Интересное исследование, которое было названо PANIC (PAN-assisted Intra-process Memory Isolation on ARM), предлагает гибкий и эффективный механизм внутрипроцессной изоляции памяти, используя инструкции Privileged Access Never (PAN) и load/store unprivileged (LSU) на AArch64. PANIC выполняет процессное кодирование в режиме ядра и разделяет код на доверенные и недоверенные компоненты. Недоверенный код ограничен в доступе к изолированной памяти, которая расположена на страницах пользователя, а доверенный код имеет доступ к изолированной памяти с помощью инструкций LSU. Для снижения угроз, связанных с выполнением кода пользователя в режиме ядра, PANIC предоставляет два новых механизма безопасности: изоляция памяти на основе шимов и эмуляция чувствительных инструкций.

PANIC предоставляет гибкий и эффективный механизм изоляции, который может быть использован в трех различных сценариях: защите чувствительных данных в CFI, создании изолированных окружений выполнения и упрочнении JIT кода кэша. Возможности прототипа PANIC были реализованы и экспериментально оценены, показав, что PANIC склоняет низкий перенос производительности и работает лучше, чем существующие методы.

**CCS CONCEPTS**

- **Security and privacy ➔ Software and application security.**

**KEYWORDS**

Intra-process Memory Isolation, ARM, LSU, UAO, PAN, Isolated Execution Environment

**ACM Reference Format:**


1 INTRODUCTION

Внутрипроцессная изоляция памяти является хорошо известным механизмом для выполнения принципа наименьших привилегий внутри одного приложения. В этом исследовании, мы предлагаем гибкий и эффективный внутрипроцессный механизм изоляции памяти, известный как PANIC, используя инструкции Privileged Access Never (PAN) и load/store unprivileged (LSU) на AArch64. PANIC выполняет процессный код в режиме ядра и разделяет код на доверенные и недоверенные компоненты. Недоверенный код ограничен в доступе к изолированной памяти, которая расположена на страницах пользователя, а доверенный код имеет доступ к изолированной памяти с помощью инструкций LSU. Для снижения угроз, связанных с выполнением кода пользователя в режиме ядра, PANIC предоставляет два новых механизма безопасности: изоляция памяти на основе шимов и эмуляция чувствительных инструкций.

PANIC предоставляет гибкий и эффективный механизм изоляции, который может быть использован в трех различных сценариях: защите чувствительных данных в CFI, создании изолированных окружений выполнения и упрочнении JIT кода кэша. Возможности прототипа PANIC были реализованы и экспериментально оценены, показав, что PANIC склоняет низкий перенос производительности и работает лучше, чем существующие методы.

**CCS CONCEPTS**

- **Security and privacy ➔ Software and application security.**

**KEYWORDS**

Intra-process Memory Isolation, ARM, LSU, UAO, PAN, Isolated Execution Environment
restricting code in some compartments from accessing data of others, thereby isolating faults or attacks within these compartments. Example use cases of intra-process memory isolation techniques include those isolating code pointers to prevent control-flow hijacking [8, 26, 33]; isolating cryptographic keys in OpenSSL servers to thwart the Heartbleed attack [29, 51]; isolating JITed code in Just-in-Time (JIT) compilers to prevent code injection [48, 49, 57].

While new features in ARM processors, such as Pointer Authentication [2] and Memory Tagging Extension [2] have been designed to enforce memory safety, no generic and efficient intra-process memory isolation technique has been proposed for 64-bit ARM (i.e., AArch64). Existing intra-process memory isolation techniques are roughly classified into address-based isolation and domain-based isolation. Address-based isolation restricts each memory access from the untrusted code to ensure that the isolated memory region cannot be accessed. Software Fault Isolation (SFI) [52] is one prominent address-based isolation technique. It needs to instrument all memory access instructions of the untrusted code to ensure that untrusted code can only access predetermined memory regions. The extensive code instrumentation usually leads to severe code bloats and poor performance for memory-intensive applications.

