Crashing Through Defenses: Exploiting Segfaults and Chaining around Intel CET

Marcos Bajo[1], Ritvik Goyal[2], Apostolos Chatzianagnostou[1], and Christian Rossow[1]

[1]CISPA Helmholtz Center for Information Security
[2]Indian Institute of Technology Kanpur

***Abstract*—Code reuse is the predominant attack strategy for exploiting memory corruption vulnerabilities in modern software. In response, Control Flow Integrity (CFI) has been adopted to restrict unintended control-flow transfers and mitigate these attacks. Intel Control-Flow Enforcement Technology (CET) is the most popular CFI implementation in modern x86_64 systems, providing hardware-based protection against conventional code reuse attacks such as ROP and SROP. Although advanced techniques have been proposed to bypass Intel CET, they typically require application-specific features or uncommon programming constructs, limiting their practical applicability.**

This paper introduces Segmentation Fault Oriented Programming (SFOP), a novel code reuse attack that exploits previously unidentified weaknesses in the interaction between Intel CET and the Linux signal handling subsystem. Unlike other code reuse techniques, SFOP does not require program-specific features, and can reliably exploit any vulnerable application on modern x86_64 Linux with Intel CET enabled. SFOP enables an attacker to execute arbitrarily many function calls with fully controlled arguments, turning a single memory corruption vulnerability into arbitrary code execution. We demonstrate the practical impact of SFOP through real-world exploits, and discuss mitigation strategies to prevent SFOP attacks.

1. Introduction

Code reuse is the predominant attack strategy for executing arbitrary code after exploiting memory-related vulnerabilities. Code-reuse attacks leverage the existing instructions within a program and its linked libraries to execute malicious operations without introducing new code. After initially compromising the program's control flow, the attacker chains together short instruction sequences (*gadgets*), which collectively mimic the intended malicious behavior.

While code reuse attacks execute valid code, the control flow during the exploitation deviates from the program's intended execution paths, creating a *weird machine*. Control Flow Integrity (CFI) schemes [2] aim to prevent this behavior by ensuring that the program's execution flow adheres to the intended valid code paths of the program, described as a Control-Flow Graph (CFG), at runtime. They instrument the program to check *backward-edge* control transfers (i.e., function returns) and *forward-edge* control transfers (i.e., indirect calls and jumps) [29] [17] [4].

Today, the most widespread CFI scheme for x86_64 systems is Intel Control-flow Enforcement Technology (CET), a hardware-assisted CFI mechanism supported in both Windows and Linux [24]. Available in Intel processors since the 11th generation (Tiger Lake), CET combines a backward-edge CFI defense known as *Shadow Stack*, with the forward-edge CFI defense known as *Indirect Branch Tracking* (IBT). CET offloads CFI checks to the hardware, resulting in negligible performance overheads. Due to this, all major compilers, including GCC, LLVM and MSVC, support CET, making it the *de facto* CFI scheme in modern systems [5]. Nowadays, CET is found active by default in critical applications on Windows, while most Linux distributions ship their libraries and applications (e.g., those downloaded from package managers like *apt*) with CET support [49].

The presence of CET in modern systems is a significant level-up from the previous status quo, effectively preventing all known code reuse attacks prior to the appearance of CFI, such as *Return-Oriented Programming* (ROP) [47] or *Sigreturn-Oriented Programming* (SROP) [9]. However, some works have introduced advanced code-reuse techniques that bypass CFI schemes including CET. This includes *Counterfeit Object-Oriented Programming* (COOP) [46], which abuses C++ virtual tables, *Functional Oriented Programming* (FOP) [20], which abuses loops using function pointers, and *Coroutine Frame-Oriented Programming* (CFOP) [3], which abuses C++ coroutines. However, as already indicated, these CET-compliant code-reuse attacks require specific primitives in the vulnerable program. This has significantly hindered their widespread adoption; they are far less generic than classic ROP attacks. Unless a vulnerable program features the required primitives, the widespread adoption of CET hinders code-reuse attacks as attackers find no viable options to exploit vulnerabilities.

In this paper, we present *Segmentation Fault-Oriented Programming* (SFOP), a novel code-reuse attack that significantly lowers the bar for exploiting CET-protected programs. Unlike previous attacks, SFOP is reproducible practically everywhere, as it does not require the presence of virtual pointers, vulnerable loops, the use of C++ or any other specific feature. Instead, SFOP can be used by default in any Linux program, allowing attackers to execute infinitely many functions with arbitrary arguments, essentially executing arbitrary code, despite the presence of CET.

SFOP leverages novel weaknesses we identify in the

implementation of Intel CET combined with the signal handling mechanism in x86_64 Linux systems. While Linux signals have been known to be exploitable, we show in § 3.3.2 that the well-known SROP attack no longer works with CET enabled, and how subsequent attempts [51] to leverage signals to bypass CFI require unreasonable assumptions, like GOT overwrites or manually sending signals to the exploited process. Instead, for building SFOP, our paper presents to the best of our knowledge the first in-depth analysis of how signals interact with Intel CET, uncovering twelve novel weaknesses that allow an attacker to bypass CET in a practical manner despite state-of-the-art defenses.

SFOP ranks among the lowest hanging fruit available for an attacker facing a program protected by CET and any standard binary hardening defenses (DEP, RELRO) [25]. It requires only a memory corruption vulnerability that allows the attacker to overwrite a single forward-edge pointer in memory, along with some data to be used during the code-reuse phase, a common primitive in real-world vulnerabilities. Programs exploited using SFOP do not require a prior signal registered or interaction with any other process. SFOP requires some syscalls during exploitation (e.g., `sigaction`), but we find them allowed by default in common defenses such as the *seccomp* filters of popular applications (see § 6).

To show the practicality of SFOP, we implement a series of Proof of Concept (PoC) exploits that showcase both attacks against sample and real-world programs that present memory corruption vulnerabilities. Specifically, we prepare PoCs that showcase SFOP against the Nginx web server and the Ladybird web browser, both running with Intel CET enabled. We use SFOP to execute arbitrary code in CET-protected but vulnerable versions of Nginx (CVE-2013-2028) and Ladybird (CVE-2021-4327).

To summarize, our contributions are:

- We present *Segmentation Fault-Oriented Programming* (SFOP), a novel code-reuse attack that allows an attacker to execute infinitely many arbitrary calls with arbitrary arguments, under a CET-enabled scenario and standard defenses.
- We study the interaction of Intel CET with Linux signals, uncovering twelve novel weaknesses that allow an attacker to bypass CET in a practical manner.
- We showcase the practicality of SFOP with PoC exploits against real-world programs, Nginx and Ladybird, both compiled with Intel CET enabled.
- We propose a series of mitigations and defenses for the Linux kernel to protect against SFOP attacks.

2. Background

We aim to show SFOP is the lowest hanging fruit for an attack under a CET-enabled scenario compared to any priorly introduced code-reuse technique. For this purpose, this section presents an analysis of previous code-reuse techniques, highlighting the attack requirements and their limitations. We compare this against the threat model of our attack, showing how SFOP is more practical and generic.

2.1. Intel CET

Intel CET consists of two main components: the *shadow stack*, which provides backward-edge CFI; and *Indirect Branch Tracking* (IBT), which provides forward-edge CFI.

The shadow stack is a protected stack that stores return addresses of invoked functions. When a function is called, its return address is pushed onto both the regular stack and the shadow stack. The moment a function returns, the return address is popped from both stacks and compared by the hardware. If the addresses do not match, a control protection exception is raised, and the program is terminated by the kernel. This mechanism prevents attackers from tampering with return addresses, a necessary step in many code-reuse attacks like ROP. The loader allocates the shadow stack in the process' virtual memory prior to calling the *main* function of the program, but the page is marked by the kernel with the special `VM_SHADOW_STACK` bit [35] that prevents the process from tampering with it.

IBT protects indirect calls and jumps by ensuring that they only target valid function entry points (i.e., the first instruction of a function). For this, compilers mark valid entry points with an *endbr64* instruction. When an indirect branch is executed, the CPU checks that the target address contains an *endbr64* instruction. If not, a control protection exception is raised and the kernel terminates the program.

IBT aims to limit one of the key techniques used in code-reuse attacks: the ability to control argument-passing registers. These registers (`rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` in x86_64 Linux) contain the first six arguments to functions. Assuming attackers hijack a forward pointer, controlling these argument registers is mandatory for successful exploitation. Most code-reuse attacks, such as JOP, rely on getting control over these registers by redirecting the control flow to the middle of functions or function prologues, where the attacker can find instruction sequences that move attacker-controlled values into argument-passing registers.

2.2. Threat Model

SFOP considers the typical threat model of CFI schemes, assuming a malicious actor that can (1) read program memory to learn the memory layout and bypass ASLR [30] protections; and (2) leverage a memory corruption vulnerability (e.g., stack overflow) to write arbitrary data in memory. Specifically, we assume the attacker can leverage a memory corruption vulnerability at least once to overwrite at least one pointer in memory, and place a reduced amount of other data in memory to be used during the code-reuse phase.

We assume every standard security feature widely available in modern compilers, including DEP/NX—the vulnerable program has no pages marked as both writable and executable ($W \oplus X$), and Full RELRO—the Global Offset Table (GOT) is read-only, preventing GOT overwrite attacks. On top of this, we assume that the program has been compiled with Intel CET enabled, providing both forward-edge and backward-edge CFI protections.

We do not assume the presence of any programming construct in the vulnerable program (e.g., C++ virtual tables), or a specific program state (e.g., pre-registered signals). We do assume that the attacker can access the `sigaction` syscall; not only is this by-default behaviour, but also we show to be usually allowed by default in common defenses such as *seccomp* (see § 6).

2.3. A Framework for Code-Reuse

In order to discuss different code-reuse techniques systematically, we propose a simple framework that describes the necessary components of every code-reuse attack. Every code-reuse attack consists of three components: (1) a *dispatcher*, (2) a *dispatcher table*, and (3) *gadgets*.

The *dispatcher* is a piece of code—some loop—that transfers control to a series of gadgets. After the execution of a given gadget, the control flow falls back to the dispatcher, which executes the next gadget. This dispatcher can be *explicit*, meaning that the loop is part of the program code, or *implicit*, meaning that the loop belongs to some mechanism not belonging to the program code.

The *dispatcher table* is a data structure in memory that contains the addresses of the gadgets. The dispatcher reads this table to determine the next gadget to be executed.

The *gadgets* are instruction sequences that perform specific operations to prepare the exploit (e.g., setting up an argument register). Usually, gadgets are instruction sequences ending in a control transfer instruction (e.g., *ret*, *jmp*) that can be chained together with the goal of setting arbitrary values for the arguments of an attacker-controlled call.

2.4. Comparison of Code-Reuse Techniques

We summarize the main properties of code-reuse techniques proposed in the past in Table 1.

2.4.1. Classic Code-Reuse Techniques. Most of the code-reuse techniques introduced prior to the introduction of CFI no longer work under Intel CET.

Return-Oriented Programming (ROP) [47] relies on an implicit dispatcher, which is the architectural mechanism that pops stack addresses into the instruction pointer (`rip`) when executing the *ret* instruction. The attacker uses the stack as the dispatcher table, executing gadgets that end in a *ret* instruction. CET prevents ROP using its shadow stack.

Jump-Oriented Programming (JOP) [7] uses an explicit dispatcher, i.e., a loop in the program that reads addresses from a dispatcher table in memory and jumps to them. These addresses are gadgets that start anywhere and end in an indirect jump (e.g., *jmp rax*), whose target is controlled towards the dispatcher again. CET prevents JOP by enforcing that indirect jumps only target valid function entry points.

Call-Oriented Programming (COP) [45] is similar to JOP, but it uses indirect calls instead of indirect jumps. CET prevents COP by enforcing that indirect calls only target valid function entry points, preventing COP gadgets.

Table 1. PRIORLY PROPOSED CODE-REUSE TECHNIQUES.

