InversOS: Efficient Control-Flow Protection for AArch64 Applications with Privilege Inversion

Zhuojia Shen
University of Rochester
Rochester, NY, USA
zshen10@cs.rochester.edu

John Criswell
University of Rochester
Rochester, NY, USA
criswell@cs.rochester.edu

Abstract

With the increasing popularity of AArch64 processors in general-purpose computing, securing software running on AArch64 systems against control-flow hijacking attacks has become a critical part toward secure computation. Shadow stacks keep shadow copies of function return addresses and, when protected from illegal modifications and coupled with forward-edge control-flow integrity, form an effective and proven defense against such attacks. However, AArch64 lacks native support for write-protected shadow stacks, while software alternatives either incur prohibitive performance overhead or provide weak security guarantees.

We present *InversOS*, the first hardware-assisted write-protected shadow stacks for AArch64 user-space applications, utilizing commonly available features of AArch64 to achieve efficient intra-address space isolation (called *Privilege Inversion*) required to protect shadow stacks. Privilege Inversion adopts unconventional design choices that run protected applications in the kernel mode and mark operating system (OS) kernel memory as user-accessible; InversOS therefore uses a novel combination of OS kernel modifications, compiler transformations, and another AArch64 feature to ensure the safety of doing so and to support legacy applications. We show that InversOS is secure by design, effective against various control-flow hijacking attacks, and performant on selected benchmarks and applications (incurring overhead of 7.0% on LMBench, 7.1% on SPEC CPU 2017, and 3.0% on Nginx web server).

CCS Concepts: • **Security and privacy → Systems security**; **Software and application security**.

Keywords: hardware-assisted protected shadow stacks, intra-address space isolation, AArch64, control-flow integrity

1 Introduction

AArch64 (64-bit ARM) processors are becoming increasingly popular, not only in embedded and mobile platforms but also in personal computers [7] and high-performance servers and data centers [5, 52, 92, 102]. Given the popularity of AArch64 processors used in production and in our daily lives, securing software on such systems is critical. In particular, a large portion of AArch64 application code is written in memory-unsafe programming languages (e.g., C and C++) and is vulnerable to control-flow hijacking attacks [111, 126] that exploit memory safety errors. While basic code injection attacks are prevented by the wide deployment of W⊕X [105], advanced code-reuse attacks like return-oriented programming (ROP) [111, 116] and jump-oriented programming (JOP) [13] are still possible. These attacks hijack a program's control flow by corrupting code pointers (e.g., return addresses and function pointers) to point to reusable code of the attacker's choosing. Worse yet, recent research [28] has demonstrated automation of ROP attacks on AArch64, necessitating effective and practical defenses to be deployed.

Control-flow integrity (CFI) [1, 2], a seminal mitigation to control-flow hijacking attacks, restricts a program's control flow to follow its intended control-flow graph. While ineffective by itself [20, 29, 36, 50], CFI has been shown to be effective when coupled with a mechanism that protects the integrity of return addresses [19], such as write-protected shadow stacks [18, 25]. However, software approaches to protecting return address integrity either suffer from high performance overhead (e.g., software-based shadow stacks [25, 30, 46, 133, 153]) or only provide probabilistic guarantees (e.g., information hiding [18, 107, 119, 155]). Hardware-assisted shadow stack protection, such as Control-flow Enforcement Technology (CET) [117] on x86, offers the best security and performance but is not natively available on AArch64.

In this paper, we present *InversOS*, a system that provides AArch64 user-space applications with hardware-assisted write-protected shadow stacks. InversOS does so without requiring the most recent hardware security features on AArch64 or modifying hardware. Instead, InversOS uses two widely available AArch64 features [9], namely *unprivileged load/store instructions* and *Privileged Access Never*, in a novel way to create an efficient domain-based instruction-level intra-address space isolation technique which we call *Privilege Inversion*. With Privilege Inversion, InversOS runs protected applications in the same privilege mode as an operating system (OS) kernel, sets up incorruptible shadow stack memory accessible only by unprivileged load/store instructions, and ensures the safety of running privileged user-space code via a combination of OS kernel modifications and compiler transformations. To keep compatibility with legacy untransformed application binaries, InversOS

repurposes another AArch64 feature to support coexistence of legacy and protected applications securely and efficiently.

We built a prototype implementation of InversOS based on the Linux kernel v4.19.219 [78] and the LLVM/Clang compiler v13.0.1 [73]. We analyzed the security of InversOS and assessed the strength of its defense against different types of control-flow hijacking attacks. Our evaluation of InversOS on a real AArch64 system and a comprehensive set of benchmarks and applications (LMBench [90], SPEC CPU 2017 [121], and Nginx [124]) shows low performance overhead (7.0% on LMBench, 7.1% on SPEC CPU 2017, and 3.0% on Nginx), indicating that InversOS is practical for deployment. We will timely open-source InversOS.

To summarize, we make the following contributions:

- We present Privilege Inversion, the first domain-based intra-address space isolation technique for AArch64 user-space applications, using only widely available features on commodity hardware.
- We designed and implemented InversOS, an OS-kernel-compiler co-design that provides the first hardware-assisted protected shadow stacks on AArch64 utilizing Privilege Inversion and is compatible with existing binaries.
- We evaluated the security and performance of InversOS and showed that InversOS is both efficacious and efficient.

The rest of the paper is organized as follows. Section 2 provides background information. Section 3 defines our threat model. Sections 4 and 5 describe the design and implementation of InversOS, respectively. Section 6 analyzes the security of InversOS. Section 7 presents the performance evaluation of InversOS, Section 8 discusses related work, and Section 9 concludes and discusses future work.

2 Background

In this section, we provide background information on protected shadow stacks. We also briefly introduce features of AArch64 instruction set architecture (ISA) that are relevant to the design and implementation of InversOS.

2.1 Protected Shadow Stacks

Control-flow hijacking attacks like ROP [111, 116] corrupt saved return addresses on the stack. One way to mitigate such attacks is to use shadow stacks [18], which keep copies of return addresses in separate memory regions. When calling a function, a return address is pushed onto both the regular stack and the shadow stack; on return, the program loads the return address from the shadow stack and either compares it to the one on the regular stack to ensure its validity [25, 33, 37] or jumps to the value loaded from the shadow stack directly [2, 53, 119, 153, 155]. To enforce return address integrity, however, shadow stacks themselves require protection that disallows illegal modifications. Prior approaches to protecting shadow stack integrity rely on system calls [25, 46, 133], software fault isolation (SFI) [30, 153], information hiding [18, 107, 119, 155], or special hardware (e.g., segmentation [2], MPK [18, 53], CET [117]). No hardware-assisted shadow stack protection exists on AArch64.

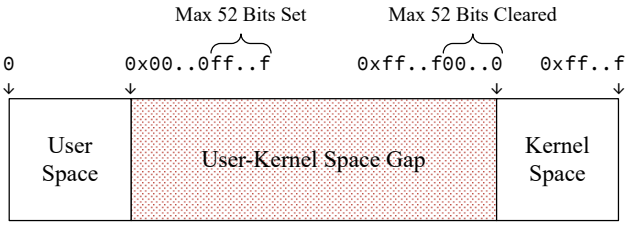

Figure 1. AArch64 Virtual Address Space

2.2 AArch64 Architecture

Exception Levels. AArch64 [9] provides four Exception Levels from EL0 to EL3, with increasing execution privileges. Typically user-space software executes in EL0 and OS kernels execute in EL1. EL2 and EL3 are for hypervisors and a secure monitor, respectively. A processor core enters from a lower Exception Level to a same or higher non-EL0 Exception Level via taking synchronous exceptions (e.g., traps, system calls) or asynchronous exceptions (e.g., interrupts) and returns via executing an `ERET` instruction. Each Exception Level ELx has a dedicated stack pointer register `SP_EL`x. Software running in ELx ($x \geq 1$) can select `SP_EL0` or `SP_EL`x as the current stack pointer, referred to as running in ELxt or ELxh (i.e., thread or handler mode). The two modes are different only in the stack pointer register in use, which also determines the set of exception vectors to use when an exception occurs that targets the same Exception Level. The Linux kernel, as of v4.19.219, executes in EL1h and leaves EL1t (and thus the corresponding set of exception vectors) unused [78]. Unless otherwise noted, hereafter we only focus on EL0 and EL1(t/h) and refer to them as unprivileged and privileged (thread/handler) modes, respectively.

Address Space and Page Tables. AArch64 [9] uses hierarchical page tables and a hardware memory management unit (MMU) to provide virtual memory, with two Translation Table Base Registers `TTBR0_EL1` and `TTBR1_EL1` holding the root page table addresses. `TTBR0_EL1` is for the lower half of the virtual address space (which typically corresponds to the user space), while `TTBR1_EL1` is for the upper half (which typically corresponds to the kernel space). Not all 64 bits of an virtual address are used in address translation; AArch64 supports a virtual address space up to 52 bits, thus leaving a gap between the two halves, as Figure 1 shows.

AArch64 [9] supports page-level access permissions, controlled by the UXN (Unprivileged eXecution Never) bit, the PXN (Privileged eXecution Never) bit, and two AP[2:1] (Access Permission) bits in last-level page table entries (PTEs). As the names imply, UXN and PXN, when set, disable unprivileged and privileged instruction access of the corresponding page, respectively. AP[1] disables unprivileged data access when cleared, and AP[2] disables write access when set.

In addition to the above PTE bits, AArch64 [9] also supports hierarchical access permission control via the UXNTable bit, the PXNTable bit, and two APTable[1:0] bits in top- and mid-level PTEs (PTEs that point to a next-level page table rather than a page). Unlike their last-level PTE counterparts, these bits can apply access restrictions to the whole corresponding address range on top of the permission of subsequent levels. When set, UXNTable and PXNTable disallow unprivileged and privileged instruction access, respectively. APTable[0] disallows unprivileged data access when set, and APTable[1] disallows write access when set. The Linux kernel, as of v4.19.219, always keeps these bits cleared and instead only controls access permissions at page level [78].

Unprivileged Load/Store Instructions. A special feature of AArch64 [9] (and many other ARM ISAs such as ARMv7-M [8]) is unprivileged load and store (LSU) instructions. These instructions, with mnemonics starting with LDTR or STTR on AArch64, check unprivileged memory access permissions even when executed in the privileged mode. This makes LSU instructions useful in accessing user-space memory inside the OS kernel (such as Linux's `get_user()` and `put_user()` functions [15]).

Architecture Extensions. AArch64 [9] has architecture extensions; the initial ISA is called ARMv8.0-A, and subsequent releases (e.g., ARMv8.1-A) are based on the previous ISA with new hardware features. Specifically, we focus on the following hardware features: Privileged Access Never (PAN), User Access Override (UAO), Hierarchical Permission Disable (HPDS), and E0PD.

PAN [9] is an ARMv8.1-A feature which prevents privileged code from accessing unprivileged-accessible data memory, similar to x86's Supervisor Mode Access Prevention (SMAP) [3, 61]. When PAN is enabled via setting the PAN bit in the processor state PSTATE, all loads and stores (except LSU instructions) executed in the privileged mode that try to access memory accessible in the unprivileged mode will generate a permission fault.