Domain-based isolation enables the access permission of the isolated memory region before the trust code accesses it, and revokes the permission when the access finishes. On ARM platforms, for example, Shreds [11] and ARMLock [60] leveraged a feature called memory domains [1], which is obsolete on AArch64, to provide intra-process memory isolation by isolating the isolated memory region within one memory domain. Hardware watchpoint has also been used to achieve intra-process memory isolation by enabling watchpoint read/write monitoring for the isolated memory region [27]. However, as configuring access rights requires privileged instructions, switching domain still needs to be trapped into the kernel, which induces high performance overhead.

To date, there is no efficient and generic intra-process memory isolation technique available on ARM. In this paper, we propose a new such technique, called PANIC. PANIC relies on AArch64’s Privileged Access Never (PAN) feature and load/store unprivileged (LSU) instructions. PAN is a means to prevent kernel code from accessing user data [2]. It is typically used as a security measure against return-to-user attacks [30]. If the kernel code does need access of the user data, either PAN has to be disabled, or LSU instructions can be used. The LSU instructions are memory access instructions that are not subject to PAN’s restrictions.

To provide intra-process memory isolation, PANIC runs the protected user process in kernel mode, and configures memory regions to which it wishes to restrict the process’s memory accesses as user pages. The protected process can thus be partitioned into trusted and untrusted code, according to whether the code needs to access the isolated memory regions or not. As the trusted code needs to access the isolated memory regions, it is revised to use LSU instructions instead of normal load/store instructions to do so. LSU instructions and the instructions to enable/disable PAN are disallowed in the untrusted code to ensure isolation.

However, the need of running user code in kernel mode brings forth new security threats. When the user code runs at higher privilege levels, it becomes capable of accessing all kernel memory pages and executing sensitive instructions that should not be permitted in the user code. Therefore, to secure PANIC, we propose two novel techniques: shim-based memory isolation that offer memory isolation between the kernel and the protected process, and sensitive instruction emulation that emulates these instructions on the fly.

- **Shim-based memory isolation.** When running the protected process in kernel mode, the traditional user-kernel isolation is no longer effective: without any restriction, the user code could access the kernel memory and the kernel code could also access the process memory. To address this, we propose a shim-based memory isolation between the kernel and the protected process. The shim is split into a user shim and a kernel shim, and interposes all control transfers between the user and the kernel. Specifically, EPD0/EPD1 are managed by the shim to enable and disable page table walk of the user/kernel, and ASID is leveraged to isolate the TLB entries used in the user and the kernel. To protect the user shim from the untrusted user process, it is designed to be as small as possible (e.g., with only one code page) and ensures the following two security properties: atomicity, the execution sequence of the instructions cannot be manipulated, and determinacy, the execution result is deterministic regardless of the environment. These two properties ensure that the shim-based isolation mechanism cannot be compromised.

- **Sensitive instruction emulation.** An instruction is sensitive if its behavior is inconsistent in user mode and kernel mode. A privileged instruction is also a sensitive instruction in that it causes undefined behaviors when executed in user mode. When running user code in kernel mode, sensitive instructions in the protected program may be executed, which can be abused to compromise the whole system. To address this, we comprehensively analyzed all ARMv8-A instructions and identified 874 sensitive instructions. We further classified them into unconditionally and conditionally sensitive instructions based on whether the behavioral difference depends on the system configuration. To prevent the illegal execution of sensitive instructions, PANIC handles sensitive instructions as follows: it first performs a binary inspection to identify all sensitive instructions in the user code page right before the page becomes executable; it then transforms it to an instruction that raises exceptions when executed—unconditionally and conditionally sensitive instructions are transformed to trigger different types of exceptions; when the exceptions are raised, PANIC is invoked to emulate the behavior of the original instruction as if it runs in user mode.