Name	CET Bypass	Works in C code	Implicit dispatcher	Additional exploitation assumptions
ROP	✗	✓	✓	None
JOP	✗	✓	✗	None
COP	✗	✓	✗	None
SROP	✗	✓	✓	None
CHOP	✓	✗	✓	C++ Exceptions
FOP	✓	✓	✗	None
COOP	✓	✗	✗	Virtual tables
CFOP	✓	✗	✗	Coroutines
eSROP	✓	✓	✓	Signal sending, GOT hijack
SFOP	✓	✓	✓	None

Sigreturn Oriented Programming (SROP) [9] leverages the `sys_rt_sigreturn` syscall, which restores all the CPU registers from a sigreturn frame structure positioned in attacker-controlled memory. CET prevents SROP (§ 3.3.2).

2.4.2. Post-CFI Code-Reuse Techniques. We now discuss CET-compliant code-reuse attacks and their assumptions.

Catch Handler Oriented Programming (CHOP) [15] relies on C++ exceptions to create an implicit dispatcher. When an exception is thrown, the C++ runtime unwinds the stack, looking for matching *catch* blocks. During this process, CET's shadow stack is not checked, so it is possible to corrupt stack data to divert the execution flow. Compared to SFOP, CHOP requires (1) the target program to feature C++ exceptions; and (2) overwrite specific unwinding structures.

Functional Oriented Programming (FOP) [20] uses complete functions as gadgets. These gadgets are executed from the function beginning, so they are valid IBT targets. FOP requires an explicit dispatcher, i.e., a loop in the program that reads addresses from a dispatcher table (e.g., an array of function pointers) and calls them; gadgets then execute a return instruction to go back to the dispatcher. Compared to SFOP, FOP (1) requires an explicit dispatcher in the code that calls function pointers in a loop and does not modify the registers between iterations; (2) a vulnerability needs to overwrite the specific memory structure used as the dispatcher table; unlike techniques with implicit dispatchers, the exploit payload needs to be placed at an *absolute* location in memory, the attacker cannot just overwrite *any* memory to kick-start the exploitation; and (3) FOP requires chaining thousands of gadgets to launch basic attacks [1], resulting in considerably large payloads (for gadget addresses and runtime data during the code-reuse phase) that most memory corruption vulnerabilities hardly allow to inject.

Counterfeit Object-Oriented Programming (COOP) [46] uses C++ virtual tables and an explicit dispatcher that loops over a series of objects calling their virtual functions. Compared to SFOP, COOP requires (1) the target program to be written in C++ and feature an object using virtual functions; and (2) overwriting a virtual table and an explicit dispatcher in the code that calls virtual functions in a loop. In addition,

defenses have been proposed to protect virtual tables [50] [31] and are available in modern compilers [27] [18] [19].

Coroutine Frame-Oriented Programming (CFOP) [3] relies on C++ coroutines for hijacking the control flow. It uses an explicit dispatcher, but this code is always present in programs using coroutines. Compared to SFOP, CFOP requires (1) the target program to use C++ coroutines; and (2) the presence of some overwritable coroutine frame.

Enhanced Sigreturn Oriented Programming (eSROP) [51] is the only priorly proposed attack that leverages signals to bypass CFI. However, it is mostly theoretical due to its impractical threat model. Compared to SFOP, eSROP (1) requires overwriting GOT entries, which Full RELRO prevents; (2) requires sending signals to the target process from an additional controlled process; (3) introduces instability due to timing, as eSROP bypasses CFI checks by interrupting with an externally-sent signal software-based CFI checks, then running a data-based attack to modify the registers holding the CFI check result.

3. Security Analysis of Signals with Intel CET

While previous work [9] has documented how signals worked before the introduction of CET, we now present, to our knowledge, the first comprehensive security analysis of Intel CET and signals, highlighting novel weaknesses that we will leverage for building SFOP. We back up our analysis with references to the Linux kernel source code.

3.1. Linux Signals

Signals are a fundamental mechanism for asynchronous event notification and inter-process communication in every UNIX-like operating system, including Linux. In essence, signals are software interrupts that notify a process when a specific event occurs. These signals can be generated by the kernel, other processes, or by the same process itself.

When a process receives a signal, the kernel evaluates the signal's *disposition*, which determines how the process should respond to the signal. Usually, the disposition can be (1) ignore the signal, discarding it; or (2) perform the default action associated with the signal, which varies depending on the signal type (e.g., terminate the process).

The signal handling subsystem allows processes to change the default disposition of signals. Next, we describe every step involved during the registration, delivery and processing of a custom signal in a process.

3.1.1. Registering the Signal Handler. Processes change the signal disposition by registering a *signal handler* using the `signal` or `sigaction` libc functions. This handler is a custom function that the process executes when the signal is received. In this work, we focus on `sigaction`; both `signal` and `sigaction` rely on the `sys_rt_sigaction` syscall internally, but the wider range of flags in `sigaction` is more versatile for an attack.

`sigaction` receives three arguments, in order: (1) the signal number of the signal to register, (2) a pointer to a sigaction structure *act*, (3) a pointer to another sigaction structure *oldact*. Only *act* contains the information related to the new signal handler registration; *oldact* is a buffer where the kernel will store the information for the signal handler previously registered for that signal.

```
1  struct sigaction {
2          void *sa_handler;
3          sigset_t sa_mask;
4          unsigned long sa_flags;
5          void *sa_restorer;
6  };
```

Listing 1. Sigaction structure passed to `sigaction` in libc.

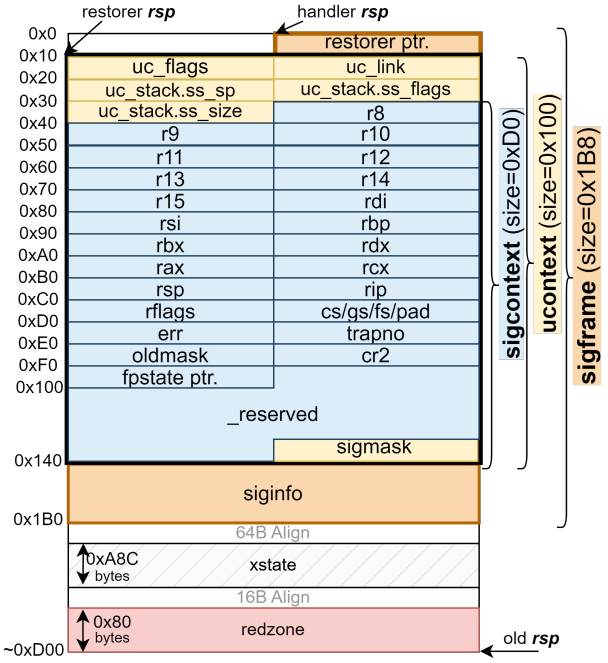

Figure 1. Complete sigframe structure as saved in the process stack.

The sigaction structure used for *act* and *oldact*, which we show in Listing 1, contains (1) a pointer to the signal handler function, (2) a mask with signals to be blocked during the execution of the handler, (3) a set of flags that alter the signal handling behaviour, and (4) a *restorer* pointer, specifying a function to be called when the signal handler returns (and that helps to return safely from the signal handler). This restorer is not intended for direct application use [28], in fact, it is always overwritten by `sigaction` internally, setting the pointer to the function `__restore_rt` in libc.

3.1.2. Delivering the Signal. When a signal is delivered to a process, to ensure that the process can safely resume execution after the signal is handled, the kernel saves the execution context (every CPU register, including the instruction pointer) before transferring control to the signal handler.

The execution context is saved in the *sigframe* structure, which contains three elements, depicted in Figure 1: (1) the *restorer* pointer fixed by libc to `__restore_rt`, (2) a

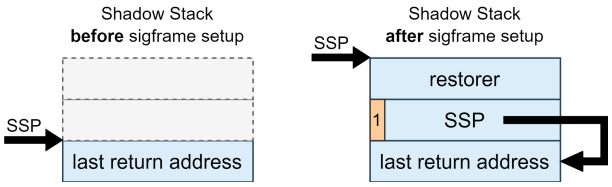

Figure 2. Shadow Stack and Shadow Stack Pointer (SSP) before and after a signal handler invocation.

ucontext structure containing a *sigcontext* structure backing the value of most CPU registers, and (3) a *siginfo* structure containing information associated with the signal. The kernel pushes the sigframe in the process stack [42].

After preparing the sigframe, the kernel transfers control to the signal handler function specified during the signal registration. Just before doing so, the kernel adjusts some registers, as to allow the signal handler to use the sigframe information: (1) the stack pointer (`rsp`) is set to point to the top of the sigframe (i.e., *handler rsp* in Figure 1), (2) the instruction pointer (`rip`) is set to the address of the signal handler function, (3) `rdi` is set to the signal number (i.e., *0xb* for *SIGSEGV*), (4) `rsi` is set to point to the *siginfo* area, and (5) `rdx` is set to point to the *ucontext* in the sigframe.

3.1.3. Returning from the Signal Handler. When the signal handler returns, it executes a `ret` instruction like any other function, popping the return address from the stack. As shown in Figure 1, this corresponds to the restorer pointer.

Any benign sigframe has the restorer set to `__restore_rt` (as libc overwrites it). This function is prepared as a safe way for the signal handler to return control to the kernel. The restorer never returns; it simply invokes the `sys_rt_sigreturn` syscall to inform the kernel that the signal handler has completed and that the execution context should be restored. During the syscall, the kernel takes the ucontext pointed by `rsp` (see *restorer rsp* in Figure 1), restoring every CPU register including the instruction pointer, effectively resuming the execution context of the process prior to handling the signal.

3.2. Intel CET during Signal Handling

The signal handling process is not transparent to CET. CET is still enforced during the signal handler and restorer, so the kernel must ensure that the shadow stack is correctly maintained across every transition [26].

The top of the shadow stack is pointed by the Shadow Stack Pointer (SSP)—just like the top of the normal stack is pointed by the stack pointer (`rsp`). When a signal is delivered, the kernel backs up the SSP prior to invoking the signal handler. For this, the kernel pushes the SSP value onto the shadow stack itself, with the most significant bit (MSB) set to "1" [39][32]. This is as to distinguish it from normal return addresses of x86_64 Linux user-space applications, which always have the MSB set to "0".

After pushing the SSP, the kernel pushes the restorer address onto the shadow stack [40]. This ensures that, when the signal handler returns to the restorer, the CET checks will succeed. We show the resulting shadow stack after both the SSP and restorer address are pushed in Figure 2.

When the `__restore_rt` restorer invokes the syscall `sys_rt_sigreturn`, the kernel restores the SSP using the value in the shadow stack. At this point, the kernel verifies that the element has MSB=1 [33], ensuring that `sys_rt_sigreturn` can only restore the SSP to a value previously saved by the kernel during the sigframe preparation. If so, the kernel zeroes the MSB and then restores the SSP, effectively to the value prior to the signal delivery (pointing to *last return address* in Figure 2).

3.3. Linux Signals Vulnerabilities

Building on our previous analysis of signal handling under Intel CET, we now present a security analysis of this mechanism. In total, we identify twelve novel weaknesses in this implementation that we combine with one previously known weakness. While none of these weaknesses are independently sufficient for exploitation, they collectively form the basis of the techniques underlying the SFOP attack.

In this section, we analyze each weakness in detail and discuss how it may induce unintended program behavior that can be leveraged by an attacker under suitable conditions. As shown later in § 4, these weaknesses can be combined in novel ways to construct powerful exploitation primitives that ultimately enable CET bypass and arbitrary code execution.

3.3.1. Known Weakness: Lack of IBT Checks. The kernel does not perform any IBT checks when invoking the signal handler itself. This implies that calling `signal` or `sigaction` with an arbitrary argument directly leads to the program reaching any arbitrary address registered as a signal handler that may not be IBT-compliant. Although this weakness is not widely known, we have found it documented in the wild [13][48]. By itself, this weakness does not lead to a general attack, as the attacker sacrifices one arbitrary call (to `signal`) to reach another target (the signal handler).

