about the kernel address space necessary to successfully achieve it.

Limitations of Proposed Techniques. Excluding `swapgs` Stack Pivoting, all of the techniques we described rely on register control or stack control as a precondition. In the case of register control, some gadgets rely on controlling a register that is never or rarely controlled at sites of Control Flow Hijacking. This would necessitate either using additional gadgets to set the register or filtering which target heap objects are corrupted in the process of turning memory corruption into control flow hijacking to those that allow the necessary register to be controlled. In the case of stack control, the attacker must have either already achieved ROP via pivoting the stack to user-controlled memory or have user-controlled data stored on the stack by kernel code executed before the hijacking occurs.

The `swapgs` Stack Pivoting technique, which does not have any preconditions, comes with a reduction in exploit stability. However, as discussed in Section 7.2, in most cases, it remains quite stable.

Applicability of Proposed Techniques. Under FineIBT, the only applicable technique for generic forward-edge control flow hijacking is `swapgs` Stack Pivoting, but the other techniques may be combined with this technique to bypass security features or hijack other system structures.

In cases where kCFI is enabled, the `swapgs` Stack Pivoting technique is less applicable. It may still be applicable in forward-edge Control Flow Hijacking if an unprotected indirect call is found. For example, calls into EFI runtime services are intentionally unprotected by kCFI, either explicitly using a compiler annotation or by 'hiding' the indirect control flow from the compiler using inline assembly [5]. Since kCFI does not protect against backward-edge Control Flow Hijacking, the technique could still be applied if an attacker can hijack a return address. In these cases, though, it would only serve as a generic stack pivoting technique rather than a bypass to kCFI.

Improvements to Proposed Techniques. The `swapgs` Stack Pivoting technique has proved so generic that it could likely be applied in an automatic exploit generation scenario. There is not a significant amount of manual effort required to port an exploit to use this technique, and it may be possible to automate that effort. The work that is currently manual includes identifying a few per-cpu variable offsets, the offset of the forged stack to overwrite, constructing a ROP chain for privilege escalation, and swapping out the control flow target for one of the syscall entry points.

11 Related Work

Bypassing Kernel Security Features. In the past decade, security issues in kernels have been extensively studied. One of the main research directions is bypassing kernel security features. For instance, Chen et al. [15] studied elastic objects in the kernel to bypass protections such as KASLR, and Liu et al. [41] managed to bypass KASLR by exploiting design flaws in Kernel Page Table Isolation (KPTI). Hund et al. [29] implemented an automated system to bypass kernel code integrity protection mechanisms. Kemerlis et al. [35] bypassed kernel isolation protections by leveraging implicit page frame sharing. Additionally, side-channel attacks have been proposed to bypass kernel security features, such as SMAP and Kernel ASLR [19, 27].

Escalating Privileges in the Kernel. A number of techniques have been proposed to escalate privileges in the kernel. For example, DirtyCred [38] was proposed to swap unprivileged and privileged kernel credentials, achieving kernel privilege escalation. SCAVY [9] automatically discovers memory

corruption targets in the Linux kernel for privilege escalation. To counter these threats, researchers have proposed various approaches to mitigate kernel privilege escalation. For instance, PrivGuard [49] protects sensitive kernel data from privilege escalation attacks. Other works [52, 59] mitigate kernel privilege escalation by observing system call privilege changes and randomizing security identifiers.

Kernel Memory Corruption. Significant efforts have been made to identify kernel memory corruption vulnerabilities. For instance, K-Miner [22] is a framework that can systematically uncover memory corruption vulnerabilities. SCAVY [10] automatically discovers memory corruption targets for privilege escalation in the Linux kernel. Correspondingly, approaches have been proposed to mitigate kernel memory corruption. Multiple Kernel Memory (MKM) [37] protects kernel code and kernel data from kernel memory corruption. Kuzuno et al. [50] survey and classify protection and mitigation technologies against memory corruption attacks.

Upgrading Kernel Exploitation Capabilities. Several works have aimed to enhance the capabilities of kernel exploitation. KEPLER [58] facilitates exploit generation by automatically generating a single-shot exploitation chain. Zeng et al. [61] improve exploitation reliability by systematically studying kernel heap exploit reliability problems. SLAKE [16] escalates the exploitability of kernel vulnerabilities by facilitating slab manipulation. Despite these advancements, none of these works focus on techniques that compromise system registers, leaving a gap in exploitation methods that could further elevate the impact and severity of kernel vulnerabilities.