PANIC provides a primitive of intra-process memory isolation that can be applied broadly. We especially demonstrate the use of PANIC in three application scenarios: 1) protecting sensitive data for memory corruption defenses, such as CFI; 2) creating an isolated execution environment (IEE); 3) protecting JITed code in JIT compilers. In the first scenario, the sensitive data is stored in the isolated memory region, which is only accessible by the CFI-instrumented code with LSU instructions. In the second case, with an IEE created by PANIC, the data inside the IEE cannot be accessed by the code outside due to PAN’s restriction, but is accessible by
the code inside by using LSU instructions. In the third setting, to protect JITed code, the original code cache is used to execute the code with the executable permission, and a shared memory region is set to be user pages with the writable permission which is used to emit the code in JIT compilers by using LSU instructions.

A prototype of PANIC is implemented on the Linux/AArch64 platform. We demonstrated the application of PANIC on the protection of CFI shadow stack, session keys of the OpenSSL library, and, the JITed code of JavaScriptCore. To evaluate the performance overhead, we use the SPEC CPU 2017 benchmarks, the Nginx web server, and the widely used JavaScript engine JavaScriptCore as our protection targets. The experimental results show that PANIC incurs very low performance overhead, and performs better than existing protections.

In general, the contributions of this paper are as follows:

- A generic and efficient intra-process memory isolation mechanism on AArch64. We propose a new intra-process memory isolation mechanism, PANIC, by using the PAN hardware feature and the LSU instructions on ARM processors and demonstrate how PANIC can be leveraged to build generic memory isolation with three case studies. Specifically, PANIC is the first to apply UAO for address space isolation. By leveraging UAO, PANIC enables/disables the LSU instructions to achieve bidirectional isolation, which prevents attackers from exploiting gadgets containing LSU to compromise the isolation.

- New techniques for securely running the user code of AArch64 in kernel mode. We propose a novel shim-based memory isolation method to prevent user code running in kernel mode from accessing kernel memory and a new sensitive instruction emulation technique that prevents attackers from abusing sensitive instructions.

- New insights from implementation and evaluation. We implement and evaluate PANIC on Apple M1 processor. The empirical results show that PANIC incurs very low performance overhead and outperforms the existing isolation approaches.

2 BACKGROUND

2.1 Exception Levels and Their Movement

The name for privilege in AArch64 is Exception level, often abbreviated to EL. AArch64 supports multiple levels of privilege, ranging from EL0 to EL3. The higher the level of privilege, the higher the number. The lowest level, EL0, is also called the unprivileged level which is used to run applications, while the other levels are privileged levels that are used to run more privileged software, such as OS runs at EL1. Exceptions are divided into two major categories: synchronous exceptions and asynchronous exceptions. Asynchronous exceptions consist of IRQ, FIQ, and SError.

When an exception occurs, CPU saves both the running status into Saved Program Status Register (SPSR_EL1) and the exception address into Exception Link Register (ELR_EL1). SPSR_EL1 stores the EL and the stack pointer register at the time of the exception. Each EL has two optional stack pointers, SP_EL1 and SP_EL0. The stack pointer in use is indicated by SPSel register. Note that the user code runs in EL0 is forced to use SP_EL0, and SP_EL1 is forced to be used when taking an exception.

Execution can move between ELs only on taking an exception, or on returning from an exception. On taking (or returning from) an exception, the EL either increases (or decreases) or remains the same. When an exception occurs, the processor must execute handler code that corresponds to the exception whose location in memory is called the exception vector. The exception vector is a vector table composed of sixteen 0x80-length bytes, and its base address is stored in the Vector Base Address Register (VBAR_EL1). The selection of the exception entry is related to the type of exception, the stack register in use, and the state of the register file at the time of the exception. ERET, ERETAA or ERETAB instructions can produce an exception return, which restores both the running status from SPSR_EL1 and the PC from ELR_EL1.