UAO [9] is an ARMv8.2-A feature which, when enabled via setting the PSTATE.UAO bit, allows LSU instructions executed in the privileged mode to act as regular loads/stores.

HPDS [9], introduced in ARMv8.1-A, allows disabling hierarchical access permission bits (UXNTable, PXNTable, and APTable[1:0]) during page table lookups. Software running in the privileged mode can set the HPD{0,1} bits in Translation Control Register TCR_EL1 to disable hierarchical access permission checks in address translation from TTBR{0,1}_EL1. However, as AArch64 allows caching TCR_EL1.HPD{0,1} in translation lookaside buffers (TLBs), flipping either bit may require a local TLB flush to take effect.

E0PD [9], introduced in ARMv8.5-A as a hardware mitigation to side-channel attacks that leverage fault timing (e.g., Meltdown [79]), prevents code running in the unprivileged mode from accessing (lower or upper or both) halves of the virtual address space and generates faults in constant time. Similar to HPDS, there are two bits TCR_EL1.E0PD{0,1} that privileged software can use to control whether unprivileged access to which half of the address space is disabled.

3 Threat Model

We assume a powerful remote attacker trying to achieve arbitrary code execution on a benign but potentially buggy application by exploiting arbitrary memory read/write vulnerabilities to hijack the control flow. We assume that the underlying OS kernel and hardware are trusted and unexploitable, providing user space with the basic W⊕X protection [105]. Non-control data attacks [22] (such as data-oriented programming [59] and block-oriented programming [63]), side-channel attacks, and physical attacks are out of scope. This threat model is in line with recent work on user-space control-flow hijacking attacks [28, 29] and defenses [18, 74, 75, 136].

4 Design

In this section, we present the design of InversOS. The goal of InversOS is to provide low-cost return address integrity to user-space applications running on commodity AArch64 systems, which may or may not come with the most recent hardware security features such as Pointer Authentication (PAuth), Branch Target Identification (BTI), and Memory Tagging Extension (MTE) [9]. To do so, InversOS must only rely on AArch64 features from the early ISA versions. We therefore require InversOS's target platform to support at least PAN and HPDS (i.e., conforming to ARMv8.1-A [9]); this allows InversOS to be deployed on most of AArch64 systems released since 2017 [139].

Overall, we devise InversOS as a co-design between an OS kernel and a compiler. The InversOS-compliant OS kernel utilizes *Privilege Inversion*, a novel intra-address space isolation technique we invented, to provide user-space applications an extra protection domain accessible only by LSU instructions. The InversOS-compliant compiler then instruments user-space code to leverage the protection domain for efficient protected shadow stacks as well as to enforce forward-edge CFI [1, 2], allowing InversOS to protect user-space applications without modifying their source code. The nature of Privilege Inversion dictates running user-space applications in the privileged mode; we therefore combine

CFI, a compile-time bit-masking compiler pass, a load-time code scanner in the OS kernel, and a set of kernel modifications to together ensure the safety and security of doing so. Lastly, InversOS supports running legacy untransformed applications to keep compatibility with existing binaries via a novel use of HPDS or E0PD (if available).

4.1 Privilege Inversion

LSU instructions in AArch64, as described in Section 2.2, show a great potential in implementing efficient intra-address space isolation; previous work [26] has explored their usage in kernel-level data isolation. However, using these instructions to compartmentalize user-space applications poses challenges as they act like regular loads/stores when executed in the unprivileged mode. Essentially the underlying hardware only supports one protection domain for unprivileged software.

We devise Privilege Inversion, a novel intra-address space isolation technique that creates a separate protection domain for AArch64 user-space applications. With Privilege Inversion, the OS kernel runs a user-space application needing an extra protection domain in the privileged mode. We dub such an application as an *elevated task*. When launching an elevated task, the OS kernel configures its memory pages as unprivileged-inaccessible (i.e., with `AP[1]` cleared in PTEs), marks its code pages as privileged-executable (i.e., with `PXN` cleared and `UXN` set in PTEs), and enables PAN during its execution. Then, pages that the elevated task wants to place in the separate protection domain are marked as unprivileged-accessible (i.e., with `AP[1]` set in PTEs). Note that the elevated task's pages are still mapped to the user space (translated by `TTBR0_EL1`); the above changes only apply to their access permission bits in the PTEs. This configuration allows LSU instructions in elevated task code to access the protected pages but forbids accesses to them made by all regular loads/stores due to PAN. In the meanwhile, it leaves all other unprotected pages in the elevated task accessible by regular loads/stores but inaccessible by LSU instructions, effectively compartmentalizing the elevated task into two separate protection domains (one for regular loads/stores and the other for LSU instructions), as Figure 2 shows. Note that in systems with UAO support, UAO has to be turned off during elevated task execution; otherwise LSU instructions would act just like regular loads/stores.

However, in order to make Privilege Inversion safe and useful, we need to address the following challenges:

Challenge 1. *As elevated tasks run in the privileged mode, kernel memory becomes accessible by their regular loads/stores.*

Challenge 2. *As elevated tasks run in the privileged mode, their control-flow transfer instructions can jump to the kernel space to execute arbitrary kernel code (i.e., kernel memory with* `PXN` *cleared).*

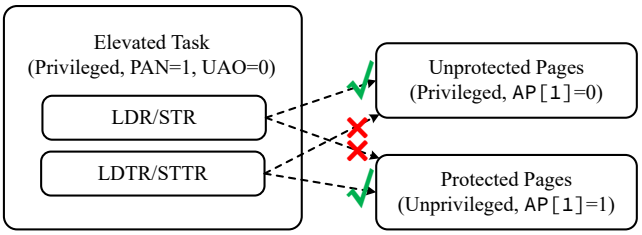

Figure 2. Compartmentalization by Privilege Inversion

Challenge 3. *As elevated tasks run in the privileged mode, they may contain and execute special privileged instructions that would only be allowed to execute in kernel code (e.g., instructions that flip* `PSTATE.PAN`*).*

To address Challenge 1, we incorporate a set of kernel modifications that mark all kernel memory as unprivileged-accessible and disable PAN during kernel execution. Such modifications, while radical in idea, effectively stop regular loads/stores in elevated tasks from accessing kernel memory and still keep the OS kernel functional. The ramifications of modifying the OS kernel in this way are two folds. First, LSU instructions in elevated tasks can now access kernel memory. We therefore require that elevated tasks not contain LSU instructions by themselves (which is the case in C/C++ code compiled by GCC or LLVM/Clang) and use a compiler pass to insert vetted LSU instructions for enforcing the desired protection policies. Our shadow stack pass described in Section 4.2 provides a good example. Second, if we are to support running legacy untransformed applications in the unprivileged mode still, they can access kernel memory as well; Section 4.3 discusses how we tackle this problem.

To address Challenge 2, we use a bit-masking compiler pass, which instruments all indirect control-flow transfer instructions (i.e., indirect calls, indirect jumps, and returns) in elevated tasks by preceding them with a bit-masking instruction that clears the top bit of the target register.[1] This limits the control-flow transfer target to be within the user space or to become an invalid pointer pointing to the user-kernel space gap. Such instrumentation alone, however, can be bypassed by attacker-manipulated control flow that jumps over the bit-masking instruction; we therefore combine it with CFI to ensure its execution, which we discuss in Section 4.2. Note that direct control-flow transfer instructions (i.e., direct calls and jumps) do not need such instrumentation; their target is PC-relative and always points to a known location within the user space.

To address Challenge 3, we add to the OS kernel a load-time code scanner which scans for privileged instructions that unprivileged software should never execute. Whenever a page in an elevated task is being marked as executable, the OS kernel invokes our code scanner to scan the whole page;

[1] AArch64 returns via the `RET` instruction, which uses the link register `LR` (by default) or another explicitly specified register as the return address [9].

if the page contains any forbidden privileged instruction, the execution permission of the whole page is denied. As AArch64 instructions are 4-byte sized and aligned [9], a linear non-overlapping scan should suffice.

4.2 Protected Shadow Stacks and Forward-Edge CFI

With Privilege Inversion creating an extra protection domain, we can now leverage the protection domain to enforce efficient shadow stack protection for the user space. Specifically, the OS kernel allocates unprivileged memory for a shadow stack when a new elevated task is launched via `exec()` or when a new thread in an elevated task is created via `clone()`. The compiler utilizes a shadow stack pass to instrument elevated task code; a copy of the return address is saved onto a shadow stack via an `STTR` instruction inserted into the prologue of functions that save the return address to the regular stack, and the return address is loaded from the shadow stack via an `LDTR` instruction inserted into the epilogue(s) of these functions. A special case for shadow stacks to handle is irregular control flow such as `setjmp()`/`longjmp()` in C and exception handling in C++. Since support for such irregular control flow depends on the specific shadow stack scheme used [18], we discuss how our InversOS prototype supports such code constructs in Section 5.2.

To form a complete control-flow protection, we couple our shadow stacks with forward-edge CFI [1, 2], which ensures that the target of indirect calls and jumps is within a set of allowed code locations. Specifically, we use a label-based CFI pass in the compiler. For each indirect call or tail-call indirect jump in elevated task code, the pass inserts a CFI label at the beginning of every function that might be the call target and inserts a CFI check before the call. Similarly, for each non-tail-call indirect jump in elevated task code, the pass inserts a CFI label at the beginning of every successor basic block and inserts a CFI check before the jump. The CFI check ensures that a proper CFI label is present at the control-flow target; otherwise it generates a fault and traps the execution.

4.3 Compatibility

Not all AArch64 user-space applications need a separate protection domain, nor can all of them be recompiled. InversOS must therefore allow existing application and library binaries that are not compiled by the InversOS-compliant compiler to run without compromising its security.

We propose two methods to allow safe execution of legacy applications in the unprivileged mode (dubbed as *legacy tasks*), depending on hardware feature availability. In systems with E0PD support (ARMv8.5-A and onward), the OS kernel can directly enable E0PD via setting `TCR_EL1.E0PD1` during legacy task execution. This way, even though kernel memory is marked unprivileged-accessible, legacy tasks running in the unprivileged mode still cannot access kernel memory translated by `TTBR1_EL1`.