12 Conclusion

Modern Linux kernel exploitation relies heavily on forward control flow hijacking. In this research, we find that the introduction of CFI mechanisms endangers forward control flow hijacking and, therefore, renders most exploitation efforts that involve it obsolete. After thoroughly exploring system registers across multiple architectures and their associated instructions, we identify a technique that circumvents FineIBT, the default CFI on x86-64 kernel builds. We also introduce other techniques that hijack system registers, which, when combined with ROP, can ease exploitation. Finally, we suggest mitigations that could protect against these techniques. We conducted a study on the capabilities and stability of the `swapgs` Stack Pivoting technique across six real-world vulnerabilities, demonstrated its ability to bypass FineIBT on a bare-metal setup, and evaluated a potential approach to mitigating the technique.

13 Acknowledgements

We would like to thank our anonymous reviewers and shepherd for their feedback. This material is based upon work supported by the Google PhD Fellowship, Defense Advanced Research Projects Agency (DARPA), Naval Information Warfare Center Pacific (NIWC Pacific), and Advanced Research Projects Agency for Health (ARPA-H) under contracts N66001-22-C-4026, HR001124C0362, and SP4701-23-C-0074. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Google, DARPA, NIWC Pacific, or ARPA-H.

14 Ethics Considerations

We informed the Linux kernel security team of our findings and disclosed all the potential security impact responsibly before submitting this work. The kernel security team evaluated our report and gave us permission to publish it, stating an explicit preference for discussing mitigations for the techniques on the public kernel-hardening mailing list. During our discussion with the kernel security team, Linus brainstormed other potential mitigations for our discovered technique. We are continuing to collaborate with the kernel-hardening team to mitigate the exploitation techniques we discovered. Our research did not involve any experiments that could harm individuals.

15 Open Science

Our artifacts have been made available at https://doi.org/10.5281/zenodo.14728440. The artifacts include our scripts for measuring the occurrences of system instructions that may serve as SRH gadgets, our kernel module-based PoCs used to demonstrate individual techniques, the modified exploits included in our evaluation, the setup for reproducing the case study on bypassing FineIBT, and our implementation of the AVX timing side channel [17] used for breaking KASLR.

References

[1] Phoronix test suite. https://www.phoronix-test-suite.com/.

[2] aarch64: Add compiler support for shadow call stack, 2022. https://gcc.gnu.org/git/?p=gcc.git;a=commit;h=ce09ab17ddd21f73ff2caf6eec3b0ee9b0e1a11e.

[3] [patch v4 00/45] x86: Kernel ibt. https://lore.kernel.org/all/20220308153011.021123062@infradead.org/, 2022.

[4] Cve 2024-41009, 2024. https://nvd.nist.gov/vuln/detail/cve-2024-41009.

[5] efi: Add missing __nocfi annotations to runtime wrappers, 2024. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=99280413a5b785f22d91e8a8a66dc38f4a214495.

[6] Shadowcallstack, 2025. https://clang.llvm.org/docs/ShadowCallStack.html.

[7] angr team. angrop. https://github.com/angr/angrop, 2024.

[8] ARM. Arm architecture reference manual supplement - armv8, for armv8-r aarch64 architecture profile. https://developer.arm.com/documentation/ddi0600/latest/.

[9] Erin Avllazagaj, Yonghwi Kwon, and Tudor Dumitras. SCAVY: Automated discovery of memory corruption targets in linux kernel for privilege escalation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7141–7158, Philadelphia, PA, August 2024. USENIX Association.

[10] Erin Avllazagaj, Yonghwi Kwon, and Tudor Dumitraș. {SCAVY}: Automated discovery of memory corruption targets in linux kernel for privilege escalation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7141–7158, 2024.

[11] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against {Cross-Privilege} spectre-v2 attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 971–988, 2022.

[12] Billy Jheng Bing-Jhong and Muhammad Alifa Ramdhan. Cve 2024-41009 exploit, 2024. https://github.com/google/security-research/tree/master/pocs/linux/kernelctf/CVE-2024-41009_lts_cos.