2.2 Unprivileged Load/Store on AArch64

The access right of a memory access instruction is determined by the current EL and the access permission set in the target memory attribute, except for the load/store unprivileged (LSU) instruction, i.e., LDTR and STTR. The LSU instructions perform memory operations at EL0 in the same way as normal load and store instructions. However, when they are executed at EL1, they are treated as if they are executed at EL0 rather than EL1.

Access Permission. The access permission of each memory page is determined by the 2-bit AP[2:1] field of the translation table entry. The AP field has four possible values, each representing the read and write permissions of the page at EL0 and at higher Exception Levels. Linux kernel treats AP[2] as an R/RW indicator and AP[1] as a privilege indicator. When AP[1] is set to be 1, the corresponding memory page is the unprivileged page (abbreviated as U-Page); otherwise, it is the privileged page (abbreviated as P-Page). The user space is usually set to be U-Pages and the kernel space is set to be P-Pages. The EL-based memory isolation is enforced by the hardware that the code runs at EL0 cannot access P-Page, i.e., the user code cannot access the kernel space.

Privileged Access Never (PAN). PAN is a hardware feature that is used to prevent privileged code from accessing unprivileged pages. That is, the normal load and store instructions in the kernel cannot access the user page when PAN is enabled. The kernel uses the PAN-based memory isolation to prevent the return-to-user attacks [30]. However, LSU instructions are not constrained by PAN due to the CPU always treats them as executed at EL0. The kernel usually uses them to access the user data legally.

Unprivileged Access Override control (UAO). UAO is a hardware feature that allows LSU instructions to behave as standard load/store instructions. When UAO is enabled, LSU instructions use the current EL rather than the forced EL0.

2.3 Address Translation Management

To perform virtual-to-physical address translation, AArch64 provides two Translation Table Base Registers, TTBR0_EL1 and TTBR1_EL1, to hold the base addresses of user and kernel page tables. TTBR0_EL1 is used to translate the bottom half of the virtual address space corresponding to the user space, and TTBR1_EL1 is used to translate the top half of the virtual address space corresponding to the kernel space. Translation Control Register (TCR_EL1) is a control register
that determines various features related to address translation, such as translation granule size.

Translation Lookaside Buffer (TLB) is a hardware component to reduce the overhead of address translation by caching recently used virtual-to-physical address mappings. But TLB still needs to be flushed for every switch of virtual address space. To address this, the Address Space Identifier (ASID) is introduced that tags each TLB entry with an identifier. On AArch64, each of TTBR0_EL1 and TTBR1_EL1 contains an ASID field, and the A1 field in TCR_EL1 selects which ASID to become the current ASID. AArch64 also presents the non-Global (nG) flag in translation table entries. By clearing the flag, the corresponding pages come to be seen from every task regardless of their ASID values. TCR_EL1 also has two fields, EPD0 and EPD1, to enable and disable translation table walk when using TTBR0_EL1 and TTBR1_EL1, respectively.

3 OVERVIEW

3.1 Core Idea

The high-level idea of PANIC is shown in Fig. 1. The protected process runs at EL1 (i.e., the kernel mode) instead of EL0 (i.e., the user mode). The regular memory regions, such as stack and heap, are set to be P-Pages; the isolated memory region is set to be U-Pages. The normal load/store instructions used in the untrusted code can still access the regular memory region, but they are not allowed to access the isolated memory region due to PAN’s protection. The trusted code is allowed to access the isolated memory region by using the LSU instructions.

Compared with existing address-based and domain-based isolation mechanisms, PANIC has two significant advantages:

- **Accesses to isolated memory regions by untrusted code are restricted by default.** The address-based method needs to instrument all load/store instructions in untrusted code to restrict their access targets, which could cause serious code bloat and poor performance. In PANIC, untrusted code needs not to be modified at all, because its accesses to isolated memory regions are restricted by default.

- **Different instructions are used to access regular and isolated memory regions.** The domain-based method needs to enable access to the isolated memory region before accessing it, and disable access when it is finished. The frequent permission switching usually incurs high performance overhead. But, such switching is not needed in PANIC due to it uses different instructions to access the regular and isolated memory regions.