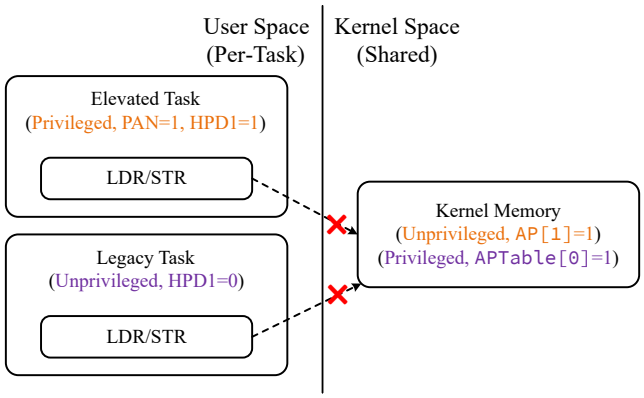

Figure 3. Different "Views" of Kernel Memory Due to HPDS

In pre-ARMv8.5-A systems without E0PD support, however, we rely on HPDS to provide a less-efficient solution. Specifically, the OS kernel first sets `APTable[0]` in all top- and mid-level PTEs of kernel memory when establishing page tables for the kernel space. This effectively marks all kernel pages as unprivileged-inaccessible even if `AP[1]` in their last-level PTEs is set. Then, the OS kernel enables HPDS via setting `TCR_EL1.HPD1` before running an elevated task, disables HPDS via clearing `TCR_EL1.HPD1` before running a legacy task, and flushes the local TLBs everytime after flipping `TCR_EL1.HPD1`. This way, legacy and elevated tasks will possess different "views" of kernel memory, as Figure 3 depicts. Specifically, legacy tasks see kernel memory as unprivileged-inaccessible due to `APTable[0]` being set, while elevated tasks see kernel memory as unprivileged-accessible because HPDS disables `APTable[0]` in top- and mid-level PTEs and `AP[1]` in last-level PTEs takes effect. As a result, both types of tasks cannot access kernel memory.

Note that relying on HPDS prevents the OS kernel from mapping kernel memory with the largest huge pages on certain systems (e.g., 1 GB huge pages with a page size of 4 KB and a 39-bit virtual address space), because such pages have no top- or mid-level PTEs for setting `APTable[0]`. However, we believe this has no practical impact on the OS kernel's address translation and memory usage; the use of the largest huge pages is rare and infrequent.

5 Implementation

We implemented a prototype of InversOS on the Linux kernel v4.19.219 [78] and the LLVM/Clang compiler v13.0.1 [73]. Using Tokei v12.1.2 [142], our kernel modifications include 1,815 lines of C code and 207 lines of assembly code, and our changes to LLVM contain 1,003 lines of C++ code. To provide complete and transparent InversOS support for user-space applications, we also modified the musl libc v1.2.2 [45] and LLVM's LLD linker [87], compiler-rt builtin runtime library [85], and libunwind [84], totalling 27 lines of C code, 131 lines of C++ code, and 299 lines of assembly code.

Table 1. Forbidden Privileged Instructions by Code Scanner

Instruction	Description
MRS*/MSR*	Read/Write System Register
IC*/DC*	Invalidate Instruction/Data Cache
TLBI	Invalidate Translation Lookaside Buffer
HVC	Hypervisor Call
SMC	Secure Monitor Call
AT	Address Translation
ERET	Exception Return
CFP/CPP/DVP	Prediction Restriction
LDGM/STGM/STZGM	Load/Store Tag Multiple (MTE)
BRB	Branch Record Buffer
SYS/SYSL	Other System Instructions

* Instructions with Certain Operands Allowed

5.1 OS Kernel Modifications

Privilege Inversion requires running elevated tasks in the privileged mode. As Linux does not use the privileged thread mode (as Section 2.2 describes), our prototype therefore utilizes it to run elevated tasks. This way, the Linux kernel can keep using the privileged handler mode for its own operations without interference from elevated tasks. It also greatly simplifies our implementation. To enable the privileged thread mode, our prototype enables an unused set of exception vectors that are responsible for taking exceptions from the privileged thread mode to the privileged handler mode. Changes were also made to Linux's existing AArch64 exception handler code so that our prototype can reuse most of the code to handle exceptions from the privileged thread mode and to resume elevated task execution properly. Note that elevated tasks in our prototype still use the `SVC` instruction for system calls, which is unnecessary because elevated tasks are already privileged; we leave system call optimizations as future work.

When launching a new task, InversOS must decide whether it should be run as a legacy or elevated task. For simplicity and ease of implementation, our prototype checks the presence of an environment variable `INVERSOS=1` to make such a decision; if it is present, the task is started as an elevated task. Production systems can use a more enhanced mechanism (e.g., checking the presence of a code signature generated by an InversOS-compliant compiler) to qualify an elevated task.

The load-time code scanner, as part of our kernel modifications, scans for illegal privileged instructions in elevated task code. Instead of directly scanning a user-space code page, our prototype maps the page to the kernel space for scanning in order to avoid frequently calling `get_user()`. Table 1 lists all types of privileged instructions that our prototype forbids, which roughly correspond to instructions that would generate a fault when executed in the unprivileged mode but might not when executed in the privileged mode [9]. In particular, `MRS`/`MSR`/`IC`/`DC` instructions with certain operands (e.g., reading the unprivileged thread ID register `TPIDR_EL0` via `MRS`) are allowed in unprivileged software, so these instructions are also permitted in elevated tasks.

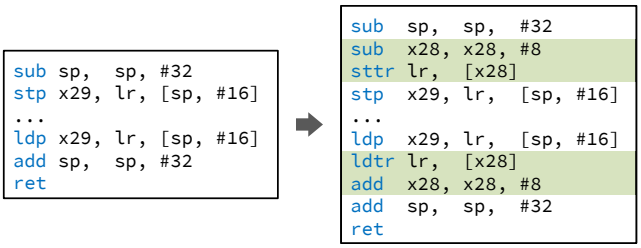

Figure 4. Shadow Stack Transformations

Our kernel modifications take responsibility of setting up and tearing down memory for protected shadow stacks in elevated tasks, as Section 4.2 describes. Each shadow stack region in an elevated task can grow as much as a regular stack can grow, supporting both parallel and compact shadow stack schemes [18]. To prevent shadow stack overflow and underflow, each shadow stack region is surrounded by two guard regions inaccessible by both regular loads/stores and LSU instructions. Mappings of shadow stack and guard regions are unmodifiable by `munmap()`, `mremap()`, and `mprotect()` requests from the user space.

Lastly, our prototype implements the HPDS support for running legacy tasks, as described in Section 4.3. We omitted implementing the E0PD alternative due to the lack of hardware that supports E0PD. As Linux has introduced support for E0PD since v5.6 [77] (which is enabled by default), a simple backport of the relevant changes would suffice.

5.2 Compiler, Linker, and Library Modifications

We implemented the shadow stack, forward-edge CFI, and bit-masking compiler passes in a single LLVM pass that transforms LLVM machine intermediate representation (IR).

Our shadow stack transformations adopt the compact shadow stack scheme [18] and reserve the `X28` register (a callee-saved register) as the shadow stack pointer register. Figure 4 demonstrates our shadow stack transformations performed on a function's prologue and epilogue. Our prototype supports C's `setjmp()`/`longjmp()` functions and C++ exception handling via modifications to the musl libc and LLVM's libunwind, respectively. Instead of directly guaranteeing the integrity of return address saved by `setjmp()` or `__unw_getcontext()`, our prototype provides shadow stack pointer integrity when restoring the saved context in `longjmp()` or `__libunwind_Registers_arm64_jumpto()`. Specifically, rather than overriding `X28` with the saved value, we unwind `X28` step by step until a matched return address is found or it reaches a guard region to cause shadow stack underflow.

Our forward-edge CFI transformations use the `BTI` instructions as CFI labels to keep forward compatibility with ARMv8.5-A's BTI [9], a hardware-assisted forward-edge CFI

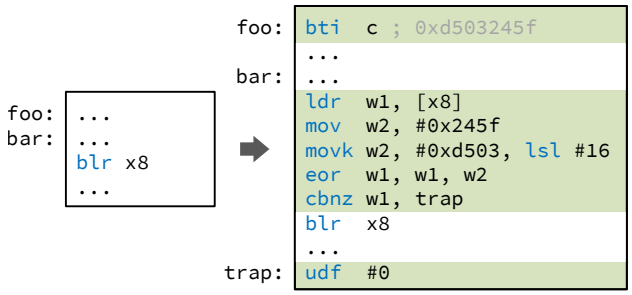

Figure 5. Forward-Edge CFI Transformations

mechanism rolling out to new AArch64 processors. Processors not supporting BTI execute a `BTI` instruction as a no-operation. An appropriate CFI check is inserted before every indirect call or jump to ensure that the target contains a correct CFI label (`BTI C` for indirect calls and tail-call indirect jumps and `BTI J` for non-tail-call indirect jumps). Figure 5 illustrates our forward-edge CFI transformations performed on an indirect call and one of its target functions. On AArch64, a non-tail-call indirect jump can only be generated from a `switch` or computed `goto` statement; the former is bounds-checked against a read-only jump table, and our prototype restricts the latter by transforming it to a `switch` statement using the `IndirectBrExpandPass` [86]. Consequently, a non-tail-call indirect jump is limited to jump within its function and cannot branch to other functions.

Our bit-masking transformation inserts an `AND` instruction before every indirect call, indirect jump, or return to clear the top bit of control-flow transfer target. For indirect calls and jumps, the instruction is placed after the CFI check.

While our all-in-one LLVM machine IR pass transforms most of elevated task code, it fails to cover certain pieces of code in the user space when compiling the application. One piece of untransformed code is the procedure linkage table (PLT) generated by the linker. We therefore also modified LLD to be able to generate CFI-checked and bit-masked PLT code. Another piece of untransformed code is Linux's virtual dynamic shared object (vDSO); it is compiled with the Linux kernel and stored within the kernel's read-only data. We therefore applied our compiler transformations to the vDSO as well during kernel compilation. The last case is assembly code (including assembly files and inline assembly statements). We manually instrumented assembly code in the musl libc and compiler-rt builtin runtime library.

5.3 Discussion

Virtualization Host Support. ARMv8.1-A adds Virtualization Host Extensions (VHE) [9] to accelerate hosted (Type 2) hypervisors such as Linux's KVM [32] and FreeBSD's bhyve [42]. In pre-VHE systems, a host OS kernel (running in EL1) needs to partition its hypervisor into a "high-visor" (running in EL1) and a "low-visor" (running in EL2) and thus incurs heavy overhead when context-switching between the two parts. VHE allows the host OS kernel to run entirely in EL2 to reduce the cost. The Linux kernel, as of v4.19.219 [78], stays in EL2h for execution when having detected VHE support during early boot. Our prototype therefore transparently supports running elevated tasks in EL2t in such a case.

AArch32 Support. Quite a few AArch64 processors still allow running AArch32 (32-bit ARM) applications for compatibility. While there are no technical difficulties to support an elevated task running in the AArch32 state (i.e., LSU instructions and PAN are also available on AArch32), we opted not to implement AArch32 support for the sake of time.

6 Security Analysis

In this section, we analyze the security of InversOS by providing answers to the following security questions:

SQ1 Why is InversOS secure (to run *instrumented* elevated tasks in the privileged mode and *arbitrary* legacy tasks in the unprivileged mode)?

SQ2 How well does InversOS mitigate control-flow hijacking attacks on elevated tasks?