[13] Erik Bosman and Herbert Bos. Framing signals-a return to portable shellcode. In *2014 IEEE Symposium on Security and Privacy*, pages 243–258. IEEE, 2014.

[14] Jérémie Boutoille. Xen exploitation part 1: Xsa-105, from nobody to root. https://blog.quarkslab.com/xen-exploitation-part-1-xsa-105-from-nobody-to-root.html, 2016.

[15] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A systematic study of elastic objects in kernel exploitation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1165–1184, 2020.

[16] Yueqi Chen and Xinyu Xing. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1707–1722, 2019.

[17] Hyunwoo Choi, Suryeon Kim, and Seungwon Shin. Avx timing side-channel attacks against address space layout randomization. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023.

[18] CVE Details. Linux: Security vulnerabilities, CVEs published in 2024, 2024. https://www.cvedetails.com/vulnerability-list/vendor_id-33/Linux.html?page=1&year=2024&order=1.

[19] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[20] Nicolas FABRETTI. Cve-2017-11176: A step-by-step linux kernel exploitation (part 4/4), 2018. https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part4.html#stack-pivoting.

[21] Alexander J Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios 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*, pages 527–546, 2023.

[22] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-miner: Uncovering memory corruption in linux. In *NDSS*, 2018.

[23] Google. Project zero: Exploiting the linux kernel via packet sockets. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[24] Google. https://github.com/google/security-research/tree/master/pocs/linux/kernelctf, 2024.

[25] Google. kernelctf vrp. https://google.github.io/security-research/kernelctf/rules.html, 2024.

[26] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: long live kaslr. In *Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings 9*, pages 161–176. Springer, 2017.

[27] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 368–379, 2016.

[28] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 368–379, 2016.

[29] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, page 383–398, USA, 2009. USENIX Association.

[30] Intel. Intel® 64 and ia-32 architectures software developer manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[31] Intel. Guidance for enabling fsgsbase. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/guidance-enabling-fsgsbase.html, 2019.

[32] Intel. Speculative behavior of swapgs and segment registers / cve-2019-1125, 2019.

[33] Hayden James. 85 https://linuxblog.io/85-of-all-smartphones-are-powered-by-linux/.

[34] Hyerean Jang, Taehun Kim, and Youngjoo Shin. Sysbumps: Exploiting speculative execution in system calls for breaking kaslr in macos for apple silicon. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 64–78, 2024.

[35] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 957–972, 2014.

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

[37] Hiroki Kuzuno and Toshihiro Yamauchi. Mitigation of kernel memory corruption using multiple kernel memory mechanism. *IEEE Access*, 9:111651–111665, 2021.

[38] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1963–1976, 2022.

[39] Jannik Linder. Linux user statistics: Dominance in supercomputing, web servers, iot, 2024. https://wifitalents.com/statistic/linux-user/.

[40] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. Meltdown: Reading kernel memory from user space. *Communications of the ACM*, 63(6):46–56, 2020.

[41] William Liu, Joseph Ravichandran, and Mengjia 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*, pages 10–18, 2023.

[42] Alexander Lobakin. Function granular kernel address space layout randomization (fg-kaslr). https://lore.kernel.org/lkml/20220209185752.1226407-1-alexandr.lobakin@intel.com/, 2022.

[43] Lukas Maar, Pascal Nasahl, and Stefan Mangard. Beyond the edges of kernel control-flow hijacking protection with hek-cfi. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1214–1230, 2024.

[44] mahaloz. ret2usr, 2021. https://ctf-wiki.mahaloz.re/pwn/linux/kernel/ret2usr/.

[45] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. Drop the rop fine-grained control-flow integrity for the linux kernel. *Black Hat Asia*, 2017, 2017.

[46] Vitaly Nikolenko. Linux kernel 3.2.0-23/3.5.0-23 (ubuntu 12.04/12.04.1/12.04.2 x64) - 'perf_swevent_init' local privilege escalation (3). https://www.exploit-db.com/exploits/33589, 2014.

[47] notselwyn. Flipping pages: An analysis of a new linux vulnerability in nf_tables and hardened exploitation techniques. https://pwning.tech/nftables/, 2024.

[48] Pawel Wieczorkiewicz. Linux kernel alternatives, 2021. https://grsecurity.net/linux_kernel_alternatives.