3.2 Threat Model

PANIC is designed to be a generic mechanism for intra-process isolation. The protected programs can be either server programs (e.g., Nginx web server) or client programs (e.g., browsers). We assume the system software, e.g., the operating system and the hypervisor, and the hardware are secure and trustworthy. We assume the kernel enforces standard DEP—an executable page must not be simultaneously mapped with write permissions.

We specifically consider three scenarios where PANIC can be useful. Each has different assumptions of the adversary’s capabilities. However, in all three scenarios, we assume the protected programs may contain memory corruption vulnerabilities and the adversary can exploit the vulnerabilities to obtain arbitrary memory read/write primitives.

**Scenario-1: Hardening CFI.** We assume that memory corruption defenses, such as CFI [7, 44–46, 58], are deployed to prevent the adversary from hijacking the control flow after exploiting the memory corruption vulnerabilities. PANIC can be used to isolate the shadow stack of CFI from the adversary. In this scenario, we assume that the adversary cannot bypass CFI to alter the control flow. In other words, CFI and PANIC protect each other. In a similar manner, PANIC can also be used to harden other memory corruption defenses such as protecting the random tags used in PACTight [26] which is an implementation of CPI [33].

**Scenario-2: Creating IEE.** IEE is used to protect the complete execution of the isolated component that stores all sensitive data and code, thereby preventing the effect of the vulnerabilities in outside components on the isolated component’s data. In this scenario, we assume the adversary is capable of accessing arbitrary memory, but also hijacking the control flow of the outside components by exploiting the vulnerabilities. The goal of the attack is to leak or corrupt the IEE data from the outside. However, we assume the code inside the IEE created by PANIC is secure.

**Scenario-3: Hardening JIT.** The protected program has integrated a just-in-time (JIT) compiler. The attack target of adversaries is to break the integrity of the code cache and inject shellcode therein, i.e., attackers use the arbitrary write primitive to modify the code cache directly. PANIC can be used to protect the code cache against such attacks. In such a scenario, PANIC shares the same assumptions with other isolation works, such as libmpk [48] and ERIM [51].

3.3 Key Challenges

We outline the following design challenges of PANIC.

C-1: Preventing accesses to the kernel. Once the protected process runs in kernel mode, the EL-based isolation mechanism can no longer prevent the protected process from accessing the kernel. The PAN-based isolation mechanism cannot be used to protect the kernel either, as it is already used in PANIC. That is, the regular memory region can be accessed by the kernel code without any restriction due to its P-Page settings. Existing methods that switching page tables [5] or adjusting the virtual address range [15] either rely on virtualization [15] or can not disable access of the whole user/kernel space [10]. Therefore, a new method needs to be introduced to both protect the kernel memory and prevent the kernel code from accessing the untrusted user memory.

C-2: Preventing abuses of sensitive instructions. When the process runs in kernel mode, privileged instructions, which can not be executed in user mode, become available. Attackers may abuse...
privileged instructions to launch more powerful attacks. However, simply forbidding all privileged instructions is not enough. We found that although some instructions are not privileged instructions, their execution has a security impact on the system. They can be executed both in user mode and kernel mode. For example, the `call` instruction is equivalent to `nop` when executed in user mode but will cause the CPU to enter a low power state when executed in kernel mode. That is, privileged instructions are only a subset of sensitive instructions. Therefore, how to define, identify and handle all sensitive instructions is challenging.

C-3: Providing more generic isolation capabilities. A straightforward use of PANIC is to store sensitive data in isolated memory regions and permit access to them with only LSU instructions. Therefore, it is straightforward to use PANIC to harden CFI (scenario-1 in §3.2). But applying PANIC in scenario-2 and scenario-3 is more challenging. For scenario-2, PANIC can prevent outside code (run in kernel mode) from directly accessing data in IEE (stored in U-Pages), but if the outside code can perform arbitrary code execution, it may leverage the code gadget inside the IEE to corrupt the IEE data. For Scenario-3, an intuitive way to use PANIC is to configure the code cache to be U-Pages with the writable and executable permissions, and permit the JIT compiler to emit code into it with LSU instructions. But at privileged levels, the hardware enforces the DEP protection on U-Page, resulting in any U-Page cannot have writable and executable permissions at the same time.