6.1 Security by Design

To answer **SQ1**, we examine *all* potential ways to compromise InversOS from a legacy or elevated task:

1. A task may try to read from/write to memory of other tasks to break their confidentiality/integrity.
2. A task may try to read from/write to kernel memory to break the confidentiality/integrity of the OS kernel.
3. A task may try to allocate an excessive amount of resources (e.g., time, memory) to break the availability of InversOS.
4. A task may try to execute detrimental instructions that could undermine the security of InversOS.
5. A task may try to jump to kernel code and use kernel code as a "confused deputy" for the above goals.

As each task's memory (sans shared memory) is mapped exclusively to the task's own address space, reading and writing other tasks' memory can only be carried out by accessing kernel memory or jumping to kernel code. Since kernel memory has `AP[1]` (and `APTable[0]`, if using HPDS) set, accessing kernel memory is disabled via PAN for elevated tasks and via HPDS or E0PD for legacy tasks. Jumping to kernel code is also impossible; having `UXN` set for kernel code prevents legacy tasks from executing kernel code, while InversOS's CFI and bit-masking instrumentation ensures that control-flow transfers in elevated tasks never reach the kernel space. As for attacks on availability, we argue that InversOS does not introduce new availability problems; running an elevated task in the privileged mode does not prioritize it on resource allocation over all other legacy or elevated tasks and the OS kernel. The remaining case is privileged instructions, the

execution of which is restricted by hardware automatically for legacy tasks and by InversOS's load-time code scanner for elevated tasks. Conclusively, InversOS does not introduce new security flaws and is secure by design.

6.2 Efficacy against Control-Flow Hijacking

To answer **SQ2**, we first define and explain a list of invariants that InversOS maintains for guaranteeing return address integrity of elevated tasks and then reason about why return address integrity significantly reduces the control-flow hijacking attack surface. Specifically, InversOS maintains the following invariants for elevated tasks:

Invariant 1. *A function in an elevated task either pushes its return address in* `LR` *to a shadow stack, or never spills the return address to memory.*

Invariant 2. *If a function in an elevated task pushed its return address to a shadow stack, its epilogue will always load the return address from the shadow stack location in which its prologue saved the return address.*

Invariant 3. *An elevated task cannot corrupt shadow stacks by itself or by using a system call as a "confused deputy" (e.g., calling* `read(fd, buf, size)` *where* `buf` *points to shadow stack memory [138]).*

Invariant 1 is easily upheld by our shadow stack pass, which instruments `LR`-saving function prologues to push `LR` to the shadow stack. With the counterpart instrumentation on epilogue(s) of these functions to pop `LR` from the shadow stack, our shadow stack pass guarantees that only a function's prologue and epilogue(s) can update the shadow stack pointer with a matched decrement/increment, contributing to Invariant 2. Since our forward-edge CFI pass ensures that all indirect calls and tail-call indirect jumps target the beginning of a function and all non-tail-call indirect jumps are restricted within their containing function, shadow stack pointer decrements and increments are guaranteed to occur in a matched order, sustaining Invariant 2. Finally, Invariant 3 is maintained because the shadow stacks are unprivileged and no LSU instructions can be exploited to corrupt the shadow stacks, and because of the benign nature of elevated tasks assumed by our threat model in Section 3.

With return address integrity, control-flow hijacking attacks that require corrupting return addresses (such as return-into-libc [126] and ROP [111, 116]) are effectively prevented. Furthermore, as non-tail-call indirect jumps cannot break the "jail" of their containing function, attacks that exploit indirect jumps (such as JOP [13]) no longer work. The remaining attack surface requires attackers to do purely *call-oriented programming* (i.e., using only corrupted function pointers); while such attacks are possible [44, 114], they are limited by forward-edge CFI and can be further restrained if InversOS refines CFI's granularity. In short, InversOS greatly reduces the control-flow hijacking attack surface for elevated tasks.

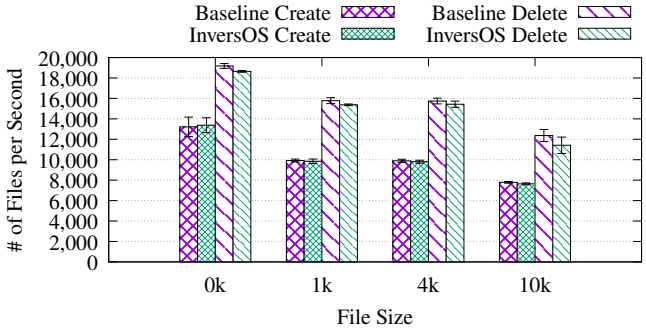

Figure 6. LMBench File Operation Rate (Higher is Better)

7 Performance Evaluation

We evaluated the performance of InversOS on a Station P2 mini-PC which has an RK3568 quad-core Cortex-A55 processor implementing the ARMv8.2-A architecture that can run up to 2.0 GHz. The mini-PC comes with 8 GB of LPDDR4 DRAM up to 1,600 MHz, 64 GB of internal eMMC storage (unused), and 1 TB of SATA SSD. It runs Ubuntu 20.04 LTS modified by the manufacturer.

We ran all our experiments using two configurations: Baseline and InversOS. In Baseline, we compiled program and library code using LLVM/Clang v13.0.1 [73] without the InversOS compiler transformations and ran the generated binary executables on a Linux v4.19.219 kernel [78] without our kernel modifications. In InversOS, all program and library code was compiled with the InversOS compiler transformations (i.e., shadow stack, forward-edge CFI, and bit-masking transformations) and executed on the same version of the Linux kernel modified with our kernel changes. When running an InversOS executable, we set an environment variable `INVERSOS=1` to inform the OS kernel that the program should be started as an elevated task, as Section 5.1 describes. As the processor lacks E0PD support, we rely on HPDS to prevent legacy tasks from accessing kernel memory. Both configurations used `-O2` optimizations and performed static linking against the musl libc v1.2.2 [45] and LLVM's compiler-rt builtin runtime library v13.0.1 [85]. C++ code in our experiments was compiled with and statically linked against libc++ [82], libc++abi [83], and libunwind [84] from LLVM v13.0.1. Libraries for Baseline and InversOS are compiled without and with our modifications described in Section 5.2, respectively.

7.1 Microbenchmarks

To understand the performance impact of the InversOS Linux kernel modifications, we used LMBench v3.0-alpha9 [90], a microbenchmark suite that measures the latency and bandwidth of various OS services. For each microbenchmark that supports parallelism, we ran four parallel workloads to reduce variance. We report an average and a standard deviation of 10 rounds of execution for each microbenchmark.

Table 2. LMBench Latency (Lower is Better)

Microbenchmark	Baseline (μs)	stdev (μs)	InversOS (×)	stdev (×)
null syscall	0.148	0.000	1.047	0.007
read	0.482	0.001	1.054	0.004
write	0.351	0.002	0.991	0.003
stat	4.928	0.023	1.066	0.003
fstat	0.422	0.003	1.052	0.005
open/close	9.744	0.017	0.989	0.003
select 500 fd	24.365	0.017	1.002	0.001
signal install	0.375	0.001	1.059	0.003
signal catch	3.801	0.009	1.493	0.002
protection fault	0.408	0.005	0.980	0.029
pipe	16.115	0.067	0.948	0.004
AF_UNIX stream	27.314	0.618	1.051	0.008
AF_UNIX connect	99.329	0.733	1.012	0.009
fork+exit	266.767	6.945	1.256	0.012
fork+exec	562.585	7.046	1.188	0.009
fork+shell	2,878.983	12.869	4.007	0.015
page fault	0.910	0.016	1.038	0.009
mmap 1 MB	42.700	3.318	1.019	0.007
udp	76.490	0.214	1.018	0.005
tcp	63.472	0.200	1.011	0.002
connect	102.196	0.503	1.004	0.006
context switch	59.318	0.880	0.993	0.014
fcntl	8.772	1.643	0.992	0.219
semaphore	3.083	0.515	0.954	0.162
usleep	78.661	1.579	0.995	0.020
Geomean	—	—	1.103	—

Table 3. LMBench Bandwidth (Higher is Better)

Microbenchmark	Baseline (MB/s)	stdev (MB/s)	InversOS (×)	stdev (×)
pipe	1,096.147	72.703	0.991	0.049
AF_UNIX stream	931.933	6.753	1.003	0.011
read 1 MB	3,706.665	65.823	0.978	0.013
read 1 MB open2close	3,474.633	45.699	0.990	0.015
mmap 1 MB	10,689.636	36.243	1.006	0.001
mmap 1 MB open2close	6,365.563	43.215	0.972	0.008
tcp	720.056	48.645	0.987	0.013
Geomean	—	—	0.989	—

Tables 2 and 3 and Figure 6 show LMBench performance of both Baseline and InversOS. Overall, InversOS incurred a geometric mean of 7.0% overhead: 10.3% on latency, 1.1% on bandwidth, and 2.2% on file operation rate. In most microbenchmarks the overhead is miniscule. Most notably, `fork+shell` exhibited a 4× slowdown because InversOS had to scan every code page of a newly executed shell. The same goes with `fork+exec`, in which the executed program is much smaller than the shell and thus incurred much less overhead (18.8%). In `fork+exit`, the 25.6% overhead comes from copying code page PTEs upfront; Linux by default only sets up shared page table mappings of a child process at page faults (i.e., when the child first accesses the page), which, however, would cause redundant code scanning in InversOS as InversOS invokes the code scanner whenever a page in an elevated task is marked executable. We therefore optimized InversOS to avoid redundant code scanning by copying an elevated task's code page PTEs during `fork()`. InversOS incurred 49.3% overhead in signal catching because of additional flipping of `PSTATE.UAO` (due to PAN being disabled) when setting up and tearing down a signal frame; this could be optimized away by simply disabling UAO support in the Linux kernel, which we opted not to in order to avoid introducing less relevant changes.

Table 4. SPEC CPU 2017 Execution Time (Lower is Better)

Benchmark (Rate)	Baseline (s)	Benchmark (Speed)	Baseline (s)
500.perlbench_r	135.795	600.perlbench_s	135.289
502.gcc_r	268.035	602.gcc_s	268.294
505.mcf_r	431.810	605.mcf_s	428.423
520.omnetpp_r	354.081	620.omnetpp_s	353.981
523.xalancbmk_r	242.465	623.xalancbmk_s	242.501
525.x264_r	96.540	625.x264_s	96.527
531.deepsjeng_r	203.713	631.deepsjeng_s	227.060
541.leela_r	216.941	641.leela_s	217.306
557.xz_r	128.610	657.xz_s	127.926
508.namd_r	157.894		
510.parest_r	330.373		
511.povray_r	25.722		
519.lbm_r	231.428	619.lbm_s	1,718.814
526.blender_r	533.649		
538.imagick_r	167.810	638.imagick_s	168.136
544.nab_r	396.789	644.nab_s	397.586

7.2 Macrobenchmarks and Applications

To see how InversOS performs on real workloads, we used SPEC CPU 2017 v1.1.9 [121] and Nginx v1.23.3 [124]. SPEC CPU 2017 is a comprehensive benchmark suite containing CPU- and memory-intensive programs written in C, C++, and/or Fortran that stress a computer system's performance. Nginx is a high performance web server written in C that has been widely used in the real world.