3.3.2. Late Kernel Safety Checks. In the known SROP attack, the attacker forges a sigframe on the stack and invokes the `sys_rt_sigreturn` syscall directly to restore its counterfeit context. Under CET, the kernel now checks the MSB during the syscall, preventing this attack completely. When the MSB check fails, the kernel simply kills the process sending a *SIGSEGV* signal [38]; the program receives a segmentation fault and ends. We notice that this safety check on `sys_rt_sigreturn` is done *after* the kernel has already restored the CPU registers from the sigframe [37]. This means that the program first gets all of its registers restored from the sigframe, and only then the kernel kills the program. This late abortion is severe: if the program can survive the segmentation fault triggered by this failed check, then the program would still have all of its registers restored from the sigframe, effectively allowing an attacker to hijack the control flow and registers of the program. We use this in SFOP to execute invalid `sys_rt_sigreturn` calls that restore attacker-controlled contexts.

```
1  libc_sigaction:
2    endbr64
3    push rbp
4    mov r8, rdx ; saving oact pointer in r8
5    mov rbp, rsp
6    ;copy act to kact in syscall format
7    mov rax, [rsi] ;copy from act
8    mov [rbp-0x140], rax ;copy into kact
9    lea rsi,[rbp-0x140] ; rsi now points to kact
10   ;store fixed restorer into kact
11   mov QWORD PTR [rbp-0x130], <__restore_rt>
12   ...
13   ;total: 144B copied from [rsi] to [rbp-0x140]
14   mov eax, 0xd ; "sigskip" gadget starts here
15   syscall ; sigaction syscall, kact as argument
16
17   ;copy koact to oact using userspace format
18   mov xmm1, [rbp-0xa0] ;copy from koact.sa_mask
19   mov [r8], xmm1 ; copied into oact.sa_mask
20   ...
21   ;total: 144B copied from [rbp-0xa0] to [r8]
22   leave
23   ret
```

Listing 2. Assembly pseudocode of function `sigaction` in libc [34].

3.3.3. Signal Restorer Hijacking. Since the kernel pushes the SSP with MSB=1 onto the shadow stack during the signal delivery, developers assumed that, after every signal handler, comes `sys_rt_sigreturn`. Executing any other function as a restorer always results in a segmentation fault: returning from the restorer fails the MSB check in the SSP element, since it is checked as a normal return address—only `sys_rt_sigreturn` interprets it as a saved SSP.

While it is true that libc overwrites any user-provided restorer pointer in the `sigaction` structure with the address of `__restore_rt`, and this ensures that in normal circumstances the restorer pointer in the sigframe will always point to `__restore_rt`, this does not fully prevent a program from registering a custom restorer.

We identify two scenarios where a restorer pointer different than `__restore_rt` can be realistically passed to `sigaction`: (1) when the attacker uses the `syscall` function in libc to call the `sys_rt_sigaction` syscall directly, and (2) when the attacker skips the instructions inside libc's `sigaction` that overwrite the restorer pointer using the techniques in § 3.3.1 or § 3.3.6. This corresponds to line 14 in Listing 2. We dub this the `sigskip` gadget. In both situations, the attacker passes a custom restorer pointer that the kernel blindly trusts and writes into the sigframe.

While the previous opens the path to set arbitrary restorers, its usefulness is not immediately obvious; we mentioned how any custom restorer would lead to a segmentation fault when returning from it. However, during SFOP, it is useful to inject arbitrary values into the shadow stack.

3.3.4. Shadow Stack Injection. We described how CET's shadow stack is unwritable and unreadable from userspace, preventing direct manipulation of the shadow stack. However, we find that registering custom restorers (§ 3.3.3) opens the door for injecting arbitrary values into the shadow stack, as the kernel pushes the restorer address—together with the SSP with the MSB set—onto the shadow stack.

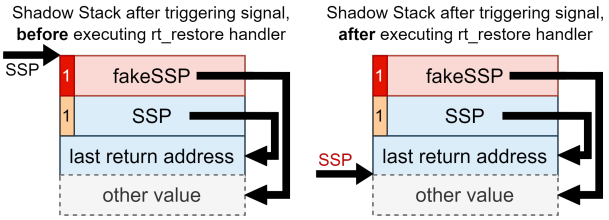

Figure 3. Hijacking the SSP by executing sigaction with a counterfeit restorer and the `__restore_rt` signal handler.

During SFOP, attackers inject values into the shadow stack to validate return addresses and successfully execute `sys_rt_sigreturn` calls.

3.3.5. SSP Hijacking. The weakness described in § 3.3.3 is not only useful for injecting an arbitrary value into the shadow stack to validate an arbitrary return address, it also opens the door for hijacking the SSP itself.

We find that the kernel does not validate whether a restorer address has the MSB during the signal registration—which should be the case, as no userspace address has the MSB set. It is therefore possible to: (1) register a signal handler with a custom restorer with the MSB set, (2) trigger the signal to push the SSP (with MSB=1) and the custom restorer (with MSB=1) onto the shadow stack, (3) since the signal handler is also arbitrary, the attacker sets it to `__restore_rt`, which invokes `sys_rt_sigreturn` and triggers the SSP restoration. At this moment, the kernel interprets the restorer address as the SSP value to restore, unsets the MSB and sets the SSP to the arbitrary attacker value. We depict this setup in Figure 3.

Hijacking the SSP can be used to control the address used for subsequent shadow stack checks. However, we identify two limitations: (1) to bypass subsequent shadow stack checks, the SSP must point to controlled memory, and (2) while the kernel only checks that the restored SSP points to userspace memory during `sys_rt_sigreturn` and can be pointed to any memory page, any subsequent use of the SSP (e.g., during a normal function return) while it points outside of the shadow stack will send *SIGSEGV* to the program, as the permissions of the page do not correspond to those of the shadow stack (see § 2.1).

Therefore, it is not possible to arbitrarily set SSP to any userspace memory. Any hijacked SSP must point to the shadow stack, and the attacker must be able to inject values into it using the techniques we described in § 3.3.4.

3.3.6. Missing CET checks. We described in § 3.3.2 how the MSB check during `sys_rt_sigreturn` prevents the SROP attack, where a forged sigframe is injected in the stack. However, if the attacker is able to overwrite a benign sigframe during the execution of a signal handler, there are no other CET-related checks performed by the kernel when restoring the CPU registers during `sys_rt_sigreturn`.

Specifically, if the attacker is able to modify the `rip` entry in the sigframe *sigcontext*, the kernel will happily restore it without any further checks. This effectively allows an attacker to bypass CET's IBT, as the destination may not be a valid indirect branch target.

While the above is a considerable weakness, it is by itself not sufficient to build a practical attack, as overwriting a sigframe can only happen during signal handling; the handler would need to present some primitive that arbitrarily modifies the sigframe in the stack with attacker-supplied values before returning to the restorer, which essentially translates to needing another memory corruption vulnerability in the signal handler—an unreasonable assumption. However, during SFOP, we bypass this limitation with the introduction of *sigframe pivoting*.

3.3.7. User-Kernel Data Collision. We described in § 3.3.3 how the kernel blindly trusting the restorer pointer passed in the `sigaction` syscall from the userspace can be a powerful primitive for an attacker. However, the userspace also blindly trusts the kernel to fill the argument-passing registers appropriately while executing the signal handler.

It is possible for an attacker to set the signal handler in some benign function of the program that was never intended to be a signal handler. If this function relies on certain values in registers `rdi`, `rsi` or `rdx`, the attacker can abuse the kernel's behavior of setting these registers during signal delivery to collide with the expected values in the function. The result is a Data-Oriented Programming (DOP) [23] [22] primitive, where the attacker can control certain values in registers when executing a benign function in the program. The attacker can call such misaligned functions using the primitives described in § 3.3.1 or § 3.3.6.

3.3.8. Signal Handler Argument Control. As described in § 3.1.1, when a signal handler is invoked, the kernel sets up three arguments in registers `rdi`, `rsi`, and `rdx`. As a reminder: (1) `rdi` is the signal number, (2) `rsi` points to the *siginfo* area, and (3) `rdx` points to the *ucontext*.

We find a minor oversight in the previous argument setup. The siginfo area is always present (see Figure 1), but it is only filled with data when the `SA_SIGINFO` flag is specified while calling `sigaction`. When the flag is not set, the kernel still sets `rsi` to point to the siginfo area, but it does not clean any values previously present there, as the kernel assumes that if the `SA_SIGINFO` flag is not set, then the signal handler will not use this argument.

This oversight allows an attacker to corrupt the siginfo area of the next signal handler's sigframe, then register the handler without the `SA_SIGINFO` flag. When the handler is triggered, `rsi` will point to a siginfo containing attacker-controlled values, effectively controlling the memory pointed by `rsi`. This is particularly useful during SFOP when combined with a user-kernel data collision.

3.3.9. Oldact Kernel Unhollowing. Similarly to § 3.3.8, the kernel does not always overwrite the memory of the `oldact` argument passed to `sigaction`. This can be used to launch a powerful user-kernel collision. We describe this issue further in § B; it is only useful for launching an advanced complementary technique in SFOP.

3.3.10. Faulty Gadgets: Deterministic Signal Triggering. As we described in § 3.1, signals can be generated by the kernel, other processes, or the same process itself.

A realistic, practical attack should not rely on any external entity (e.g., another exploited process) to trigger the signal, or calling `raise` in the program code. Still, we ideally want a signal that can be triggered deterministically (i.e., always at the moment when we want).

Segmentation faults (signal *SIGSEGV*) fit these criteria perfectly. An attacker can corrupt the memory of the process (through the original vulnerability), and easily trigger a segmentation fault at a certain instruction. This usually comes in the form of dereferencing a null or invalid pointer written into memory (e.g., `mov rax,[rax]` where `rax` is not a valid memory address). We dub such functions that contain instructions that can trigger a segmentation fault given corrupted data as *faulty gadgets*.

During SFOP, we leverage a faulty gadget in the `sigaction` function, known as the *Early Kill* (§ 4.1.2).

3.3.11. Signal Nesting: Bypassing Signal Masking. When a signal handler is triggered for a given signal, the kernel automatically *masks* (i.e., blocks) the same signal from being delivered again until the `sys_rt_sigreturn` syscall is invoked in the restorer.

We find two ways this masking can be omitted, either by re-registering the signal handler inside the handler itself (by calling `sigaction` again), or by setting the `SA_NODEFER` flag in `sigaction`, which disables the automatic masking of the signal. While both techniques are intended behavior, nesting signals increases the attacker capabilities for creating weird machines when applied to signals that can be triggered without external intervention, such as *SIGSEGV* as we showed in § 3.3.10. Combined with the ability to control handlers and restorers (see § 3.3.2), an attacker can create complex signal handling chains containing faulty gadgets.

We use both techniques in SFOP to create an *implicit dispatcher* that loops every time a faulty gadget is executed. An important observation is that, unlike other implicit dispatchers, the kernel modifies the values of `rdi`, `rsi`, and `rdx` every time a signal is triggered (see § 3.3.8). Therefore, the attacker may only rely on other registers (e.g., `rbp`) to maintain their value across nested signal handlers.

3.3.12. Surviving CET Kills. Every time the kernel detects a shadow stack violation, it kills the process by sending a *SIGSEGV* signal. This is counterproductive against SFOP, because it turns every violation into a faulty gadget (see § 3.3.10) that can be deterministically triggered by the attacker. The attacker can call any function in the program skipping its prologue using weaknesses § 3.3.1 or § 3.3.6, execute it without secondary effects, and trigger a faulty gadget when returning from the function, as the return

address is popped during the function epilogue and the shadow stack gets compared to the invalid memory below.

This unsafe signal choice is not unique to CET violations; other security measures such as stack canaries also trigger catchable signals like *SIGABRT* upon a violation. During our experimentation, we found that restricted forms of SFOP can be built using these other signals as well.

3.3.13. Insecure Doppelgängers. Libc acts as the interface between userspace programs and kernel syscalls; almost every program must go through it to invoke syscalls. However, the dynamic linker *ld.so* is loaded by the kernel before libc itself, as it is in charge of loading the libraries, including libc. For this reason, *ld.so* also contains and uses its own implementations of many libc functions, including `sigaction`, which are identical to the ones in libc in terms of functionality and assembly code. However, we find one interesting difference: *ld.so* functions never contain stack canary checks, while libc functions do. The reason is that *ld.so* initializes the stack canaries during program startup, so it cannot rely on them being present in its own code.

In SFOP, we will not be overflowing any stack buffers, so SFOP can leverage both the libc and *ld.so* implementations of `sigaction` and any other functions interchangeably. However, if the attacker uses the IBT-bypass techniques described in § 3.3.1 or § 3.3.6 to jump to the `sigskip` gadget for registering a custom restorer, the attacker must account for the faulty gadget being triggered during the failed canary check, not at the function's final `ret`. In this case, the triggered signal is not *SIGSEGV* but *SIGABRT*.