[49] Weizhong Qiang, Jiawei Yang, Hai Jin, and Xuanhua Shi. Privguard: Protecting sensitive kernel data from privilege escalation attacks. *IEEE Access*, 6:46584–46594, 2018.

[50] Takamichi Saito, Ryohei Watanabe, Shuta Kondo, Shota Sugawara, and Masahiro Yokoyama. A survey of prevention/mitigation against memory corruption attacks. In *2016 19th International Conference on Network-Based*

Information Systems (NBiS), pages 500–505. IEEE, 2016.

[51] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*, pages 138–157. IEEE, 2016.

[52] Lifeng Wei, Yudan Zuo, Yan Ding, Pan Dong, Chenlin Huang, and Yuanming Gao. Security identifier randomization: a method to prevent kernel privilege-escalation attacks. In *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 838–842. IEEE, 2016.

[53] Johannes Wikner and Kaveh Razavi. {RETBLEED}: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3825–3842, 2022.

[54] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. Phantom: Exploiting decoder-detectable mispredictions. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 49–61, 2023.

[55] willsroot. Entrybleed: Breaking kaslr under kpti with prefetch (cve-2022-4543). https://www.willsroot.io/2022/12/entrybleed.html, 2022.

[56] Nicolas Wu. Dirty pagetable: A novel exploitation technique to rule linux kernel, 2023.

[57] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. {KEPLER}: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1187–1204, 2019.

[58] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. {KEPLER}: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1187–1204, 2019.

[59] Toshihiro Yamauchi, Yohei Akao, Ryota Yoshitani, Yuichi Nakamura, and Masaki Hashimoto. Additional kernel observer to prevent privilege escalation attacks by focusing on system call privilege changes. In *2018 IEEE Conference on Dependable and Secure Computing (DSC)*, pages 1–8. IEEE, 2018.

[60] Shimizu Yutaro. *ctf 2019 writeup (blind-pwn, heap master, hack_me). https://shift-crops.hatenablog.com/entry/2019/04/30/131154, 2019.

[61] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for {K (H) eaps}: Understanding and improving linux kernel exploit reliability. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 71–88, 2022.

[62] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany 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*, page 3093–3107, 2023.

[63] Peter Zijlstra. x86/ibt: Implement fineibt. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=931ab63664f0, 2022.

16 Appendix

16.1 Gadgets on x86-64

The results of our measurements of potential and validated gadgets on x86-64 across several kernel builds are located in Table 5 and Table 6.

16.2 Gadgets on aarch64

The results of our measurements of potential and validated gadgets aarch across several aarch64 kernel builds can be found in Table 7 and Table 8.

Kernel	mov cr0	mov cr3	mov cr4	wrmsr	lidt	lgdt	lldt	ltr	wrgsbase	wrfsbase	swapgs	popfq	stac	clac	iretq
5.15.0-1073-aws_5.15.0-1073.79	14	44	16	113	8	8	61	137	4	2	23	10629	57	143	204
5.19.0-1029-aws_5.19.0-1029.30	19	45	17	125	13	7	65	51	4	2	24	10008	62	140	223
6.2.0-1018-aws_6.2.0-1018.18	10	45	19	126	7	7	63	74	4	2	23	13503	89	166	168
6.5.0-1024-aws_6.5.0-1024.24	9	46	18	131	7	7	65	66	4	2	23	14132	109	186	231
6.8.0-1021-aws_6.8.0-1021.23	9	48	20	137	17	17	79	67	4	2	23	12842	115	191	228
5.19.0-46-generic_5.19.0-46.47	14	45	19	118	11	7	62	55	4	2	24	10379	62	141	202
6.2.0-39-generic_6.2.0-39.40	19	45	18	127	7	7	63	64	4	2	23	13435	89	166	196
6.5.0-45-generic_6.5.0-45.45	12	47	18	124	16	11	62	63	4	2	23	14119	108	187	204
6.8.0-50-generic_6.8.0-50.51	14	49	17	137	18	17	81	58	4	2	23	12669	113	191	234
6.11.0-12-generic_6.11.0-12.13	8	51	18	137	14	16	102	49	4	2	23	14915	116	199	297
6.8.0-63.eln136.1	11	44	19	526	7	11	24	42	4	2	23	7087	116	184	80
6.9.0-0.rc6.52.eln136	11	46	17	540	6	11	25	34	4	2	23	7281	118	186	105
6.10.0-0.rc7.58.eln141	10	47	17	542	6	12	35	39	4	2	23	9621	118	196	104
6.11.0-63.eln142	9	48	19	548	7	11	42	35	4	2	23	7462	118	196	126
6.12.0-65.eln143	20	48	20	553	6	12	40	38	4	2	23	7576	112	193	103
6.8.0-63.fc41	9	46	18	121	6	9	63	24	4	2	23	8742	117	194	107
6.9.0-64.fc41	12	49	16	123	4	16	74	29	4	2	23	12064	116	201	108
6.10.0-64.fc41	14	49	16	118	4	16	78	27	4	2	23	9635	116	202	129
6.11.0-63.fc42	13	50	19	120	6	14	81	32	4	2	23	10100	116	201	112
6.12.0-65.fc42	10	49	18	121	15	17	70	23	4	2	23	10983	111	198	124
Averages:	12	47	18	229	9	12	62	50	4	2	23	10859	104	183	164