For SPEC CPU 2017, we evaluated 28 (out of 43) benchmark programs in C/C++ as LLVM/Clang cannot compile Fortran code. We used the `train` (instead of the larger `ref`) input set because `train` yielded execution time of at least 20 seconds in each benchmark already. We report average execution time with 10 rounds of execution for each benchmark; standard deviations are negligible (less than 1%).

For Nginx, we used Nginx to host randomly generated static files ranging from 1 KB to 512 MB with one worker process listening to port 8080 for HTTP requests. We then ran ApacheBench (`ab`) [6] on the same machine to measure Nginx's bandwidth of transferring files within a period of 10 seconds. We report an average and a standard deviation over 10 rounds of execution for each file size.

Table 4 and Figure 7 present the Baseline performance of SPEC CPU 2017 and Nginx, respectively. Figures 8 and 9 show

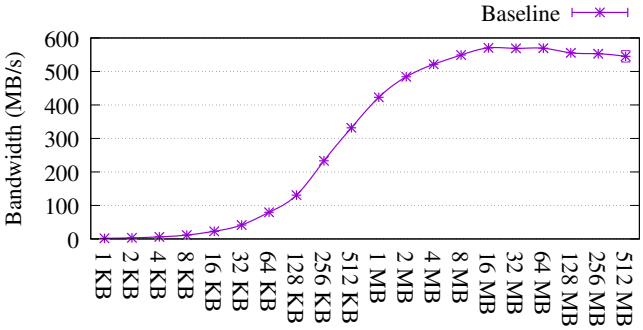

Figure 7. Nginx Bandwidth (Higher is Better)

the performance overhead InversOS incurred on SPEC CPU 2017 and Nginx, respectively. Overall, InversOS increased the execution time of SPEC CPU 2017 by a geometric mean of 7.1% and degraded the bandwidth of Nginx by a geometric mean of 3.0%. We studied the overhead on SPEC CPU 2017 and discovered that our software-based forward-edge CFI caused most of the overhead; with that disabled, the overhead decreased to a geometric mean of 1.9% (in particular, `xalancbmk`'s overhead dropped down from more than 40% to less than 3%). This indicates that InversOS's shadow stack and bit-masking transformations and kernel modifications have minimal performance impact on SPEC CPU 2017, compared with software-based forward-edge CFI. Incorporating BTI [9], we expect InversOS's performance overhead to be greatly reduced; with BTI, no explicit CFI checks (as shown in Figure 5) are needed. However, as BTI does not provide protected shadow stacks by itself, (post-)ARMv8.5-A systems can still leverage InversOS's Privilege Inversion to protect the integrity of shadow stacks. Nginx saw significant variance especially on file sizes ≤ 128 KB. We suspect that the cause of high variance is caching and file system behaviors.

8 Related Work

8.1 Control-Flow Integrity

Since the introduction of the original CFI work [1, 2], a long line of research has been proposed to improve its precision, performance, and/or applicability [4, 12, 14, 16–18, 21, 24, 27, 31, 33–35, 37, 39, 41, 43, 48, 49, 53, 56, 58, 60, 62, 64–68, 74–76, 80, 88, 93, 97–100, 103, 107, 108, 117, 125, 128, 129, 133, 134, 136, 137, 143, 146–151, 153, 155]. As InversOS leverages label-based CFI for forward edges and protected shadow stacks for backward edges, we compare InversOS with various types of CFI schemes.

Stateless CFI. The original CFI [1, 2] restricts forward-edge indirect control-flow targets via a coarse-grained context-insensitive analysis, which statically assigns a distinct label to allowed targets (an equivalence class or EC) of each indirect call or jump and inserts checks for a matched label at indirect call and jump sites. Subsequent research on stateless forward-edge CFI makes trade-offs between granularity and performance [12, 97–99, 107, 125, 129, 134, 148, 150, 151], strengthens other security policies [21, 43, 93, 149], or applies to new platforms [4, 14, 17, 31, 34, 41, 49, 64–66, 100, 108, 133, 137, 153]. Hardware support for stateless forward-edge CFI (such as HAFIX [35], HCFI [27], Intel CET [117], and ARM BTI [9]) has been proposed, which further lowers the performance overhead but only provides coarse-grained protection similar to the original CFI. InversOS's forward-edge CFI, while currently prototyped with two labels, can seamlessly adopt any of the above available finer-grained schemes for better security. It can also utilize BTI on newer processors for better performance.

Stateful CFI. Due to imprecision of context-insensitive CFI, researchers have focused on context-sensitive CFI policies that take previous execution history into account. Using a runtime monitor (inlined or as a separate process), these systems track executed branches [24, 56, 103, 143, 147], paths [39, 58, 128], call-sites [67, 68], code pointer origins [68], or complete control flows [48, 80] to reduce the size of ECs. However, such dynamic CFI schemes require hardware features only found on x86 processors, such as Branch Trace Store (BTS) [143], Last Branch Record (LBR) [24, 103, 128, 147], Performance Monitoring Unit (PMU) [147], Processor Trace (PT) [39, 48, 56, 58, 80], Transactional Synchronization Extensions (TSX) [67, 68], and Memory Protection Extensions (MPX) [68], limiting their applicability on AArch64. Compared with stateful CFI, InversOS offers a weaker protection on forward edges but provides the strongest security on backward edges with better performance and less resource consumption.

Shadow Stacks. The original CFI [1, 2] uses shadow stacks for backward-edge protection; their debut dates back to RAD [25] and StackGhost [46], which all used the compact shadow stack design. Dang et al. [33] proposed the parallel shadow stack design, improving the performance but wasting more memory. As described in Section 2.1, in order to guarantee return address integrity, shadow stacks need a protection mechanism that forbids unauthorized tampering. A few systems [33, 37] simply leave shadow stacks unprotected, while some rely on system calls [25, 46, 133] or SFI [30, 153] for protection but incur prohibitive overhead. More commonly used is information hiding (i.e., ASLR [106]), which places shadow stacks at a random location in the address space to increase the difficulty for attackers to locate the shadow stacks [18, 107, 119, 155]. Though achieving the best performance among software-only solutions, information hiding provides the weakest guarantee and is vulnerable to information disclosure attacks [11, 47, 51, 101, 120, 122]. Hardware-assisted shadow stack protection significantly lowers the performance cost and can be fulfilled differently on different ISAs. On x86_32, segmentation [1, 2] provides the most efficient implementation. CET [117] offers

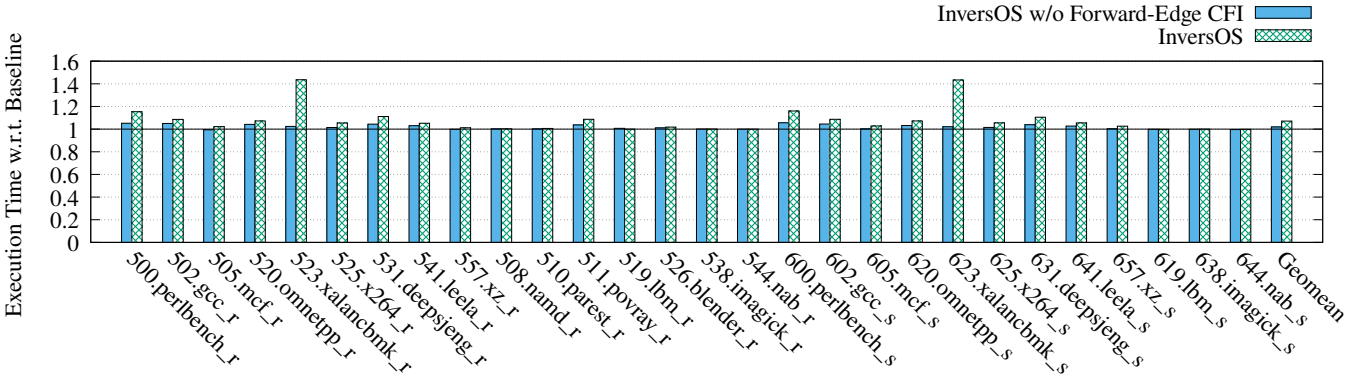

Figure 8. SPEC CPU 2017 Execution Time (Normalized, Lower is Better)

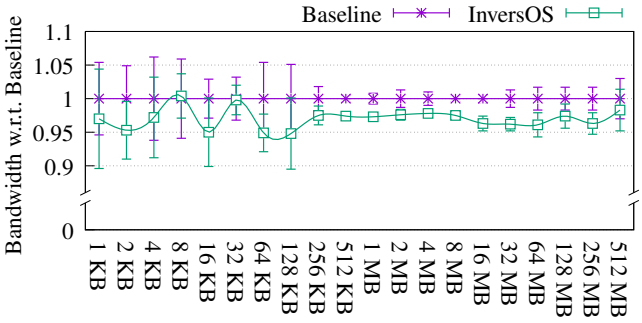

Figure 9. Nginx Bandwidth (Normalized, Higher is Better)

native support for protected shadow stacks on x86_64 but is only available on most recent processors [3, 61]; a few solutions repurposed MPX [18, 60, 65] or MPK [18, 53] for non-CET-equipped Intel processors but reported vastly different overhead numbers. HCFI [27] implements an in-chip non-memory-mapped shadow stack on SPARC via a custom ISA extension. In the microcontroller world, Silhouette [153] and Kage [41] transform regular store instructions into LSU stores on ARMv7-M [8], while CaRE [100] and TZmCFI [66] leverage TrustZone-M on ARMv8-M [10]. To the best of our knowledge, InversOS is the first to provide hardware-assisted protected shadow stacks on AArch64; our Privilege Inversion technique is inspired by Silhouette-Invert [153].

Cryptographic CFI. Mashtizadeh et al. [88] created Cryptographic CFI (CCFI), which uses message authentication codes (MACs) to sign and verify code pointers and leverages x86's AES-NI instructions to accelerate MAC calculation. ARMv8.3-A's PAuth [9] adds hardware support for pointer authentication codes (PACs) and places PACs in unused upper bits of pointers. Qualcomm has adopted PAuth to enforce CFI [110]. However, CCFI and plain PAuth suffer from pointer reuse attacks, in which attackers use buffer overread vulnerabilities [122] to harvest signed pointers for later reuse. Utilizing PAuth, PARTS [76] signs code pointers with type IDs; this limits reuse of signed return addresses within the same functions and signed function pointers within the same types. PACStack [75] and PACtight [62] are also based on PAuth; both solutions sign a return address with the PAC of the previous return address, creating an authenticated stack. PACtight further signs a function pointer with its address and a random tag. Studies on type-ID-based PACs [136] and authenticated chain of return addresses [74] have also been explored on RISC-V as custom ISA extensions. PAL [146] uses PAuth to provide CFI for OS kernels.