We will use *ld.so*'s `sigaction` whenever we want to leverage the `sigskip` gadget, as it is more straightforward to build a chain using only `SIGSEGV`s. We have verified the presence of this function across major Linux distributions (Debian, Ubuntu, Arch Linux), including from libc version 2.39 (when CET was enabled at runtime) to the newest 2.42.

4. Segmentation Fault-Oriented Programming

Segmentation Fault-Oriented Programming (SFOP) is a code-reuse attack that leverages the previously unexplored weaknesses we identified in the Linux signal handling mechanism and Intel CET. SFOP turns signal handling into a powerful implicit dispatcher in a CET-enforced scenario.

Challenges in SFOP: To design SFOP, we have to overcome several fundamental challenges. Most importantly, the SFOP dispatcher is locked behind the ability to register a custom signal handler in the first place. While we assume the attacker can corrupt a forward-edge pointer, this does not immediately allow them to control function arguments. For SFOP to be general and applicable to any program, we must find a reliable method to set the arguments for `sigaction` without relying on any program particularities.

Moreover, even after registering a custom signal handler, building a practical and general dispatcher is not straightforward. Some of the challenges that attackers face are: (1) *Signal Delivery*: How can the attacker reliably trigger the signal handler after registering it, without requiring binary-specific gadgets? (2) *Register Resetting*: Unlike classic dispatchers, where a loop does not alter the value of the registers, the kernel resets the value of every argument-passing register every time a new signal is triggered; the attacker cannot rely on controlling `rdi`, `rsi` or `rdx` during the signal handler or restorer execution. (3) *Dispatcher Soft-lock*: Without re-registering the signal handler within itself, the attacker is stuck executing the same handler and restorer repeatedly. (4) *Memory Clobbering*: Even if the attacker controls both the signal handler and restorer, executing a function in the handler inevitably corrupts stack memory and alters register values by the time the restorer is reached, rendering it unreliable and practically useless.

We build SFOP in such a way that we overcome both the problem of the signal registration and the other dispatcher-related challenges. SFOP is built with generality in mind, i.e., it does not require any further vulnerabilities to be triggered (after the initial forward-edge pointer hijack) or any other particularities in the target program. The techniques we show can be applied in every scenario, independently of the attacked binary, as long as it is CET-protected.

SFOP Variants: We develop two main variants of SFOP, which combine the weaknesses we described in § 3.3 in different ways: (1) ***Forward SFOP*** (***fSFOP***), where the attacker abuses faulty gadgets to build a dispatcher (i.e., CET kills the program in a loop) and, (2) ***Backward SFOP*** (***bSFOP***), where the attacker takes control of the shadow stack to validate an arbitrary exploit control flow.

Both fSFOP and bSFOP share a key advantage of SFOP attacks: their payloads are *stack agnostic*. This means that, every time a signal handler is executed, the attacker is free to choose where in memory the payloads are positioned; the attacker does not need to write the payload contiguously, nor does it need to position them greatly separated (to account for stack growth during a signal handler execution). We discuss this property in detail in § D.

4.1. Forward SFOP (fSFOP)

In fSFOP, the attacker leverages faulty gadgets and the diverse killing weaknesses of the kernel (see § 3.3.2, § 3.3.12) to create a dispatcher. In essence, the attacker abuses *SIGSEGV* signals triggered by faulty gadgets to repeatedly execute a signal handler where, by means of a technique known as *sigframe pivoting*, the attacker reliably gains control over the sigframe used during the subsequent restorer `sys_rt_sigreturn` call. This way, the attacker can chain multiple calls to arbitrary functions with arbitrary arguments, as the sigframe allows full control of every CPU register. We show a sketch of fSFOP in Figure 4.

4.1.1. Initial Signal Registration. fSFOP attacks start by registering a custom signal handler for *SIGSEGV*. However, for this, the attacker only has a hijacked forward-edge pointer and corrupted memory; the attacker cannot directly call `sigaction` with controlled arguments.

As it turns out, these primitives are sufficient to call `sigaction` with controlled arguments by means of chain-

Table 2. SIMPLE CHAIN TO GO FROM CORRUPTED MEMORY, TO CALLING `SIGACTION` WITH TWO ARBITRARY ARGUMENTS.

Order	Function name	Library	Action
1	__pthread_once_slow	*libc.so*	Load `rdi` from memory
2	_obstack_newchunk	*libc.so*	Load `rsi` based on `rdi` Clobber `rdx`

ing CET-compliant gadgets ending in indirect branches. This idea of using indirect branches to chain gadgets has been explored in non-CFI enforced scenarios with COP or JOP [45] [7], as well as to bypass CFI protections *without* a shadow stack in combination with ROP-style return gadgets [21][10]. However, to the best of our knowledge, we show here for the first time that indirect branches *alone* are sufficient to set *at least* two argument-passing registers in a x86_64 Linux system protected by CET, given a hijacked forward-edge pointer and corrupted memory. Moreover, we show that this technique can be used even in minimal programs, where the attacker has limited number of gadgets to choose from.

The gadgets we chain consist of any code sequence that starts with an *endbr64* instruction and ends with an indirect branch (i.e., an indirect call or jump instruction). We show in § E some sample gadgets. The first gadget is called via the initially hijacked forward-edge pointer, while each subsequent gadget is reached via the indirect branch of the previous gadget. Every indirect branch at the gadget end computes its target from the value stored or pointed by a register, which the attacker must control to chain gadgets together. We show in § F the number of CET-compliant indirect branch gadgets existing in popular libraries.

A complete chain consists of gadgets which, when executed sequentially, result in at least two argument-passing registers (`rdi`, `rsi`) being set to attacker-controlled values before calling an arbitrary function using the final gadget. Register values are set using the instructions in each gadget to load values from memory (which the attacker controls) or between registers (once one register is controlled).

The chains we build are *dispatcher-less*; unlike complete code-reuse attacks, there exists no dispatcher to chain gadgets together. This is because loops calling function pointers will be those from FOP-style explicit dispatchers; as discussed in § 2.4 come with the disadvantage of needing to overwrite a specific dispatcher table in memory with the vulnerability. Instead, each gadget directly transfers control to the next gadget in the chain using its final indirect branch. Hence, the chain ends with the final transition to the attacker-chosen function, which restricts its utility to *only* calling a single arbitrary function with controlled arguments; after that, the attacker loses control of the execution flow.

Together with the hijacked pointer and corrupted memory, an attacker may control *some* register before calling the first gadget (e.g., because some corrupted memory is loaded into a register prior to executing the hijacked pointer). This increases the range of possible gadgets with which to start the chain, as some gadgets may rely on such registers to

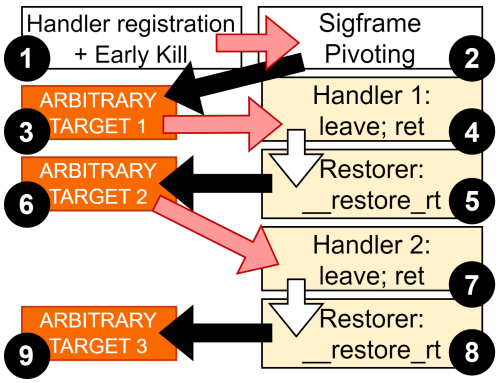

Figure 4. fSFOP simplified. Black arrows represent calls to sigreturn with attacker-controlled sigframes, red arrows represent SIGSEGV signals triggered, and white arrows represent normal return instructions.

control the target of their indirect branch. Nevertheless, we show in Table 2 an example of chain that requires no registers controlled, but exclusively corrupted memory, using only gadgets found in libc. Since every program links, at a minimum, the libc and ld libraries (statically linked binaries are exceptions), building a chain with only libc shows that chains are realistic in every program. We show two additional gadget chains during our real-world SFOP exploits in § 5; additional chains can be consulted in our published PoC exploits. Each chain we show can be used to call `sigaction` with two attacker-controlled arguments, as well as clobbering `rdx` for purposes we describe next.

Note that our dispatcher-less chain can also be used to call a single arbitrary other function instead (e.g., `system`). While attackers could attempt to compromise systems with single-function exploits, we discuss in § H that exploits in practice typically require (or at least significantly benefit) from *multiple* arbitrary function calls, which was also the goal of all other exploitation techniques discussed in § 2.4. The next steps of the fSFOP attack thus present techniques for turning a single arbitrary function call into multiple.

4.1.2. Early Kill. After converting the initial forward-edge pointer hijack into a call to `sigaction` with custom arguments, we register a custom signal handler for *SIGSEGV* ❶. However, to invoke this signal handler, the attacker must crash the program with a *SIGSEGV* signal as soon as possible, ideally even *before* returning from `sigaction` (to avoid that SFOP depends on whatever function is executed next) and without relying on any external factors, such as relying on returning to another function with a faulty gadget.

The most reliable method for triggering the signal handler right after its registration is to exploit a faulty gadget in `sigaction` itself. This faulty gadget can be easily triggered right after the handler is registered by putting a non-null invalid memory value into register `rdx`, in other words, passing an invalid *oldact* structure. This register is clobbered during the initial gadget chain, as with the one shown in Table 2. This way, `sigaction` causes a *SIGSEGV* right after it registers the attacker-provided signal

handler (line 19 in Listing 2). We name this technique where the attacker can register a signal handler *and* trigger a *SIGSEGV* immediately after as the *Early Kill*.

We have verified that attackers can leverage the Early Kill faulty gadget in every version of libc released since its adoption of Intel CET (glibc 2.39) until the most recent at the time of writing (glibc 2.42). Note that other faulty gadgets may also be used, using `sigaction` is just the most straightforward and reliable method.

4.1.3. Initial Sigframe Pivoting. With the Early Kill gadget, the attacker executes an arbitrary signal handler ❷. At this point, we are guaranteed that subsequent *SIGSEGV* signals will trigger the same signal handler again; the attacker can now focus on executing arbitrary function calls.

The best technique for this purpose is to leverage `sys_rt_sigreturn` syscalls with counterfeit sigframes, as this allows full control of every CPU register, including `rip` and every argument-passing register. However, corrupting a sigframe with arbitrary data is not straightforward, as the attacker must do it using only the signal handler. Prior to the handler, the kernel has not yet positioned the sigframe, and afterwards the restorer cannot overwrite the sigframe as it is occupied executing `__restore_rt`.

To this end, we introduce *sigframe pivoting*, a technique that allows the attacker to control every element in the sigframe using an arbitrary signal handler. We take advantage of the fact that the `sys_rt_sigreturn` syscall expects the `rsp` register pointing to the ucontext area (see *restorer rsp* in Figure 1). If the attacker manages to alter `rsp` during the signal handler, the kernel will happily restore every register from the new `rsp`-pointed sigframe.

We identify three reliable methods to achieve this.

Classic Sigframe Pivoting. The most straightforward way to change `rsp` is to leverage the typical stack pivoting techniques used in ROP attacks. In this case, the attacker sets the signal handler to be a `leave; ret` gadget, which sets `rsp` to `rbp` and pops the top of the stack into `rbp`. By filling the memory pointed by `rbp` with a counterfeit sigframe during the initial memory corruption vulnerability, the attacker now has full control of the ucontext used during the subsequent restorer `sys_rt_sigreturn`. We illustrate this technique in Figure 5.

Classic sigframe pivoting has one major limitation: even if the attacker uses the memory corruption vulnerability to overwrite the memory that will be pointed by `rbp` during the signal handler execution, there is no guarantee of this memory remaining unaltered until the time the signal handler is executed. In fact, many functions push the values of `rbp` and other registers on the stack as part of the function prologue, overwriting the corrupted memory. This prevents the attacker from reliably using classic pivoting in some scenarios, but particularly during the handler executed after the initial `sigaction` of the Early Kill, as the memory pointed by `rbp` is overwritten during `sigaction`'s prologue.