We will next introduce the overview of PANIC in §3.4 and detail how we address each of these three challenges in §4, §5, and §6, respectively.

3.4 Architecture Overview

An overview of PANIC’s architecture is shown in Fig. 2. The core of PANIC is a kernel module that supports running user processes securely in kernel mode. It identifies the target process that uses PANIC at process startup and places it into kernel mode to run. It includes three key components: memory management, runtime shim and instruction scanner.

The runtime shim component is used to perform bidirectional memory isolation between the kernel-mode process and the kernel — preventing the untrusted user from accessing the kernel memory and protecting the kernel from accessing the untrusted user memory. It is implemented as a transparent trampoline that interposes all control transfers between the user and the kernel. It consists of two parts: the user shim and the kernel shim. The user shim is mapped into the address space of protected processes at load time to mediate all communication with the kernel and to enable/disable access to the kernel memory when needed. The kernel shim cooperates with the user shim to transfer the control flow and enables/disables access to user memory space as needed.

The instruction scanner component scans each code page of the protected program at runtime, before they are mapped to be executable, and looks for sensitive instructions that may affect the entire kernel. Once a sensitive instruction is identified, it will be transformed in place to an instruction that can raise exceptions, which can be captured by PANIC and emulated to produce the same effect as executed in user mode.

The memory management component is used to configure the regular and isolated memory regions, track the lifetime of the code pages, and lock the isolated memory region and the user shim. The regular memory region is configured to be P-Pages, and the isolated memory region is configured to be U-Pages. When a code page is about to appear in user space, it will notify the instruction scanner component to scan sensitive instructions. Since the isolated memory region and the user shim are only accessed (i.e., read, write and execute) through instructions, any operations to them through system calls are not allowed.

4 SHIM-BASED MEMORY ISOLATION

In this section, we introduce how PANIC achieve bidirectional isolation between the protected process and the kernel.

4.1 Shim Design

As mentioned before, existing memory isolation works in kernel [5, 15] cannot be used to achieve bidirectional isolation, as they either rely on virtualization or cannot disable access of the whole user/kernel space. To address this, we take advantage of the separated user page tables and kernel page tables on ARM, and enable and disable the user/kernel page table walk to switch access permission of kernel/user space by using EPD0 and EPD1.

Therefore, shown in Fig. 3, we design a shim which is a sequence of instructions that consists of two parts: the user shim and the kernel shim. The user shim is mapped into the user space at load time to interpose all communication with the kernel. This is achieved by setting VBAR_EL1 register to point to the user shim in user. Exceptions are redirected to the user shim, it first exposes the kernel space by setting the EPD1 field of TCR_EL1 register, and then jumps to the kernel shim. The kernel shim conceals the user space by setting
the EPD0 field of TCR_EL1 register. During exception returns, the operations are opposite. 

But the above method cannot prevent address translation if the virtual-to-physical mapping is already cached in TLB, that is EPD0 and EPD1 are only effective when TLB is missed. Therefore, we use ASID to isolate TLB entries that are used by the protected process and the kernel. As shown in Fig. 3, PANIC assigns a unique ASID (i.e., the ASIDu and the ASIDk) to the protected process and the kernel, respectively, and stores them into TTBR0_EL1 and TTBR1_EL1 registers. When jumping to the kernel, the user shim also needs to switch the current ASID to the ASIDk by setting the A1 field of TCR_EL1 register; when jumping back, the user shim switches back to the ASIDu.