As PACStack [75] and PACtight [62] share the most similar threat model, assumptions, and security guarantees with InversOS, we compare InversOS with them in more detail. PACStack claims that its authenticated stack "achieves security comparable to hardware-assisted shadow stacks *without requiring dedicated hardware*"; we show that InversOS achieves hardware-assisted shadow stacks *with even less hardware requirements* (ARMv8.1-A's PAN and HPDS vs. ARMv8.3-A's PAuth). Furthermore, PACStack requires forward-edge CFI but reported performance numbers without accounting its overhead. For an apples-to-apples comparison, InversOS without forward-edge CFI outperforms PACStack (1.9% vs. ≈3.0% on SPEC CPU 2017 and ≤3.0% vs. 6–13% on Nginx). PACtight enforces finer-grained forward-edge CFI than InversOS and its performance (4.0% on Nginx) is roughly on par with InversOS. However, PACtight maintains an in-memory metadata storage for the random tags at runtime and relies on ASLR [106] to hide its location. Essentially, PAC-based systems only offer probabilistic security even if the entropy they provide is large. In contrast, InversOS's shadow stacks are integrity-enforced, providing the strongest guarantees.

Other Approaches. Kuznetsov et al. [71] developed code-pointer integrity (CPI), an approach to ensuring memory safety of all code pointers and data related to code pointers. CPI identifies such data via static analysis and instrumentation and places the data in isolated safe regions. Again, segmentation [3, 61] and ASLR [106] were used to protect the

safe regions on x86_32 and x86_64, respectively. PACtight-CPI [62] implements CPI using PAuth, incurring 4.07% performance overhead on average. InversOS's Privilege Inversion provides an alternative option to protect CPI's safe regions with potentially less overhead. μRAI [4] enforces return address integrity on microcontrollers by encoding return addresses in a reserved register and ensuring that the register value is never corrupted; it relies on system calls to spill the register value to protected memory when needing to fold a call chain longer than what a single register can hold. While μRAI is in theory applicable to general-purpose systems like x86 and AArch64, we believe such an approach provides poor scalability and may incur high performance overhead due to more nested function calls than on microcontrollers.

8.2 Intra-Address Space Isolation

InversOS uses Privilege Inversion for efficient intra-address space isolation. We omit discussing custom hardware modifications that compartmentalize software (e.g., CODOMs [130] and Mondrian [140, 141]) and limit our discussion on related work utilizing recent commodity hardware. Approaches used to enforce CFI are also not repeated here.

SFI [89, 132] instruments program loads and stores to prevent them from accessing certain memory regions and has been used to sandbox untrusted code [70, 115, 145]. While some systems [40, 69] accelerate SFI checks using MPX on x86, the overhead of SFI is still considered high (on both performance [138] and memory usage [18]) and grows as the number of isolated regions increases. Furthermore, SFI often requires CFI to ensure that SFI checks are not bypassed by attacker-manipulated control flow. Another address-based isolation technique is hardware-enforced address range monitoring. PicoXOM [118] enforces execute-only memory (XOM) by configuring ARM debug registers to watch over a code segment against read accesses. Such approaches are limited by hardware resources available and cannot scale up.

Recent defenses enforce domain-based isolation; memory regions are associated with a protection domain, and different mechanisms are used to allow or disallow accesses to the protection domain at runtime. On x86, researchers have explored domain-based memory access control using hardware features such as Virtual Machine Extensions (VMX) [54, 57, 69, 81, 91, 96, 109, 138], MPK [54, 55, 57, 104, 112, 113, 123, 127, 131, 135], SMAP [138], and CET [144]. ARMlock [154] and Shreds [23] use ARM domains, which are only available on AArch32 [9]. Previous work has also used LSU instructions for isolation. ILDI [26] utilizes LSU instructions and PAN to protect a safe region inside the OS kernel; it relies on a more privileged hypervisor to moderate sensitive kernel operations. uXOM [72] transforms regular loads/stores to LSU instructions to enforce XOM on microcontrollers, where application code typically executes in the privileged mode already. InversOS, employing Privilege Inversion, is the first to extend domain-based isolation to AArch64 user space.

We notice that Privbox [70] and SEIMI [138], like InversOS, also proposed executing user-space code in the privileged mode (x86's ring 0). Privbox does so to accelerate system call invocation and uses SFI to safely run elevated code. The overhead of its heavy instrumentation, however, may outweigh its speedup from faster system calls on certain programs. InversOS can benefit from the idea of system call acceleration for elevated tasks, which we leave as future work. SEIMI flips SMAP (x86 equivalence to PAN) to create a safe region for trusted user-space code; its OS kernel is then elevated to run in ring -1 via VMX. Compared with SEIMI, InversOS's Privilege Inversion provides instruction-level isolation and requires no frequent domain switching.

9 Conclusions and Future Work

In conclusion, we presented InversOS, a hardware-assisted protected shadow stack implementation for AArch64, which utilizes common hardware features to create novel and efficient intra-address space isolation and safely executes user-space code in the privileged mode via OS kernel and compiler restraints. Our analysis shows that InversOS is secure and effective in mitigating attacks, and our performance evaluation demonstrates the low costs of InversOS on real-world benchmarks and applications. We will open-source our prototype of InversOS in a timely manner.

We see several directions for future work. First, we can explore system call optimizations (such as Privbox [70]) for elevated tasks; these tasks already run in the privileged mode and can accelerate system call invocation by avoiding the costly SVC instructions. Second, we can leverage Privilege Inversion to enforce other security policies such as CPI [71] and full memory safety [38, 94, 95, 152], reducing their overheads significantly. Finally, we intend to investigate potential performance improvements to InversOS by using more recent ISA features (e.g., BTI and E0PD) on real hardware.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*. ACM, Alexandria, VA, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security* 13, 1, Article 4 (Nov. 2009), 40 pages. https://doi.org/10.1145/1609956.1609960

[3] Advanced Micro Devices Inc. 2023. *AMD64 Architecture Programmer's Manual*. Advanced Micro Devices Inc. https://www.amd.com/en/support/tech-docs/amd64-architecture-programmers-manual-volumes-1-5 40332 Rev 4.06.

[4] Naif Saleh Almakhdhub, Abraham A. Clements, Saurabh Bagchi, and Mathias Payer. 2020. μRAI: Securing Embedded Systems with Return Address Integrity. In *Proceedings of the 2020 Network and Distributed System Security Symposium (NDSS '20)*. Internet Society, San Diego, CA, USA, 18 pages. https://doi.org/10.14722/ndss.2020.24016

[5] Amazon Web Services 2023. *Amazon EC2 A1 Instances: Optimized cost and performance for scale-out workloads.* https://aws.amazon.com/ec2/instance-types/a1

[6] Apache 2023. *ab - Apache HTTP server benchmarking tool.* https://httpd.apache.org/docs/current/programs/ab.html

[7] Apple 2020. *Apple unleashes M1.* https://www.apple.com/newsroom/2020/11/apple-unleashes-m1

[8] Arm Holdings 2021. *Arm® v7-M Architecture Reference Manual.* Arm Holdings. https://developer.arm.com/documentation/ddi0403/ee DDI 0403E.e.

[9] Arm Holdings 2022. *Arm® Architecture Reference Manual: for A-profile architecture.* Arm Holdings. https://developer.arm.com/documentation/ddi0487/ia DDI 0487I.a.

[10] Arm Holdings 2022. *Arm® v8-M Architecture Reference Manual.* Arm Holdings. https://developer.arm.com/documentation/ddi0553/bv DDI 0553B.v.

[11] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, San Jose, CA, USA, 227–242. https://doi.org/10.1109/SP.2014.22

[12] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. 2011. Mitigating Code-Reuse Attacks with Control-Flow Locking. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*. ACM, Orlando, FL, USA, 353–362. https://doi.org/10.1145/2076732.2076783

[13] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11)*. ACM, Hong Kong, China, 30–40. https://doi.org/10.1145/1966913.1966919

[14] Dimitar Bounov, Rami Gökhan Kıcı, and Sorin Lerner. 2016. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS '16)*. Internet Society, San Diego, CA, USA, 15 pages. https://doi.org/10.14722/ndss.2016.23421

[15] Daniel P. Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel* (3rd ed.). O'Reilly & Associates Inc, Sebastopol, CA, USA.

[16] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *Comput. Surveys* 50, 1, Article 16 (April 2017), 33 pages. https://doi.org/10.1145/3054924

[17] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. 2018. CFIXX: Object Type Integrity for C++. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS '18)*. Internet Society, San Diego, CA, USA, 14 pages. https://doi.org/10.14722/ndss.2018.23279

[18] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP '19)*. IEEE Computer Society, San Francisco, CA, USA, 985–999. https://doi.org/10.1109/SP.2019.00076

[19] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (Security '15)*. USENIX Association, Washington, DC, USA, 161–176. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[20] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the 23rd USENIX Security Symposium (Security '14)*. USENIX Association, San Diego, CA, USA, 385–399. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini

[21] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. 2009. Fast Byte-Granularity Software Fault Isolation. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP '09)*. ACM, Big Sky, MT, USA, 45–58. https://doi.org/10.1145/1629575.1629581

[22] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium (Security '05)*. USENIX Association, Baltimore, MD, USA, 177–191. https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats

[23] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-Grained Execution Units with Private Memory. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP '16)*. IEEE Computer Society, San Jose, CA, USA, 56–71. https://doi.org/10.1109/SP.2016.12

[24] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. 2014. ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks. In *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS '14)*. Internet Society, San Diego, CA, USA, 14 pages. https://doi.org/10.14722/ndss.2014.23156

[25] Tzi-Cker Chiueh and Fu-Hau Hsu. 2001. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS '01)*. IEEE Computer Society, Mesa, AZ, USA, 409–417. https://doi.org/10.1109/ICDSC.2001.918971

[26] Yeongpil Cho, Donghyun Kwon, and Yunheung Paek. 2017. Instruction-Level Data Isolation for the Kernel on ARM. In *Proceedings of the 54th ACM/EDAC/IEEE Annual Design Automation Conference (DAC '17)*. ACM, Austin, TX, USA, Article 26, 6 pages. https://doi.org/10.1145/3061639.3062267

[27] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. 2016. HCFI: Hardware-Enforced Control-Flow Integrity. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY '16)*. ACM, New Orleans, LA, USA, 38–49. https://doi.org/10.1145/2857705.2857722

[28] Tobias Cloosters, David Paaßen, Jianqiang Wang, Oussama Draissi, Patrick Jauernig, Emmanuel Stapf, Lucas Davi, and Ahmad-Reza Sadeghi. 2022. RiscyROP: Automated Return-Oriented Programming Attacks on RISC-V and ARM64. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '22)*. ACM, Limassol, Cyprus, 30–42. https://doi.org/10.1145/3545948.3545997

[29] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, Denver, CO, USA, 952–963. https://doi.org/10.1145/2810103.2813671

[30] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. 2005. Using DISE to Protect Return Addresses from Attack. *SIGARCH Computer Architecture News* 33, 1 (March 2005), 65–72. https://doi.org/10.1145/1055626.1055636

[31] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, San Jose, CA, USA, 292–307. https://doi.org/10.1109/SP.2014.26