Unwinding-based Pivoting. This is a more powerful version of classic sigframe pivoting that works even when `rbp` is overwritten in the previous stack frame, so it is viable

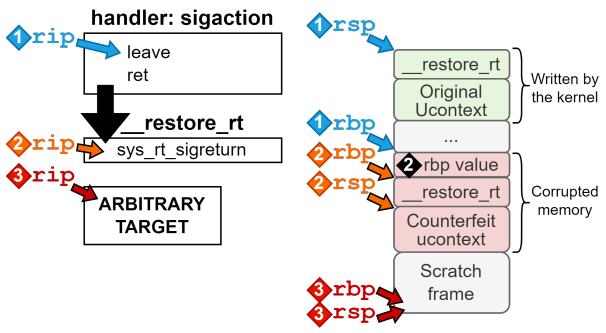

Figure 5. Classic Sigframe Pivoting simplified. ❶ shows state prior to the pivoting gadget, ❷ shows state after the *leave; ret* instructions, and ❸ shows state after the subsequent `sys_rt_sigreturn`.

after the Early Kill. Typically, when a nested function is executed, `rbp` and other registers are pushed into the stack during the function prologue (`push rbp`), and `rbp` is set to `rsp` (`mov rbp, rsp`). If multiple functions are nested, and only `rbp` is pushed, then every `rbp` value will be a pointer to the previous `rbp` value in the stack, creating a single-linked list of `rbp` values. This list ends whenever the function prologue pushes another register together with `rbp`, the function does not include a function prologue, or when we reach the maximum-depth stack frame.

To exploit this behaviour, the attacker can register a `leave; ret` gadget as the signal handler during the Early Kill, and fill the memory at the end of the single-linked `rbp` list with an arbitrary value. As this value will typically be in a deeper stack frame than the stack frame where the initial forward-edge pointer hijacked is located, the attacker can commonly trust this value not to be overwritten.

Next, a nested sequence of `leave; ret` signal handlers is executed, each one crashing with *SIGSEGV* at the end, as `rbp` is set to the previous `rbp` value in the stack, while `rsp` points to some invalid memory at the time of the `ret`. This continuously triggers nested signal handlers if the attacker has used the `SA_NODEFER` flag. Eventually, the hijacked value of `rbp` at the end of the linked list is moved into `rsp`, allowing the attacker to control both `rsp` and the return address. This effectively allows the attacker to return to `__restore_rt` and control the sigframe used during the subsequent `sys_rt_sigreturn`.

While unwinding-based pivoting is more powerful than classic sigframe pivoting, it comes with the limitation that the final element at the end of the `rbp` linked list must not be overwritten until the restorer executes `sys_rt_sigreturn`. For example, if a stack frame later clobbers the attacker-modified memory, this breaks the pivot.

Double-Partial Sigframe Pivoting. To address the restrictions of the previous two techniques, we introduce *double-partial sigframe pivoting*, the most powerful pivoting technique that does not rely on `rbp` pointing to controlled memory. It can be used in every scenario, including after the Early Kill. We describe this technique in § C.

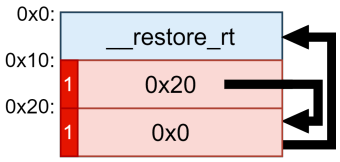

Figure 6. Recursive shadow stack example.

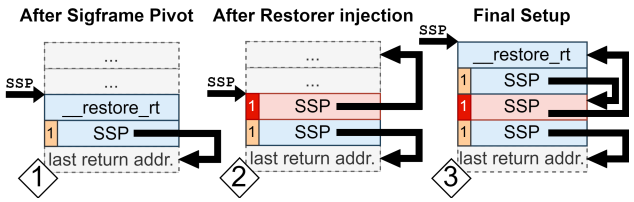

Figure 7. Shadow stack grooming during the bSFOP attack.

4.1.4. Infinite Segfault Looping. After the initial sigframe pivoting process, the attacker reaches the restorer with a fully controlled sigframe. This restorer, fixed to `__restore_rt`, executes the `sys_rt_sigreturn` syscall; because the sigframe is pivoted, the attacker gains control over every register, including `rip`. In other words, the attacker now calls the desired arbitrary function ❸, fully controlling the six argument-passing registers.

Once the target function returns, the function crashes due to a CET shadow stack check, as the function was not called but reached via `sys_rt_sigreturn` (see § 3.3.2). This triggers another *SIGSEGV* signal, handled by the same pivoting gadget registered previously ❹. Since the attacker could control `rbp` in the previous `sys_rt_sigreturn`, and any arbitrary function we executed is guaranteed to preserve this value (`rbp` is either not modified, or preserved in the prologue and restored in the epilogue), the signal handler can be again used for sigframe pivoting, allowing the attacker to reach another `__restore_rt` ❺ with a fully controlled sigframe again. Using this arbitrary `sys_rt_sigreturn`, the attacker executes another arbitrary call ❻, infinitely repeating the process ❼❽❾.

4.2. Backward SFOP (bSFOP)

In bSFOP, the attacker combines shadow stack injection (§ 3.3.4) and SSP hijacking (§ 3.3.5) to set up a counterfeit shadow stack that validates the exploit control flow. However, due to the restrictions on the SSP hijacking technique (SSP may not point out of the shadow stack memory), the attacker—who ideally wants to execute a ROP chain—must be able to push every return address used during the exploit to the shadow stack. We show in Figure 8 a sketch of bSFOP.

Unfortunately, the shadow stack injection primitives are also too limited: (1) if the attacker calls a function, the function must eventually return; while its address remains in the shadow stack after returning, the SSP now points to a higher memory address and will overwrite the entry at the next occasion; (2) if the attacker injects addresses by setting custom restorers, this pushes the restorer address but also the customary SSP with MSB=1, which will never be a valid return address and crash any ROP chain.

For these reasons, it is impossible to construct a counterfeit shadow stack that validates an arbitrary ROP chain. However, we find that it is possible to build a counterfeit shadow stack that validates SROP chains, i.e., calls to `sys_rt_sigreturn`. For this, the attacker must construct what we call a *recursive shadow stack*.

We show in Figure 6 the shortest recursive shadow stack that can be practically built in any program. As it can be seen, the SSP loops infinitely between three addresses, which allows the attacker to execute any number of `sys_rt_sigreturn` calls infinitely many times: (1) when the SSP points to the topmost entry, the attacker simply needs to return to `__restore_rt`, which is easy as it is the default restorer every signal handler returns to; (2) when the SSP points to the second entry, the attacker is free to execute `sys_rt_sigreturn`, which will restore the SSP to point to the bottom entry; (3) when the SSP points to the bottom entry, the attacker may again execute `sys_rt_sigreturn`, restoring the SSP to the topmost entry of the recursive stack and restarting the loop.

Naturally, two challenges arise: (1) how to build the recursive shadow stack in the first place, and (2) how to leverage the recursive shadow stack for continuously calling `sys_rt_sigreturn` with a pivoted sigframe. For this, we follow a similar step as with fSFOP, but with an additional step for grooming the shadow stack.

4.2.1. Initial Signal Registration and Early Kill. Just like with fSFOP, bSFOP starts by ❶ using a gadget chain to reach the Early Kill in `sigaction`. This is used to register a signal handler for *SIGSEGV* and immediately trigger the signal using `sigaction`'s faulty gadget.

4.2.2. Initial Sigframe Pivoting. The arbitrary signal handler that is triggered ❷ is used for sigframe pivoting using one of the three techniques (classic, unwinding or double-pivot). However, these pivots must come with a difference: the attacker not only wishes to control the sigframe, but also to groom the shadow stack to build the recursive shadow stack, which as we mentioned in § 3.3.4 requires registering a restorer. For this reason, instead of directly using `leave; ret` as the pivoting gadget, we use the `sigskip` gadget that we described in § 3.3.3 to re-register the signal handler with a custom restorer *and* pivot the sigframe. As a reminder, this gadget consists of the usual `sigaction` function but skipping the instructions in libc that fix the restorer to `__restore_rt`, allowing the attacker to pass a custom restorer to the `sys_rt_sigaction` syscall. This re-registration overwrites the previous handler; the next time the signal is triggered, the new handler and restorer are used.

Leveraging the `sigskip` gadget requires a user-kernel data collision (see § 3.3.7), in combination with the signal handler argument control weakness (see § 3.3.8). In essence, we take advantage of the kernel not hollowing the contents

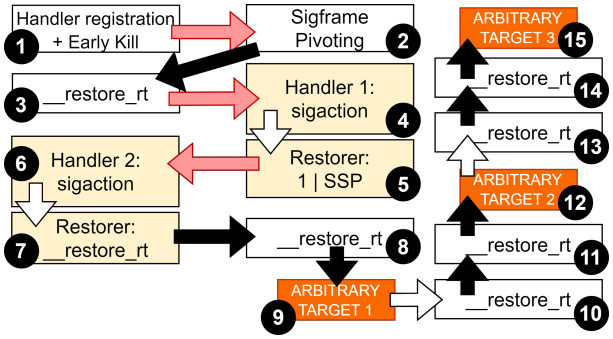

Figure 8. bSFOP simplified. Black arrows represent calls to sigreturn with attacker-controlled sigframes, red arrows represent SIGSEGV signals triggered, and white arrows represent normal return instructions.

of the siginfo area in the sigframe when the `SA_SIGINFO` flag is not set. The data collision occurs as follows: (1) `rdi` is set by the kernel to the signal number, while `sigaction` expects it to be the signal number as well; (2) `rsi` is set by the kernel to point to the siginfo area, while `sigaction` expects it to point to the `act` structure; and (3) `rdx` is set by the kernel to point to the ucontext in the sigframe, while `sigaction` expects it to point to the `oldact` structure.

This collision is perfectly suited for registering a new signal handler, as `rdi` already holds the appropriate signal number for registering the handler with the same signal, and `rsi` points to arbitrary memory (since the kernel did not hollow the area), which we can use to set up a counterfeit `act` structure as to register an arbitrary handler and restorer.

Using the previous collision, together with the simultaneous sigframe pivoting, the attacker gains control of three key elements: (1) the next signal handler is registered to `sigskip` again, (2) the next restorer is set to be a recursive SSP with MSB=1, and (3) the pivoted sigframe will set `rip` to `__restore_rt` and controls every register.

4.2.3. Shadow Stack Grooming. Once the pivoted sigframe is restored by the kernel during `sys_rt_sigreturn`, the attacker reaches the targeted `__restore_rt` function ❸. Now comes a crucial step for bSFOP: a series of signal handlers whose sole purpose is to groom the shadow stack as to build the recursive shadow stack we described earlier. We describe the shortest and easiest grooming sequence, but longer sequences are possible as well.

Firstly, in the `__restore_rt` function we reached, we can directly call `sys_rt_sigreturn`: the sigframe is automatically pivoted since the attacker already controlled `rsp` during the previous sigframe pivoting. The purpose of executing `sys_rt_sigreturn` again is simply to trigger a faulty gadget; as we explained in § 3.3.2, the kernel restores the sigframe but fails to pass the SSP check, triggering another *SIGSEGV* signal. On top of this, the pivoted sigframe allows the attacker to control `rsp` for the next signal handler, maintaining stack agnosticism (see § D).

Now is where our malicious handler and restorer address we registered during the initial sigframe pivoting come into play, as the kernel pushes to the shadow stack the restorer address and a SSP with MSB=1. This restorer acts as the last entry of the recursive shadow stack we want to build, obtaining the intermediate setup (2) shown in Figure 7.

At this point, the attacker finds themselves in ❹ a signal handler with ❺ an invalid restorer (due to the MSB being set). This causes yet another *SIGSEGV*, invoking the ❻ second handler we show in Figure 8. This also pushes `__restore_rt`'s address and a valid SSP onto the shadow stack, completing the recursive shadow stack and obtaining the final (3) setup shown in Figure 7.

During the execution of this second handler, the attacker performs a final classic sigframe pivoting, gaining control over the sigframe used during the ❼ subsequent restorer. The attacker uses this hijacked `sys_rt_sigreturn` to call the function `__restore_rt` ❽, marking the start of the infinite bSFOP loop, as now the shadow stack will validate any number of `sys_rt_sigreturn` calls infinitely.

4.2.4. Infinite Sigreturn Looping. Once in the final `__restore_rt`, the attacker can freely call its first arbitrary target with arbitrary arguments ❾. This pops the forged SSP value from the bottom entry of the recursive shadow stack and loops over to the topmost entry.

Once the arbitrary function returns, the `__restore_rt` address is popped from the shadow stack, and the attacker (which controls `rsp` and `rbp`) is free to return to ❿ `__restore_rt` again, where it can proceed to pop the second entry of the shadow stack using a dummy `sys_rt_sigreturn` call, continuing with yet another ⓫ `sys_rt_sigreturn` call used to jump to the ⓬ next arbitrary function with arbitrary arguments. We show this infinite looping (two `sys_rt_sigreturn`, one arbitrary call) in Figure 8 ⓭⓮⓯.