Note that all kernel memory pages are set to be non-Global pages to ensure the isolation in TLB entries and other processes use only one ASID as usual. The user shim can be accessed by both the kernel and the user due to its virtual-to-physical mappings being both cached in user and kernel TLB entries. But it is secure because the user shim is trusted and protected by the PANIC module to ensure that it cannot be corrupted.

4.2 Self Protection of User Shim

Except for sensitive instructions used in the user shim, such as those configuring TCR_EL1 register, all sensitive instructions in user code pages are disallowed (see §5). Following the principle of least privileges, the user shim has been designed as small as possible. Moreover, as the user shim is not isolated from the user code, it must ensure two properties to guarantee its security:

- **Atomicity**: The execution of the user shim can only start from the entrance through exceptions. For example, jumping to the middle of this sequence is not allowed;
- **Determinacy**: The execution sequence is deterministic in the sense that it has the same behavior regardless of the current environment, including all registers and the user memory.

4.2.1 Enforcing Atomicity and Determinacy. To ensure the first property, PANIC performs exception context checking to ensure that the user shim can only be executed from the entry point through exceptions, jumping to the entrance or middle of the instruction sequence cannot pass the check. To avoid unrecoverable destructive effects [28] caused by hijacked execution of sensitive instructions before the checking function, PANIC use debug registers to set breakpoints at sensitive instructions, preventing their execution entirely. Hence, abusing sensitive instructions will trigger breakpoint exceptions; To ensure the second property, the sequence of instructions in the user shim is carefully designed so that they do not access the user space memory, and registers must be defined before being used.

Listing 1 gives the complete instruction sequence used in the user shim. This instruction sequence is stored only on one code page. It consists of two parts: user_shim_vectors (lines 37-42) and user_shim_exit (lines 44-45). The user_shim_vectors are configured as the exception entries by setting VBAR_EL1 register. Although there are 16 entries (vectors) for the user_shim_vectors, only the first four are valid due to running in kernel mode. Each entry code is expanded by the macro function entry Routine. The user_shim_exit is the entry when switching back from the kernel shim. The code of user_shim_exit is expanded by the macro function exit Routine.

In Listing 1, the lines without a background color contain functional instructions, while the lines with a gray background are security-related instructions. Five debug registers are reserved for the protection of instructions that set TCR_EL1 register. Since TPIDR_EL0 register is used to be the temporary register in the whole system of Linux, it is used to store the value of the scratch register X30 in PANIC. The instruction sequences in entry Routine and exit Routine are similar. They work as follows:

- **Step-1**: Clear the debug register to enable the execution of TCR_EL1 register setting instruction (lines 5-6 and lines 21-22).
- **Step-2**: Set TCR_EL1 register to enable (lines 7-9) and disable (lines 23-25) the access to the kernel space.

```
  .macro entryRoutine, n:req, t:req  # n: num of debug register
  .align 7
  1: mrs x30, tcr_el1  # 2) get current tcr_el1
  2: movk x30, 0x7f50, 1sl 16 # 3) get current tcr_el1
  3: mrs x30, tcr_el1  # 4) get the address of label 1
  4: isb  # instruction barrier
  5: mrs x30, tcr_el1  # 5) get current tcr_el1
  6: movk x30, 0x7f50, 1sl 16 # 6) load kernel_shim's location
  7: ret  # jump to kernel shim
  .endm
  .macro exitRoutine, n:req  # n: num of debug register
  1: mrs x30, tcr_el1 # 1) clear tcr_el1 (label 3)
  2: movk x30, 0x7f50, 1sl 16 # 2) set breakpoint at label 1
  3: mrs x30, tcr_el1, x30 # 3) toggle fields: EPD0 = 0, A1 = 0
  4: mrs x30, tcr_el1  # 4) get the address of label 3
  5: isb  # instruction barrier
  6: mrs x30, tcr_el1  # 5) set breakpoint at label 3
  7: movk x30, 0x7f50, 1sl 16 # 6) clear tpidro_el0
  8: ret  # jump to kernel shim
  .endm
```

Listing 1: The entry and exit routines in the user shim.
• **Step-3**: Reset the debug register to disable the execution of TCR_EL1 register setting instruction (lines 10-12 and lines 26-28).