[32] Christoffer Dall and Jason Nieh. 2014. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, Salt Lake City, UT, USA, 333–348. https://doi.org/10.1145/2541940.2541946

[33] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS '15)*. ACM, Singapore, Republic of Singapore, 555–566. https://doi.org/10.1145/2714576.2714635

[34] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2012. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proceedings of the 2012 Network and Distributed System Security Symposium (NDSS '12)*. Internet Society, San Diego, CA, USA, 17 pages. https://www.ndss-symposium.org/ndss2012/ndss-2012-programme/mocfi-framework-mitigate-control-flow-attacks-smartphonesoverlay-contextmocfi-framework-mitigate-control-flow-attacks-smartphones

[35] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. 2015. HAFIX: Hardware-Assisted Flow Integrity Extension. In *Proceedings of the 52nd ACM/EDAC/IEEE Annual Design Automation Conference (DAC '15)*. ACM, San Francisco, CA, USA, Article 74, 6 pages. https://doi.org/10.1145/2744769.2744847

[36] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Security Symposium (Security '14)*. USENIX Association, San Diego, CA, USA, 401–416. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi

[37] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: A Detection Tool to Defend against Return-Oriented Programming Attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11)*. ACM, Hong Kong, China, 40–51. https://doi.org/10.1145/1966913.1966920

[38] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, Ottawa, ON, Canada, 144–157. https://doi.org/10.1145/1133981.1133999

[39] Ren Ding, Chenxiong Qian, Chengyu Song, William Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient Protection of Path-Sensitive Control Security. In *Proceedings of the 26th USENIX Security Symposium (Security '17)*. USENIX Association, Vancouver, BC, Canada, 131–148. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding

[40] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. 2018. Shielding Software from Privileged Side-Channel Attacks. In *Proceedings of the 27th USENIX Security Symposium (Security '18)*. USENIX Association, Baltimore, MD, USA, 1441–1458. https://www.usenix.org/conference/usenixsecurity18/presentation/dong

[41] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J. Walls, and John Criswell. 2022. Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage. In *Proceedings of the 31st USENIX Security Symposium (Security '22)*. USENIX Association, Boston, MA, USA, 2281–2298. https://www.usenix.org/conference/usenixsecurity22/presentation/du

[42] Alexandru Elisei. 2019. bhyvearm64: CPU and Memory Virtualization on Armv8.0-A. In *The BSDCan Conference*. Ottawa, ON, Canada. https://www.bsdcan.org/2019/schedule/events/1074.en.html

[43] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Seattle, WA, USA, 75–88. https://www.usenix.org/conference/osdi-06/xfi-software-guards-system-address-spaces

[44] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, Denver, CO, USA, 901–913. https://doi.org/10.1145/2810103.2813646

[45] Rich Felker et al. 2021. *musl libc*. https://musl.libc.org

[46] Mike Frantzen and Mike Shuey. 2001. StackGhost: Hardware Facilitated Stack Protection. In *Proceedings of the 10th USENIX Security Symposium (Security '01)*. USENIX Association, Washington, DC, USA, 11 pages. https://www.usenix.org/conference/10th-usenix-security-symposium/stackghost-hardware-facilitated-stack-protection

[47] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. 2016. Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS '16)*. Internet Society, San Diego, CA, USA, 15 pages. https://doi.org/10.14722/ndss.2016.23262

[48] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. Griffin: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, Xi'an, China, 585–598. https://doi.org/10.1145/3037697.3037716

[49] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-Grained Control-Flow Integrity for Kernel Software. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroSP '16)*. IEEE Computer Society, Saarbruecken, Germany, 179–194. https://doi.org/10.1109/EuroSP.2016.24

[50] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, San Jose, CA, USA, 575–589. https://doi.org/10.1109/SP.2014.43

[51] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining Information Hiding (and What to Do about It). In *Proceedings of the 25th USENIX Security Symposium (Security '16)*. USENIX Association, Austin, TX, USA, 105–119. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/goktas

[52] Google Cloud 2023. *Arm VMs on Compute*. https://cloud.google.com/compute/docs/instances/arm-on-compute

[53] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. 2021. Fast Intra-Kernel Isolation and Security with IskiOS. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21)*. ACM, San Sebastian, Spain, 119–134. https://doi.org/10.1145/3471621.3471849

[54] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. EPK: Scalable and Efficient Memory Protection Keys. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC '22)*. USENIX Association, Carlsbad, CA, USA, 609–624. https://www.usenix.org/conference/atc22/presentation/gu-jinyu

[55] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. 2020. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-Kernel Isolation and Communication. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC '20)*. USENIX Association, Virtual Event, 401–417. https://www.usenix.org/conference/atc20/presentation/gu

[56] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY '17)*. ACM, Scottsdale, AZ, USA, 173–184. https://doi.org/10.1145/3029806.3029830

[57] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC '19)*. USENIX Association, Renton, WA, USA, 489–503. https://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[58] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, Toronto, ON, Canada, 1470–1486. https://doi.org/10.1145/3243734.3243797

[59] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP '16)*. IEEE Computer Society, San Jose, CA, USA, 969–986. https://doi.org/10.1109/SP.2016.62

[60] Wei Huang, Zhen Huang, Dhaval Miyani, and David Lie. 2016. LMP: Light-Weighted Memory Protection with Hardware Assistance. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC '16)*. ACM, Los Angeles, CA, USA, 460–470. https://doi.org/10.1145/2991079.2991089

[61] Intel Corporation 2022. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html Order Number: 325462-078US.

[62] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. 2022. Tightly Seal Your Sensitive Pointers with PACTight. In *Proceedings of the 31st USENIX Security Symposium (Security '22)*. USENIX Association, Boston, MA, USA, 3717–3734. https://www.usenix.org/conference/usenixsecurity22/presentation/ismail

[63] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, Toronto, ON, Canada, 1868–1882. https://doi.org/10.1145/3243734.3243739

[64] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS '14)*. Internet Society, San Diego, CA, USA, 15 pages. https://doi.org/10.14722/ndss.2014.23287

[65] Ethan Johnson, Colin Pronovost, and John Criswell. 2022. Hardening Hypervisors with Ombro. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC '22)*. USENIX Association, Carlsbad, CA, USA, 415–435. https://www.usenix.org/conference/atc22/presentation/johnson

[66] Tomoaki Kawada, Shinya Honda, Yutaka Matsubara, and Hiroaki Takada. 2021. TZmCFI: RTOS-Aware Control-Flow Integrity Using TrustZone for Armv8-M. *International Journal of Parallel Programming* 49 (April 2021), 216–236. https://doi.org/10.1007/s10766-020-00673-z

[67] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. 2019. Adaptive Call-Site Sensitive Control Flow Integrity. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroSP '19)*. IEEE Computer Society, Stockholm, Sweden, 95–110. https://doi.org/10.1109/EuroSP.2019.00017

[68] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. 2019. Origin-Sensitive Control Flow Integrity. In *Proceedings of the 28th USENIX Security Symposium (Security '19)*. USENIX Association, Santa Clara, CA, USA, 195–211. https://www.usenix.org/conference/usenixsecurity19/presentation/khandaker

[69] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys '17)*. ACM, Belgrade, Serbia, 437–452. https://doi.org/10.1145/3064176.3064217

[70] Dmitry Kuznetsov and Adam Morrison. 2022. Privbox: Faster System Calls Through Sandboxed Privileged Execution. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC '22)*. USENIX Association, Carlsbad, CA, USA, 233–247. https://www.usenix.org/conference/atc22/presentation/kuznetsov

[71] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association, Broomfield, CO, USA, 147–163. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[72] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. 2019. uXOM: Efficient eXecute-Only Memory on ARM Cortex-M. In *Proceedings of the 28th USENIX Security Symposium (Security '19)*. USENIX Association, Santa Clara, CA, USA, 231–247. https://www.usenix.org/conference/usenixsecurity19/presentation/kwon

[73] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO '04)*. IEEE Computer Society, Palo Alto, CA, USA, 12 pages. https://doi.org/10.1109/CGO.2004.1281665

[74] Jinfeng Li, Liwei Chen, Qizhen Xu, Linan Tian, Gang Shi, Kai Chen, and Dan Meng. 2020. Zipper Stack: Shadow Stacks Without Shadow. In *Proceedings of the 25th European Symposium on Research in Computer Security (ESORICS '20)*. Springer-Verlag, Guildford, UK, 338–358. https://doi.org/10.1007/978-3-030-58951-6_17

[75] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. 2021. PACStack: an Authenticated Call Stack. In *Proceedings of the 30th USENIX Security Symposium (Security '21)*. USENIX Association, Virtual Event, 357–374. https://www.usenix.org/conference/usenixsecurity21/presentation/liljestrand

[76] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *Proceedings of the 28th USENIX Security Symposium (Security '19)*. USENIX Association, Santa Clara, CA, USA, 177–194. https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand

[77] Linux 2020. *Linux Kernel Source Tree v5.6*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/?h=v5.6

[78] Linux 2021. *Linux Kernel Stable Tree v4.19.219*. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/?h=v4.19.219

[79] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (Security '18)*. USENIX Association, Baltimore, MD, USA, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[80] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA '17)*. IEEE Computer Society, Austin, TX, USA, 529–540. https://doi.org/10.1109/HPCA.2017.18

[81] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS*

'15). ACM, Denver, CO, USA, 1607–1619. https://doi.org/10.1145/2810103.2813690

[82] LLVM 2021. *"libc++" C++ Standard Library*. https://libcxx.llvm.org

[83] LLVM 2021. *"libc++abi" C++ Standard Library Support*. https://libcxxabi.llvm.org

[84] LLVM 2021. *libunwind LLVM Unwinder*. https://bcain-llvm.readthedocs.io/projects/libunwind

[85] LLVM 2022. *"compiler-rt" runtime libraries*. https://compiler-rt.llvm.org

[86] LLVM 2023. *lib/CodeGen/IndirectBrExpandPass.cpp File Reference*. https://llvm.org/doxygen/IndirectBrExpandPass_8cpp.html

[87] LLVM 2023. *LLD - The LLVM Linker*. https://lld.llvm.org

[88] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, Denver, CO, USA, 941–951. https://doi.org/10.1145/2810103.2813676

[89] Stephen McCamant and Greg Morrisett. 2006. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th USENIX Security Symposium (Security '06)*. USENIX Association, Vancouver, BC. Canada, 209–224. https://www.usenix.org/conference/15th-usenix-security-symposium/evaluating-sfi-cisc-architecture

[90] Larry McVoy and Carl Staelin. 1996. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC '96)*. USENIX Association, San Diego, CA, USA, 16 pages. https://www.usenix.org/legacy/publications/library/proceedings/sd96/mcvoy.html

[91] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys '19)*. ACM, Dresden, Germany, Article 9, 15 pages. https://doi.org/10.1145/3302424.3303946

[92] Microsoft Azure 2022. *Azure Virtual Machines with Ampere Altra Arm–based processors—generally available*. https://azure.microsoft.com/en-us/blog/azure-virtual-machines-with-ampere-altra-arm-based-processors-generally-available