4.3. fSFOP vs. bSFOP

Both fSFOP and bSFOP are powerful techniques for calling infinitely many arbitrary functions with arbitrary arguments in a CET-protected system. While fSFOP leverages faulty gadgets to create a dispatcher, bSFOP uses shadow stack manipulation to validate the exploit control flow.

We find fSFOP generally easier to set up, as the bSFOP grooming process requires more steps. However, bSFOP has the advantage of not relying on a dispatcher made of faulty gadgets, which simplifies the exploit development process after the grooming is done (e.g., the attacker does not need to account for the kernel overwriting memory with useless sigframes), and increases stealthiness by avoiding continuous repeated crashes due to CET kills. Nonetheless, fSFOP is also capable of working in scenarios where *signal counting* is enforced—the `sys_rt_sigreturn` syscall is restricted to only be leveraged after a signal handler is called—which is a non-implemented defense proposed to patch SROP (prior to the introduction of CET) [8].

5. Evaluation

We implement six PoC exploits in synthetic programs to showcase the two varieties of SFOP and the techniques described in § 4. In addition, as to demonstrate the practicality of SFOP, we implement two complete exploits leveraging real-world vulnerabilities in popular applications. All the code, artifacts and materials are available in our GitHub repository *https://github.com/signal-sfop/sfop*.

5.1. Nginx PoC

We developed a PoC exploit for one of the most popular web servers, Nginx. For this PoC, we picked Nginx version 1.4.0, vulnerable to the CVE-2013-2028 stack-based buffer overflow vulnerability. This vulnerability is well documented and has been exploited using ROP, as can be seen in its Metasploit module [12]. In this case, we now demonstrate that this vulnerability can still be exploited in a CET-enforced environment using fSFOP.

The CVE-2013-2028 vulnerability resides in the `ngx_http_parse_chunked` function, which parses chunked HTTP requests from the network. This function suffers from an integer signedness error whenever a negative content length is specified in the HTTP request. As a result, the function writes outside a buffer allocated on the stack, leading to corruption of stack memory.

Our exploit, consisting of a malicious HTTP request, triggers the vulnerability and overwrites stack memory including a forward-edge pointer in a deeper stack frame, at the function `ngx_http_finalize_request`. When nginx processes such pointer, it leads to a forward-edge control-flow hijack. Using this pointer, we leverage a gadget chain to reach our Early Kill `sigaction`. The gadget chain uses only one gadget, `__pthread_cleanup_routine`, as `rdi` is by chance already pointing to some stack area, which simplifies the chain. After this gadget, `sigaction` is reached and the first signal handler is executed.

Once in the first signal handler of fSFOP, we perform a double-partial pivot and proceed with the fSFOP chaining as described in § 4.1. In this exploit, an infinite number of functions can be executed with arbitrary arguments; without losing generality, we proceed to execute three distinct commands using `execve` once and `system` twice.

5.2. Ladybird PoC

We developed a second PoC exploit for Ladybird, a popular open-source web browser for Linux. For this PoC, we reintroduced CVE-2021-4327 in the latest version of Ladybird—an integer overflow vulnerability that can be exploited for corrupting arbitrary memory. As with Nginx, this vulnerability has been exploited using ROP; now we enforce Intel CET and demonstrate the exploit using fSFOP.

The vulnerability resides in the typed array parsing functionality of the internal Javascript engine of the browser, found in its *LibJS* library. This module suffers from an integer overflow that results in corrupting process memory. In our PoC, we create a malicious web page that triggers the overflow and allows us to overwrite a forward-edge pointer in the program stack. When the browser processes such pointer, during the `get_identifier_reference` function, it leads to a control-flow hijack. We use this initial foothold to leverage a gadget chain of two gadgets (`hb_color_line_get_extend` and `duk_realloc_raw`) and reach our Early Kill.

Once the first signal handler is executed, we perform a double-partial pivot and proceed to follow fSFOP to execute three arbitrary calls with arbitrary arguments; without losing generality, we execute `system("whoami")` three times.

5.3. Synthetic PoCs

In addition to the two real-world exploits, we implement six PoC exploits in synthetic programs to demonstrate the techniques described in § 4 and to showcase both varieties of SFOP. These programs are intentionally vulnerable to a memory corruption vulnerability that overwrites memory in the stack, including a single forward-edge pointer. The artifacts include: (1) One PoC to showcase how to register a malicious signal handler even when the attacker may only corrupt limited memory; (2) two fSFOP PoCs: using an Early Kill, and using a prepared gadget for registering the initial signal handler; and (3) three bSFOP PoCs: using an Early Kill, using a prepared gadget, and with a multi-stage payload demonstrating how attackers can leverage a single call to receive further payload from the attacker.

6. Defenses

SFOP attacks exploit specific weaknesses in the signal handling mechanism when mixed with Intel CET, and rely on the ability to register signal handlers, even if such mechanism is available by default. Therefore, we can distinguish two main categories of defenses: (1) defenses that harden the signal handling mechanism itself and (2) defenses that restrict the ability of an attacker to register signal handlers.

6.1. Hardened Signal Handling

CET Enforcement in User-Kernel Transitions. IBT checks are performed automatically upon indirect branches; however, additional checks should be incorporated by the kernel—both before preparing the sigframe for a signal handler and before processing the sigframe in the `sys_rt_sigreturn` syscall (see § 3.3.1 and § 3.3.6).

Restorers and Shadow Stacks. Custom restorers do not currently work with CET, so their use should be deprecated altogether on CET-enabled systems as to prevent attacks on the shadow stack (§ 3.3.5, § 3.3.4). Alternatively, the kernel should at least verify that no restorer has MSB=1 as to prevent interpreting it as a shadow stack pointer.

Improved Kernel Logic. In weakness § 3.3.2, we highlight how the kernel performs certain checks too late in the sigframe restoring process, under the assumption that

the order of the operations is not critical. However, as we show, it can have realistic security implications if the process manages to catch a signal from the kernel. We suggest that the `sys_rt_sigreturn` syscall only restores the sigcontext *after* the shadow stack has been verified.

User-kernel Collisions. Collisions (§ 3.3.7) are unfortunately an inherent weakness of the existing signal handling mechanism and cannot be easily fixed, since any function can be registered as a signal handler. However, their effect can be mitigated by ensuring that the kernel actively wipes (i.e., zeroes) any memory that is not intended to be used, such as the siginfo area, reducing collisions with attacker-controlled data (§ 3.3.8, § 3.3.9).

Sigframe Hardening. Even if CET prevents counterfeit sigframes from being restored, benign sigframes can still be corrupted to divert control flow. To prevent this, we propose the same mitigation as for regular SROP attacks: The kernel could embed a secret value in the sigframe when delivering a signal and verify it upon `sys_rt_sigreturn`, similarly to how stack canaries are used [9].

Signal Re-registrations and Catching. Unless strictly necessary, signals commonly used to indicate serious faults should not be allowed by the kernel to be re-registered inside a signal handler or nested in any way (§ 3.3.11). This includes `SIGSEGV` and `SIGABRT`, which are useful to build faulty gadgets. In addition, the kernel should not send catchable signals to a process that is misbehaving (e.g., causing a CET violation), as this gives the process an opportunity to bypass the protection. Instead, the kernel should directly terminate the process in such cases (e.g., by sending uncatchable signals such as `SIGKILL`).

6.2. Preventing Signal Handler Installations

Signals are fundamental to Unix-like systems and widely used by applications. However, for programs that do not require signal handlers, restricting handler registration—such as by blocking the `sigaction` and `signal` syscalls—can prevent SFOP. The most straightforward approach is to use *seccomp* to block these syscalls.

Unfortunately, seccomp requires explicit opt-in, which few applications use, and even those actively using it often explicitly allow `sigaction` by default, including Chromium, Firefox, and Docker in their default seccomp filter configuration [11] [16] [14]; real-world applications very rarely use the *strict* seccomp configuration that would block `sigaction` and `signal`.

6.3. SFOP Linux Kernel Patches

As detailed in § 8, we initiated a coordinated disclosure process with the Linux kernel security team to provide developers the opportunity to implement mitigations for the weaknesses described in this work and patch the SFOP attack. At the time of publication, the Linux kernel security team has acknowledged our findings and we have prepared a patch series that is currently under review and will foreseeably be merged in the near future.

6.3.1. Implemented Patches. Our patch series hardens the signal handling mechanism by implementing the following mitigations proposed in § 6.1: (1) *Restorers and Shadow Stacks*: the kernel now verifies that no restorer has MSB=1. (2) *Improved Kernel Logic*: during `sys_rt_sigreturn`, the sigcontext is restored only after the shadow stack is verified. (3) *User-kernel Collisions*: the siginfo area of the sigframe is zeroed when a signal registered without the `SA_SIGINFO` flag is triggered. (4) *Signal Catching*: the kernel now triggers `SIGKILL` instead of `SIGSEGV` when a CET violation is detected during signal handling.

6.3.2. Result of the Patches. Our patch series fixes weaknesses § 3.3.2, § 3.3.3, § 3.3.4, § 3.3.5, § 3.3.8 and § 3.3.12. As a result, bSFOP is fully patched, and fSFOP is partially mitigated, mainly because the `sigaction` faulty gadget no longer presents a useful user-kernel collision, which limits the advanced sigframe pivoting techniques.

6.3.3. Patching fSFOP. Fully patching fSFOP requires enforcing the user-kernel transitions as proposed in our *CET Enforcement in User-Kernel Transitions* and *Sigframe Hardening* mitigations. However, these remain a significant engineering effort as they ultimately arise from design limitations in Intel CET: IBT checks are done by the hardware, while the kernel merely reacts to violations (see § 2.1). Preventing fSFOP would require software IBT checks in the kernel during signal delivery and protection for the saved `rip` in the sigframe, which is not shadow stack protected. We have raised attention to these issues and our proposed mitigations to the Linux kernel developers, but they require a substantial redesign of the signal handling mechanism and thus may not be easily implemented in the short term.

6.4. Other Defenses Against SFOP

SFOP has been designed to bypass Intel CET in x86_64 systems, and the patches described in § 6.3 are specific to the Linux kernel. Still, we believe that SFOP may be applicable to other POSIX-compliant OSes (that use similar signal handling mechanisms) and other architectures protected by CET or similar coarse-grained CFI mechanisms. As a general mitigation against SFOP in these unpatched systems, we recommend the use of stricter seccomp filters that block `sigaction` when unnecessary (see § 6.2), as well as the possible use of finer-grained CFI mechanisms.

7. Conclusion

We presented SFOP, a novel code-reuse attack that exploits novel weaknesses in signal handling and Intel CET. SFOP enables powerful exploits that evade CET in any program, escalating a single hijacked pointer into an infinite sequence of arbitrary calls with arbitrary arguments. It is currently among the lowest hanging fruits for an attacker, as it does not require any particular features in the exploited application. We demonstrated SFOP's practicality with PoC exploits for two popular programs, and proposed mitigations to harden signal handling and prevent SFOP attacks.

8. Ethics Considerations

All experiments carried out in this paper were done in an isolated manner against locally-hosted services. We did not interact with real-world systems to avoid any damage.

Since this paper shows major weaknesses in the Linux kernel's signal handling implementation when combined with Intel CET, we initiated a coordinated disclosure process with the Linux kernel security team to provide developers the opportunity to implement our proposed mitigations. We shared this paper and our PoC exploits with the security team immediately after submission. We withheld public release of our paper and artifacts during the disclosure process, ensuring developers have sufficient time to address the issues before the paper publication.

At the time of publication, the Linux kernel security team has acknowledged our findings and is reviewing our patch series that patches bSFOP and mitigates fSFOP.

We declare no conflicts of interest or other ethical issues. We have not received funding from the affected party.

References

[1] LMS Phrack 40. Bypassing cet & bti with functional oriented programming. https://phrack.org/issues/71/7_md. (29-10-2025).

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery.

[3] Marcos Bajo and Christian Rossow. Await() a second: evading control flow integrity by hijacking c++ coroutines. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, USA, 2025. USENIX Association.