Table 5: Counts of potential x86-64 gadgets for each instruction across five versions of Ubuntu AWS, Ubuntu Generic, Fedora Enterprise Linux Next, and Fedora Core.

Kernel	mov cr0	mov cr3	mov cr4	wrmsr	lidt	lgdt	lldt	ltr	wrgsbase	wrfsbase	swapgs	popfq	stac	clac	iretq
5.15.0-1073-aws_5.15.0-1073.79	3	10	3	22	4	2	4	79	2	1	9	385	0	0	204
5.19.0-1029-aws_5.19.0-1029.30	3	9	2	21	9	2	5	36	2	1	10	476	0	0	223
6.2.0-1018-aws_6.2.0-1018.18	3	9	2	23	4	2	3	45	2	1	10	493	0	0	168
6.5.0-1024-aws_6.5.0-1024.24	5	8	2	28	4	2	4	37	2	1	9	520	0	0	231
6.8.0-1021-aws_6.8.0-1021.23	4	9	2	23	14	3	1	31	2	1	9	570	0	0	228
5.19.0-46-generic_5.19.0-46.47	3	9	2	22	7	2	3	42	2	0	10	462	0	0	202
6.2.0-39-generic_6.2.0-39.40	3	9	2	24	4	2	1	42	2	0	10	479	0	0	196
6.5.0-45-generic_6.5.0-45.45	5	9	2	23	11	2	0	30	2	0	9	467	0	0	204
6.8.0-50-generic_6.8.0-50.51	5	9	2	23	15	3	1	28	2	0	9	503	0	0	234
6.11.0-12-generic_6.11.0-12.13	3	10	3	22	8	3	3	20	2	0	9	547	0	0	297
6.8.0-63.eln136.1	5	6	3	184	4	1	4	12	2	0	9	413	0	0	80
6.9.0-0.rc6.52.eln136	5	7	2	199	2	1	1	5	2	0	9	411	0	0	105
6.10.0-0.rc7.58.eln141	4	7	2	201	2	1	1	6	2	0	9	437	0	0	104
6.11.0-63.eln142	3	8	3	206	2	1	1	5	2	0	9	415	0	0	126
6.12.0-65.eln143	3	8	3	207	2	1	1	8	2	0	9	468	0	0	103
6.8.0-63.fc41	4	8	2	31	3	2	0	9	2	0	9	428	1	0	107
6.9.0-64.fc41	3	9	2	30	1	2	2	6	2	0	9	2484	1	0	108
6.10.0-64.fc41	3	9	2	30	1	4	1	4	2	0	9	517	1	0	129
6.11.0-63.fc42	4	10	3	30	1	3	1	7	2	0	9	555	1	0	112
6.12.0-65.fc42	5	10	3	30	6	2	0	3	2	0	9	612	1	0	124
Averages:	4	9	2	69	5	2	2	23	2	0	9	582	0	0	164

Table 6: Counts of validated x86-64 System Register Hijacking gadgets, across five versions of Ubuntu AWS, Ubuntu Generic, Fedora ELN, and Fedora FC.