[93] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. 2015. Opaque Control-Flow Integrity. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS '15)*. Internet Society, San Diego, CA, USA, 15 pages. https://doi.org/10.14722/ndss.2015.23271

[94] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, Dublin, Ireland, 245–258. https://doi.org/10.1145/1542476.1542504

[95] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM '10)*. ACM, Toronto, ON, Canada, 31–40. https://doi.org/10.1145/1806651.1806657

[96] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2020. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*. ACM, Lausanne, Switzerland, 157–171. https://doi.org/10.1145/3381052.3381328

[97] Ben Niu and Gang Tan. 2013. Monitor Integrity Protection with Space Efficiency and Separate Compilation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS '13)*. ACM, Berlin, Germany, 199–210. https://doi.org/10.1145/2508859.2516649

[98] Ben Niu and Gang Tan. 2014. Modular Control-Flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, Edinburgh, UK, 577–587. https://doi.org/10.1145/2594291.2594295

[99] Ben Niu and Gang Tan. 2015. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, Denver, CO, USA, 914–926. https://doi.org/10.1145/2810103.2813644

[100] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N. Asokan. 2017. CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID '17)*. Springer-Verlag, Atlanta, GA, USA, 259–284. https://doi.org/10.1007/978-3-319-66332-6_12

[101] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking Holes in Information Hiding. In *Proceedings of the 25th USENIX Security Symposium (Security '16)*. USENIX Association, Austin, TX, USA, 121–138. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/oikonomopoulos

[102] Oracle Cloud Infrastructure 2023. *Ampere A1 Compute*. https://www.oracle.com/cloud/compute/arm

[103] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22nd USENIX Security Symposium (Security '13)*. USENIX Association, Washington, DC, USA, 447–462. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/pappas

[104] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC '19)*. USENIX Association, Renton, WA, USA, 241–254. https://www.usenix.org/conference/atc19/presentation/park-soyeon

[105] PaX Team 2000. *Non-Executable Pages Design & Implementation*. https://pax.grsecurity.net/docs/noexec.txt

[106] PaX Team 2001. *Address Space Layout Randomization*. https://pax.grsecurity.net/docs/aslr.txt

[107] Mathias Payer, Antonio Barresi, and Thomas R. Gross. 2015. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '15)*. Springer-Verlag, Milan, Italy, 144–164. https://doi.org/10.1007/978-3-319-20550-2_8

[108] Jannik Pewny and Thorsten Holz. 2013. Control-Flow Restrictor: Compiler-Based CFI for iOS. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC '13)*. ACM, New Orleans, LA, USA, 309–318. https://doi.org/10.1145/2523649.2523674

[109] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP '20)*. IEEE Computer Society, San Francisco, CA, USA, 563–577. https://doi.org/10.1109/SP40000.2020.00041

[110] Qualcomm 2017. *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions*. White Paper. Qualcomm Technologies, Inc. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf

[111] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security* 15, 1, Article 2 (March 2012), 34 pages. https://doi.org/10.1145/2133375.2133377

[112] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. ACM, Virtual Event, 546–558. https://doi.org/10.1145/3445814.3446731

[113] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *Proceedings of the 29th USENIX Security Symposium (Security '20)*. USENIX Association, Boston, MA, USA, 1677–1694. https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel

[114] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, San Jose, CA, USA, 745–762. https://doi.org/10.1109/SP.2015.51

[115] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the 19th USENIX Security Symposium (Security '10)*. USENIX Association, Washington, DC, USA, 11 pages. https://www.usenix.org/conference/usenixsecurity10/adapting-software-fault-isolation-contemporary-cpu-architectures

[116] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, Alexandria, VA, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[117] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '19)*. ACM, Phoenix, AZ, USA, Article 8, 11 pages. https://doi.org/10.1145/3337167.3337175

[118] Zhuojia Shen, Komail Dharsee, and John Criswell. 2020. Fast Execute-Only Memory for Embedded Systems. In *Proceedings of the 2020 IEEE Secure Development Conference (SecDev '20)*. IEEE Computer Society, Atlanta, GA, USA, 7–14. https://doi.org/10.1109/SecDev45635.2020.00017

[119] Zhuojia Shen, Komail Dharsee, and John Criswell. 2022. Randezvous: Making Randomization Effective on MCUs. In *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC '22)*. ACM, Austin, TX, USA, 28–41. https://doi.org/10.1145/3564625.3567970

[120] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, San Francisco, CA, USA, 574–588. https://doi.org/10.1109/SP.2013.45

[121] Standard Performance Evaluation Corporation 2022. *SPEC CPU®2017*. https://www.spec.org/cpu2017

[122] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. 2009. Breaking the Memory Secrecy Assumption. In *Proceedings of the 2nd European Workshop on System Security (EuroSec '09)*. ACM, Nuremburg, Germany, 1–8. https://doi.org/10.1145/1519144.1519145

[123] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*. ACM, Lausanne, Switzerland, 143–156. https://doi.org/10.1145/3381052.3381326

[124] Igor Sysoev et al. 2022. *nginx*. https://nginx.org/en

[125] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium (Security '14)*. USENIX Association, San Diego, CA, USA, 941–955. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice

[126] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. 2011. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID '11)*. Springer-Verlag, Menlo Park, CA, USA, 121–141. https://doi.org/10.1007/978-3-642-23644-0_7

[127] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium (Security '19)*. USENIX Association, Santa Clara, CA, USA, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[128] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, Denver, CO, USA, 927–940. https://doi.org/10.1145/2810103.2813673

[129] Victor van der Veen, Enes Göktaş, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough `call`: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP '16)*. IEEE Computer Society, San Jose, CA, USA, 934–953. https://doi.org/10.1109/SP.2016.60

[130] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting Software with Code-Centric Memory Domains. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Computer Society, Minneapolis, MN, USA, 469–480. https://doi.org/10.1109/ISCA.2014.6853202

[131] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-Based Sandboxing. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys '22)*. ACM, Rennes, France, 266–282. https://doi.org/10.1145/3492321.3519560

[132] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM, Asheville, NC, USA, 203–216. https://doi.org/10.1145/168619.168635

[133] Robert J. Walls, Nicholas F. Brown, Thomas Le Baron, Craig A. Shue, Hamed Okhravi, and Bryan C. Ward. 2019. Control-Flow Integrity for Real-Time Embedded Systems. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS '19)*. Schloss Dagstuhl–Leibniz-Zentrum füer Informatik, Stuttgart, Germany, 2:1–2:24. https://doi.org/10.4230/LIPIcs.ECRTS.2019.2

[134] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. 2015. Binary Code Continent: Finer-Grained Control Flow Integrity for Stripped Binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC '15)*. ACM, Los Angeles, CA, USA, 331–340. https://doi.org/10.1145/2818000.2818017

[135] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. 2020. Secure and Efficient In-Process Monitor (and Library) Protection with Intel MPK. In *Proceedings of the 13th European Workshop on Systems Security (EuroSec '20)*. ACM, Heraklion, Greece, 7–12. https://doi.org/10.1145/3380786.3391398

[136] Yu Wang, Jinting Wu, Tai Yue, Zhenyu Ning, and Fengwei Zhang. 2022. RetTag: Hardware-Assisted Return Address Integrity on RISC-V. In *Proceedings of the 15th European Workshop on Systems Security (EuroSec '22)*. ACM, Rennes, France, 50–56. https://doi.org/10.1145/3517208.3523758

[137] Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, Oakland, CA, USA, 380–395. https://doi.org/10.1109/SP.2010.30

[138] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. 2020. SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation. In *Proceedins of the 2020 IEEE Symposium on Security and Privacy (SP '20)*. IEEE Computer Society, San Francisco, CA, USA, 592–607. https://doi.org/10.1109/SP40000.2020.00087

[139] Wikipedia 2023. *Comparison of ARM processors*. https://en.wikipedia.org/wiki/Comparison_of_ARM_processors#ARMv8-A

[140] Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '02)*. ACM, San Jose, CA, USA, 304–316. https://doi.org/10.1145/605397.605429

[141] Emmett Witchel, Junghwan Rhee, and Krste Asanović. 2005. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, Brighton, UK, 31–44. https://doi.org/10.1145/1095810.1095814

[142] XAMPPRocky et al. 2021. *Tokei: Count your code, quickly.* https://github.com/XAMPPRocky/tokei

[143] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. 2012. CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '12)*. IEEE Computer Society, Boston, MA, USA, 12 pages. https://doi.org/10.1109/DSN.2012.6263958

[144] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. 2022. CETIS: Retrofitting Intel CET for Generic and Efficient Intra-Process Memory Isolation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. ACM, Los Angeles, CA, USA, 2989–3002. https://doi.org/10.1145/3548606.3559344

[145] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy (SP '09)*. IEEE Computer Society, Oakland, CA, USA, 79–93. https://doi.org/10.1109/SP.2009.25

[146] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. 2022. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In *Proceedings of the 31st USENIX Security Symposium (Security '22)*. USENIX Association, Boston, MA, USA, 89–106. https://www.usenix.org/conference/usenixsecurity22/presentation/yoo

[147] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. 2015. Hardware-Assisted Fine-Grained Code-Reuse Attack Detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID '15)*. Springer-Verlag, Kyoto, Japan, 66–85. https://doi.org/10.1007/978-3-319-26362-5_4

[148] Bin Zeng, Gang Tan, and Úlfar Erlingsson. 2013. Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors. In *Proceedings of the 22nd USENIX Security Symposium (Security '13)*. USENIX Association, Washington, DC, USA, 369–382. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/zeng

[149] Bin Zeng, Gang Tan, and Greg Morrisett. 2011. Combining Control-Flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, Chicago, IL, USA, 29–40. https://doi.org/10.1145/2046707.2046713

[150] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, San Francisco, CA, USA, 559–573. https://doi.org/10.1109/SP.2013.44

[151] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium (Security '13)*. USENIX Association, Washington, DC, USA, 337–352. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang

[152] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, Providence, RI, USA, 631–644. https://doi.org/10.1145/3297858.3304017

[153] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J. Walls. 2020. Silhouette: Efficient Protected Shadow Stacks for Embedded Systems. In *Proceedings of the 29th USENIX Security Symposium (Security '20)*. USENIX Association, Boston, MA, USA, 1219–1236. https://www.usenix.org/conference/usenixsecurity20/presentation/zhou-jie

[154] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014. ARMlock: Hardware-Based Fault Isolation for ARM. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS '14)*. ACM, Scottsdale, AZ, USA, 558–569. https://doi.org/10.1145/2660267.2660344

[155] Philipp Zieris and Julian Horsch. 2018. A Leak-Resilient Dual Stack Scheme for Backward-Edge Control-Flow Integrity. In *Proceedings of the 2018 ACM Asia Conference on Computer and Communications Security (ASIACCS '18)*. ACM, Incheon, Republic of Korea, 369–380. https://doi.org/10.1145/3196494.3196531