[4] Markus Bauer, Ilya Grishchenko, and Christian Rossow. Typro: Forward cfi for c-style indirect function calls using type propagation. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, page 346–360, New York, NY, USA, 2022. Association for Computing Machinery.

[5] Lucas Becker, Matthias Hollick, and Jiska Classen. Sok: on the effectiveness of control-flow integrity in practice. In *Proceedings of the 18th USENIX Conference on Offensive Technologies*, WOOT'24, USA, 2024. USENIX Association.

[6] Lorenzo Binosi, Gregorio Barzasi, Michele Carminati, Stefano Zanero, and Mario Polino. The Illusion of Randomness: An Empirical Analysis of Address Space Layout Randomization Implementations. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[7] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security, ASIACCS*, 2011.

[8] Erik Bosman. x86: Srop mitigation: implement signal counting. https://lkml.org/lkml/2014/5/15/858. (12-11-2025).

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

[10] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow bending: On the effectiveness of Control-Flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., August 2015. USENIX Association.

[11] Chromium. syscall_sets.cc - chromium github. https://github.com/chromium/chromium/blob/main/sandbox/linux/seccomp-bpf-helpers/syscall_sets.cc#L353. (12-11-2025).

[12] Exploit Database. Nginx 1.3.9 - 1.4.0 - chuncked encoding stack buffer overflow (metasploit). https://www.exploit-db.com/exploits/25775. (29-10-2025).

[13] V8 Developers. Control-flow integrity in v8. https://v8.dev/blog/control-flow-integrity. (29-10-2025).

[14] Docker Docs. Seccomp security profiles for docker. https://docs.docker.com/engine/security/seccomp/. (12-11-2025).

[15] Victor Duta, Fabian Freyer, Fabio Pagani, Marius Muench, and Cristiano Giuffrida. Let me unwind that for you: Exceptions to backward-edge protection. In *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[16] Mozilla Firefox. Sandboxfilter.cpp - firefox github. https://github.com/mozilla-firefox/firefox/blob/main/security/sandbox/linux/SandboxFilter.cpp#L1178. (12-11-2025).

[17] 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*, RAID '23, page 527–546, New York, NY, USA, 2023. Association for Computing Machinery.

[18] GNU. Gcc wiki - vtv. https://gcc.gnu.org/wiki/vtv. (12-11-2025).

[19] GNU. Program instrumentation options. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html. (12-11-2025).

[20] Yingjie Guo, Liwei Chen, and Gang Shi. Function-oriented programming: A new class of code reuse attack in c applications. In *2018 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, 2018.

[21] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589, 2014.

[22] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of Data-Oriented exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, Washington, D.C., August 2015. USENIX Association.

[23] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986, 2016.

[24] Intel. A technical look at intel® control-flow enforcement technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html. (29-10-2025).

[25] Seunghoon Jeong, Jaejoon Hwang, Hyukjin Kwon, and Dongkyoo Shin. A cfi countermeasure against got overwrite attacks. *IEEE Access*, 8:36267–36280, 2020.

[26] Linux Kernel. Control-flow enforcement technology (cet) shadow stack. https://docs.kernel.org/arch/x86/shstk.html. (12-11-2025).

[27] LLVM. Control flow integrity design documentation. https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html. (12-11-2025).

[28] Linux manual page. Sigaction(2) - linux manual page. https://man7.org/linux/man-pages/man2/sigaction.2.html. (05-10-2025).

[29] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 577–587, New York, NY, USA, 2014. Association for Computing Machinery.

[30] PaX. Address space randomization. https://pax.grsecurity.net/docs/aslr.txt, 2003. (12-11-2025).

[31] Aravind Prakash, Xunchao Hu, and Heng Yin. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 01 2015.

[32] Bootlin Elixir Cross Referencer. create_rstor_token (linux kernel source code). https://elixir.bootlin.com/linux/v6.15.2/source/arch/x86/kernel/shstk.c#L64. (29-10-2025).

[33] Bootlin Elixir Cross Referencer. get_shstk_data (linux kernel source code). https://elixir.bootlin.com/linux/v6.15.2/source/arch/x86/kernel/shstk.c#L272. (29-10-2025).

[34] Bootlin Elixir Cross Referencer. __libc_sigaction (glibc source code). https://elixir.bootlin.com/glibc/glibc-2.33/source/sysdeps/unix/sysv/linux/sigaction.c#L42. (29-10-2025).

[35] Bootlin Elixir Cross Referencer. pte_mkwrite_shstk (linux kernel source code). https://elixir.bootlin.com/linux/v6.15.2/source/arch/x86/include/asm/pgtable.h#L491. (29-10-2025).

[36] Bootlin Elixir Cross Referencer. rt_sigaction (linux kernel source code). https://elixir.bootlin.com/linux/v6.15.7/source/kernel/signal.c#L4644. (29-10-2025).

[37] Bootlin Elixir Cross Referencer. rt_sigreturn l266 (linux kernel source code). https://elixir.bootlin.com/linux/v6.15.2/source/arch/x86/kernel/signal_64.c#L266. (29-10-2025).

[38] Bootlin Elixir Cross Referencer. rt_sigreturn l275 (linux kernel source code). https://elixir.bootlin.com/linux/v6.15.2/source/arch/x86/kernel/signal_64.c#L275. (29-10-2025).

[39] Bootlin Elixir Cross Referencer. setup_signal_shadow_stack l364 (linux kernel source code). https://elixir.bootlin.com/linux/v6.15.2/source/arch/x86/kernel/shstk.c#L364. (29-10-2025).

[40] Bootlin Elixir Cross Referencer. setup_signal_shadow_stack l370 (linux kernel source code). https://elixir.bootlin.com/linux/v6.15.2/source/arch/x86/kernel/shstk.c#L370. (29-10-2025).

[41] Bootlin Elixir Cross Referencer. Sigaction (linux kernel source code). https://elixir.bootlin.com/linux/v6.15.2/source/arch/x86/include/uapi/asm/signal.h#L93. (05-10-2025).

[42] Bootlin Elixir Cross Referencer. Sigframe (linux kernel source code). https://elixir.bootlin.com/linux/v6.15.2/source/arch/x86/include/asm/sigframe.h#L59. (05-10-2025).

[43] Bootlin Elixir Cross Referencer. __sigset_t (glibc source code). https://elixir.bootlin.com/glibc/glibc-2.33/source/signal/bits/types/sigset_t.h#L7. (29-10-2025).

[44] Bootlin Elixir Cross Referencer. sigset_t (linux kernel source code). https://elixir.bootlin.com/linux/v6.15.7/source/arch/x86/include/asm/signal.h#L25. (29-10-2025).

[45] AliAkbar Sadeghi, Salman Niksefat, and Maryam Rostamipour. Pure-call oriented programming (pcop): chaining the gadgets using call instructions. *Journal of Computer Virology and Hacking Techniques*, 14:1–18, 05 2018.

[46] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Proceedings of the IEEE Symposium on Security and Privacy, SP*, 2015.

[47] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM conference on Computer and Communications Security, CCS*, 2007.

[48] sroettger. Ierae ctf 2024. https://gist.github.com/sroettger/fe66f7eb0cb10a8ebd1454875a7131ea. (29-10-2025).

[49] Ubuntu. Compilerflags - ubuntu wiki. https://wiki.ubuntu.com/ToolChain/CompilerFlags. (29-10-2025).

[50] Chao Zhang, Scott A. Carr, Tongxin Li, Yu Ding, Chenyu Song, Mathias Payer, and Dawn Song. Vtrust: Regaining trust on virtual calls. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[51] Tianning Zhang, Miao Cai, Diming Zhang, and Hao Huang. esrop attack: Leveraging signal handler to implement turing-complete attack under cfi defense. In Fengjun Li, Kaitai Liang, Zhiqiang Lin, and Sokratis K. Katsikas, editors, *Security and Privacy in Communication Networks*, pages 752–769, Cham, 2023. Springer Nature Switzerland.

```
struct sigaction {
        __sighandler_t sa_handler;
        unsigned long sa_flags;
        __sigrestore_t sa_restorer;
        sigset_t sa_mask;
};
```

Listing 3. Sigaction interface for system calls.

```
struct k_sigaction {
        __sighandler_t sa_handler;
        unsigned long sa_flags;
        __sigrestore_t sa_restorer;
        sigset_t sa_mask;
        __sigrestore_t ka_restorer; //unused
};
```

Listing 4. Internal sigaction struct in the kernel.

Appendix A. Sigaction Arguments For Syscalls

While `sigaction` in libc uses the sigaction structure shown in Listing 1, the kernel expects a different layout by the time the syscall is invoked. Internally, libc's `sigaction` reorders the elements of the structure before invoking the syscall [41], as we show in Listing 3.

This is relevant when using the `sigskip` gadget (see § 3.3.3), as the syscall will receive the reordered structure; attackers must anticipate this when preparing the payload.

Appendix B. OldAct Kernel Unhollowing

As described in § 3.1.1, the `sigaction` function in libc receives a third argument, `oldact`, which is a pointer to a `sigaction` structure where the kernel will write the previous signal handler information for the given signal. We also discussed in § A how the kernel expects the `sigaction` structure in a different layout than libc, and therefore libc reorders the elements before invoking the syscall.

Interestingly, there exists a *third* format for the `sigaction` structure, which is the one used internally by the kernel itself. We show such `k_sigaction` format in Listing 4.

If we compare the kernel `k_sigaction` with the `sigaction` struct passed to the syscall (see Listing 3), we may only notice one difference: the kernel's version contains an extra element at the end, `ka_restorer`. However, this is unnoticeable in practice, as this element is only actively used in SPARC architectures.

Instead, the critical difference lies in the size of the `sigset_t` structure used for the signal mask, which is 128 bytes long in libc (and in the syscall interface) [43] but only 8 bytes in the internal representation of the kernel [44]. While this size mismatch is not an issue normally, we find some cases where it can be abused by an attacker.

When the `oldact` argument in the `sigaction` is not null, the kernel writes the signal handler information

`act` into the `oldact` structure passed by the user during the syscall execution. However, since the kernel is the one performing the copy, it copies the four first elements of the `k_sigaction` structure (writes 32 bytes, omitting `ka_restorer`) into the user-provided `oldact` structure of size 144 bytes [36]. As a result, the remaining 112 bytes of the user-provided `oldact` structure are left untouched by the kernel.

By itself, this issue is barely noticeable, as it does not overflow the userspace buffer, instead just leaving the last 112 bytes unmodified. However, it opens the door for an attacker to place arbitrary data in the address where the `oldact` buffer will land before the execution of `sigaction`; then reusing that memory area in combination with other weaknesses, such as a user-kernel data collision (see § 3.3.7).

Appendix C. Double-Partial Sigframe Pivoting

To address the limitations of classic and unwinding sigframe pivoting, we design another sigframe pivoting technique that does not rely on the memory directly pointed by the initial `rbp` remaining unaltered until the restorer execution. Instead, this technique relies on the user-kernel data collision we identified in the `sigaction` function in libc, and on the oldact unhollowing weakness we described in § B. While more tedious to set up, it works in every scenario.

As we discussed earlier, when `sigaction` is used as a signal handler, `rdi` and `rsi` are positioned perfectly for registering a new arbitrary signal handler and restorer. As it turns out, `rdx` is also well-positioned: `sigaction` interprets `rdx` as the pointer to the `oldact` structure, but the kernel sets it to point to the ucontext in the sigframe.

The result of this user-kernel data collision at `rdx` between the ucontext area of the sigframe and the `oldact` argument is as follows, looking at Listing 2: (1) `rdx` is saved into `r8` at the beginning of `sigaction` (line 4); (2) during the syscall, the kernel writes into the stack buffer of size 144 bytes starting at `rbp-0xa0` the contents of the kernel-internal `k_sigaction` structure (which is 32 bytes long), leaving 112 bytes still controlled by the attacker (explained in § 3.3.9); (3) libc copies the auxiliary buffer, whose starting 32 bytes were overwritten by the kernel, and the remaining 112 bytes are attacker-controlled, into the `oldact` structure pointed by `r8` (line 20); (4) due to the collision, not only the `oldact` structure is written, but also the ucontext area in the sigframe.

The result is a 144-byte-long overwrite of the ucontext area in the sigframe, of which 112 are controlled by the attacker. We illustrate the bytes that can be controlled in Figure 9.