Kernel	elr_el1	pan	sctlr_el1	spsr_el1	tcr_el1	ttbr0_el1	ttbr1_el1	vbar_el1
5.15.0-1073-aws_5.15.0-1073.79	20	31	52	18	27	2570	2586	44
5.19.0-1029-aws_5.19.0-1029.30	21	23	53	20	28	1475	1450	9
6.2.0-1018-aws_6.2.0-1018.18	20	23	50	19	28	1487	1467	10
6.5.0-1024-aws_6.5.0-1024.24	21	23	42	19	25	1673	1651	9
6.8.0-1021-aws_6.8.0-1021.23	23	23	43	21	24	1698	1677	8
5.19.0-46-generic_5.19.0-46.47	21	23	53	20	28	1408	1383	9
6.2.0-39-generic_6.2.0-39.40	20	23	50	19	28	1437	1417	10
6.5.0-45-generic_6.5.0-45.45	21	23	43	19	24	1484	1464	9
6.8.0-50-generic_6.8.0-50.51	23	23	43	21	23	1481	1462	8
6.11.0-12-generic_6.11.0-12.13	26	23	46	24	20	1414	1398	8
6.8.0-63.eln136.1	23	33	39	21	24	25	4	8
6.9.0-0.rc6.52.eln136	23	33	40	21	20	21	4	8
6.10.0-0.rc7.58.eln141	24	33	42	22	21	22	4	8
6.11.0-63.eln142	26	33	42	24	21	22	4	8
6.12.0-65.eln143	26	35	43	24	23	24	6	8
6.8.0-63.fc41	23	23	43	21	24	2801	2780	8
6.9.0-64.fc41	23	23	44	21	21	2819	2801	8
6.10.0-64.fc41	24	23	46	22	20	2807	2790	8
6.11.0-63.fc42	26	23	46	24	20	3146	3129	8
6.12.0-65.fc42	26	25	47	24	22	3147	3130	8
Averages:	23	26	45	21	24	1548	1530	10

Table 7: Count of potential aarch64 gadgets for each system register studied, across five versions of Ubuntu AWS, Ubuntu Generic, Fedora ELN, and Fedora FC.

Kernel	elr_el1	pan	sctlr_el1	spsr_el1	tcr_el1	ttbr0_el1	ttbr1_el1	vbar_el1
5.15.0-1073-aws_5.15.0-1073.79	2	0	18	1	7	133	136	38
5.19.0-1029-aws_5.19.0-1029.30	0	1	4	0	6	5	7	2
6.2.0-1018-aws_6.2.0-1018.18	1	1	3	1	6	8	7	2
6.5.0-1024-aws_6.5.0-1024.24	4	1	14	4	6	190	185	5
6.8.0-1021-aws_6.8.0-1021.23	4	1	16	6	8	206	201	3
5.19.0-46-generic_5.19.0-46.47	0	1	4	0	6	4	6	2
6.2.0-39-generic_6.2.0-39.40	1	1	3	1	6	8	7	2
6.5.0-45-generic_6.5.0-45.45	4	1	15	4	6	197	192	5
6.8.0-50-generic_6.8.0-50.51	5	1	16	6	8	222	217	3
6.11.0-12-generic_6.11.0-12.13	3	1	18	4	5	254	250	3
6.8.0-63.eln136.1	6	0	11	7	4	1	0	3
6.9.0-0.rc6.52.eln136	4	0	11	5	3	1	0	3
6.10.0-0.rc7.58.eln141	4	0	13	5	5	4	0	3
6.11.0-63.eln142	4	0	13	5	5	4	0	3
6.12.0-65.eln143	4	1	13	5	6	4	0	3
6.8.0-63.fc41	6	1	15	7	7	113	109	3
6.9.0-64.fc41	3	1	15	4	6	111	107	3
6.10.0-64.fc41	3	1	17	4	4	113	110	3
6.11.0-63.fc42	3	1	17	4	4	137	134	3
6.12.0-65.fc42	3	2	17	4	5	139	136	3
Averages:	3	1	13	4	6	93	90	5

Table 8: Counts of validated aarch64 System Register Hijacking gadgets, across five versions of Ubuntu AWS, Ubuntu Generic, Fedora Enterprise Linux Next, and Fedora Core.