As it can be noticed, the attacker can never reach the `rsp` entry in the ucontext, which was the goal for the stack pivot, nor can it overwrite `rip`, which would be necessary for hijacking the execution flow during the subsequent

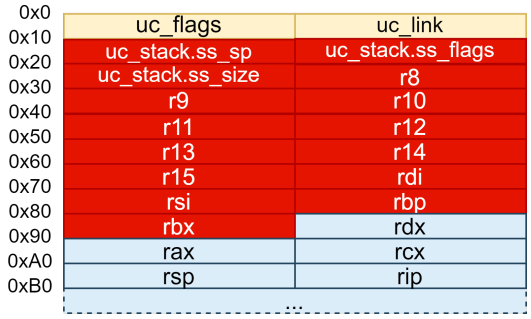

Figure 9. Overwritable ucontext (in red) in the first stage of a double-partial sigframe pivoting.

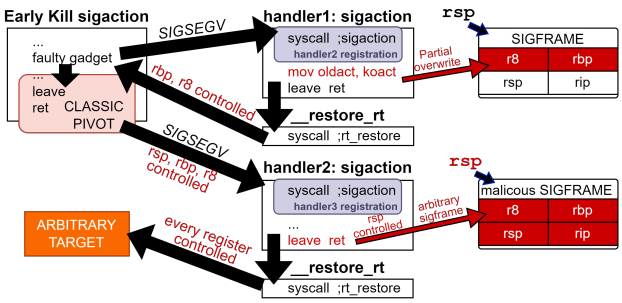

Figure 10. Double Partial pivot for hijacking an arbitrary sigframe and maintaining control over the next handler.

`__restore_rt`. It is at this point where the second part of the pivot comes into play. We show this process in Figure 10.

Firstly, the attacker leverages the partial overwrite to hijack the value of the `rbp` register. At the same time, on top of this partial overwrite, the attacker registers a new signal handler pointing now at a `leave; ret` gadget, just like during classic sigframe pivoting.

After `__restore_rt` processes the sigframe and restores the corrupted value of `rbp`, any prior function—another `sigaction`—will always execute its epilogue (if it was the Early Kill `sigaction`, we can also put a valid value into `r8` in the partial overwrite to avoid re-triggering the faulty gadget). During the execution of this epilogue `leave; ret`, the attacker maintains control of `rbp`, gains control of `rsp`, and is free to crash the program by trying to return to an invalid address. The result is the execution of the next signal handler, where now `rsp` can be pivoted like in a classic sigframe pivoting, allowing the attacker to fully control the sigframe used during the subsequent `sys_rt_sigreturn`.

An important observation is that, if instead of registering a `leave; ret` gadget as the new signal handler, the attacker registers the handler to start right before the syscall instruction (line 15, also known as the `sigskip` gadget), then the attacker can not only pivot the sigframe but also register a custom restorer at the same time.

Finally, we would like to note that the technique described here is not the only possible way to build a double-partial sigframe pivoting; other user-kernel data collisions

can be used to achieve a partial overwrite, and there even exist alternative setups in `sigaction` (e.g., by hijacking register `r8` in the sigframe, it can be moved to point to the `rsp` and `rip` in the sigframe); we show some of them in our PoCs.

Appendix D. SFOP Payload Positioning

Any code-reuse attack requires the attacker to position payload data somewhere in memory that can be used during the exploit (e.g., stack data during a ROP chain). SFOP is no exception, but this technique comes with certain particularities.

Firstly, as we depicted in Figure 1, every time a signal handler is executed, the kernel will displace `rsp` downwards in memory to make room for the sigframe, along with the xstate and the redzone. While we don't generally care about these two last elements for SFOP, it is important to acknowledge their existence as to predict where `rsp` will land after multiple signals have been triggered.

Even with ASLR, the attacker can accurately predict the position of `rsp` after one or multiple signals, only by knowing the initial value of `rsp` or some stack-based address before any signal is triggered (which can be leaked using standard techniques). This is analogous as how ROP attacks need to know the position of the stack in memory to place their payloads (e.g., injecting into memory the string *"/bin/sh"* and then needing its stack address as to pop it into `rdi` before returning to `system()`).

Given an initial `rsp` value, the attacker can easily compute the resulting signal handler `rsp` value considering the following elements: (1) first, the kernel always adds 0x80 bytes to `rsp` for the redzone, then proceeds to align the resulting address to a 16-byte boundary; (2) then, the kernel adds the size of the xstate. This area depends on the CPU features enabled (e.g., AVX); we verified it to be 0xA8C bytes long on modern Intel CPUs with AVX enabled and 0x100 bytes shorter on AMD CPUs; two systems with the same configuration will always have the same size for the xstate. Then, the attacker aligns the resulting address again to a 64-byte boundary; (3) finally, the kernel adds the size of the sigframe itself, including the siginfo, ucontext and restorer pointer; this area is 0x1B0 bytes long.

Not only is `rsp` important for predicting `rsp` during a signal handler; since it defines the start of the sigframe, other elements such as the siginfo pointer (`rsi`) and the ucontext pointer (`rdx`) are also defined relative to `rsp`. These two values are computed by the kernel during the signal triggering process based on the userspace value of `rsp`.

At first glance, it may seem inconvenient for an attacker to have its payloads positioned relative to such a widely changing `rsp`; during SFOP, multiple signal handlers are executed, each one (when they are nested, via `SA_NODEFER` or reregistration) displaces `rsp` further down the stack. This would mean that, for every nested signal triggered, the attacker would need to position the payload further down the stack as well, at a distance of `0xd00` bytes; this is a considerable distance and not every vulnerability would allow this.

Having said this, the SFOP attack stages (one stage is one nested signal handler) are *stack agnostic*, i.e., the attacker is free to choose the location of the payloads in memory during each stage, and this `0xd00` displacement is not a requirement. Three techniques guarantee stack agnosticism of bSFOP and fSFOP payloads: (1) Pivoting the sigframe allows *forward* control of `rsp`: right before any nested signal is triggered, by altering `rsp` (e.g., `rsp+=0xc00`), the attacker can compensate for the kernel displacement during the next signal handler (e.g., the kernel now applies `rsp-=0xd00`, but the resulting total offset has been `rsp-=0x100`). This also holds for unwinding-based pivoting; independently of how many nested signal handlers there are, the attacker pivots the final value of `rsp` before the next signal is triggered. (2) *Backward* control is guaranteed using `sys_rt_sigreturn`, as many such calls occur during SFOP with controlled sigframes; every time `sys_rt_sigreturn` is executed, the attacker can prevent the kernel from undoing the displacement (e.g., setting `rsp+0xd00`) by altering the `rsp` value in the sigframe itself. (3) For the Early Kill stage, where it is too early for the attacker to have pivoted `rsp`, the attacker can still control the starting position of `rsp` using the starting indirect branch gadget chain that reaches `sigaction`.

The previous techniques allow the attacker to position any payload anywhere in the stack during SFOP; as seen the `0xd00` displacement needs to be accounted for internal calculations during pivoting, but it does not limit the attacker in any way. Moreover, in practice, this automatic kernel displacement of `rsp` is not only not a problem, but actually an advantage, as it automatically pivots the stack for the attacker during each nested signal handler, allowing it to hit new user-kernel data collisions.

Only one property must be guaranteed with the position of the payloads: the attacker must be aware that, for every signal handler executed, the kernel *writes* into memory the whole contents of the sigframe and xstate, as shown in Figure 1. This means that, in general, it is advisable that the attacker positions payloads for each deeper stage in a lower memory address than the previous stage, to avoid overwriting any data. For instance, the payloads (e.g., each counterfeit sigframe) can be positioned at `rsp-0x200`, `rsp-0x400`, `rsp-0x600`, and so on, for each deeper nested signal handler. We are also aware of the possibility of using the redzone area for payloads, as it is by design a non-written area by the kernel.

Appendix E. Examples of CET-Compliant Indirect Branch Gadgets

For reference, we include some examples of CET-compliant indirect branch gadgets found in the libc library.

```
authunix_marshal:
    endbr64
    mov     rax,rdi
    mov     rdi,rsi
    mov     rsi,QWORD PTR [rax+0x40]
    mov     rax,QWORD PTR [rdi+0x8]
    mov     edx,DWORD PTR [rsi+0x1c8]
    add     rsi,0x38
    jmp     QWORD PTR [rax+0x18]
```

Listing 5. Example of indirect branch gadget in the libc library that is CET-compliant.

```
__pthread_once_slow+311:
    endbr64   ;<-- valid IBT gadget start
    mov rbx, rax
    jmp <pthread_once_slow.cold>
_pthread_once_slow.cold:
    cmp DWORD PTR [rbp-0x50], 0x0
    je <somewhere unuseful>
    mov rdi, QWORD PTR [rbp-0x58]
    call QWORD PTR [rbp-0x60]
```

Listing 6. Gadget in libc that sets `rdi` based on stack memory.

Appendix F. Number of Indirect Branch Gadgets in Common Libraries

Table 3. CET-COMPLIANT INDIRECT BRANCH GADGET COUNTS IN POPULAR LIBRARIES.

libc	ld-linux-x86-64	libstdc++	libcrypto	libX11	libz
208	10	354	207	1083	8

Appendix G. Predicting the Shadow Stack Page

As described in § 4.2, attackers can inject malicious SSP values into the CET shadow stack that will later be restored during the `sys_rt_sigreturn` syscall. However, in order to construct the bSFOP recursive shadow stack, attackers must be able to know the address of the shadow stack page in memory in the first place.

As it turns out, the loader is responsible for allocating the shadow stack page when CET is enabled for a process during the program startup. This means that the shadow stack page is allocated prior to any other library, including libc. As a result, the shadow stack page is always allocated at a lower address than any other library mapped in the process memory. Moreover, mapping offsets between libraries are typically consistent even if ASLR is enabled [6]; libraries are allocated sequentially, so we often find the shadow stack page at a fixed offset below the first mapped library in the process memory across different runs in a given system.

By leveraging this property, attackers can predict the shadow stack page address by leaking any address from the first library mapped in the process memory (i.e., usually, a libc address, as it is typically the first mapped library), then calculate the shadow stack page address by subtracting the known offset. This offset can be obtained by inspecting the memory mappings of the program in the attacker local machine (with the same configuration as the target), just like offsets inside libc are calculated during exploitation by inspecting the same libc version as in the attack target. We attach to our artifacts a script that automates this calculation.

Appendix H. Exploits using Dispatcher-less Chaining

The gadget chaining technique we present to register the signal handler used in SFOP can not only be used to call `sigaction` with controlled arguments, but also any other available single arbitrary function. Here we address whether this single arbitrary call can be used to build a complete exploit by itself.

The most straightforward way to build an exploit using only one arbitrary function call is to call `system` or `execve`, which execute arbitrary programs. While this might be sufficient in synthetic scenarios to spawn a reverse shell, there are major benefits of using a complete code-reuse attack such as SFOP instead.

Firstly, exactly because `system` and `execve` can be used to execute arbitrary programs, their usefulness is limited if no useful programs exist in the target system. For instance, in minimal environments, such as distroless containers designed to only run a single application or any other system without scripting programs, these functions are useless as there is no shell or other programs to execute.

Secondly, `system` and `execve` are arguably the most monitored functions in any hardened system, as their misuse is a common indicator of compromise. In fact, while (as shown in § 6) `sigaction` is not blocked by default seccomp configurations in popular software like Firefox and Chromium, `execve` is blocked in their default configuration (and `system` uses the `execve` syscall internally).

In addition, from a practical perspective, a Turing-complete code reuse technique that allows calling multiple arbitrary targets from within the same process context has great advantages. For instance, attackers can use SFOP to mark pages as executable (e.g., using `mprotect`, available in the seccomp policy of Firefox and Chromium) and then execute shellcode directly in memory; maximizing stealth and flexibility. Other times, some exploits could necessarily require operating from within the same process context, e.g., when the exploit goal is to leak some sensitive data from the process memory (e.g., reading in-memory secrets and then sending it over the network).

Finally, the ability of executing infinitely many functions elevates our technique to the same level as traditional code-reuse attacks while bypassing existing defenses; every technique described in § 2.4 allows the attacker to call arbitrary functions multiple times.