• **Step-4**: Check SPSEl register to ensure that currently running under the exception context (lines 13-15 and lines 29-31). SPSEl could reveal the exception context because the protected process uses SP_EL0 as its stack pointer and SP_EL1 is forced to be used when taking exceptions.

• **Step-5**: Leave the user shim and then enter the kernel shim (lines 16-17) or return from the exception (line 34).

Note that although the user shim uses a total of 5 debug registers, we could reduce the number of registers to 2 by sharing most of the code in the first four *entry_routines*.

### 4.2.2 Security Analysis

The code integrity of the user shim can be ensured by the memory management component. The only attack surface is the control-flow hijacking that jumps to an arbitrary location in the instruction sequence. The user shim is designed to thwart such attacks. To illustrate the defense design clearly, we enumerate all situations in Table 1. Since abusing sensitive instructions used in the instruction sequence is the attackers’ goal, we list them by line in the table, and other consecutive non-sensitive instructions occupy a separate row. Note that the MRS instruction that reads TPIDRRO_EL0 register (line 32) is not a sensitive instruction, it can be used in user mode. Jumping to the different locations of the instruction sequence can achieve different attack intentions. For example, when the jump target is line 5 in Listing 1, attackers can clear the breakpoint at line 9, and their intention is to corrupt TCR_EL1 after disabling the execution protection at line 9. In sum, the intentions can be summarized into five types:

- For the attack intentions ①, ③ and ④ in Table 1, the user shim will trigger the breakpoint instruction exception (line 15 and line 31 in Listing 1) due to the exception context checking;
- For the attack intention ② in Table 1, the user shim will trigger the breakpoint exception (line 9 and line 25 in Listing 1) due to the protection of debug register;
- For the attack intention ⑤, the user shim will trigger the breakpoint instruction exception when jumping to BRK instruction, or trigger the translation fault (a synchronous exception) when jumping to the instructions that enter the kernel shim (line 16 in Listing 1) due to the kernel space is unaccessible, or rebound to execute the user instructions when jumping to use RET/ERET instructions. Note that ELR_ELx and SPSEl used in ERET instruction are privileged registers that only record the context of the most recent exception triggered in user space. Executing ERET instruction will only return to the location of the last user exception without any security issues;
- Any exceptions triggered in the user shim will enter into the *entry_routine* and jump to the kernel shim, its behavior is determined and cannot be manipulated. The kernel shim will check the exception address, and if the exception occurs in the user shim, the target process will be killed immediately. This is because the legitimate execution of the user shim must come from exceptions, and interrupts are disabled by the CPU when taking on exceptions. Since attackers cannot disable interrupts when executing the code

#### Table 1: Our defense against the control-flow hijacking attacks.

<table>
<thead>
<tr>
<th>Jump Target</th>
<th>Lines in Listing 1</th>
<th>Attack Intention</th>
<th>Defensive Act.</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>1-3</td>
<td>Enable access to</td>
<td>A1</td>
</tr>
<tr>
<td>write_tpidro</td>
<td></td>
<td>the kernel space</td>
<td></td>
</tr>
<tr>
<td>write_dbgbvr</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read_tcr</td>
<td>6</td>
<td></td>
<td>A2</td>
</tr>
<tr>
<td>...</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write_tcr</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write_dbgbvr</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>11</td>
<td></td>
<td>A1</td>
</tr>
<tr>
<td>read_spsel</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>13</td>
<td></td>
<td>A1</td>
</tr>
<tr>
<td>write_tpidro</td>
<td>14-17</td>
<td></td>
<td>A1/A3/A1/A4</td>
</tr>
<tr>
<td>eret</td>
<td>18</td>
<td></td>
<td>A4</td>
</tr>
<tr>
<td>...</td>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>30-32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>34</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**5 SENSITIVE INSTRUCTIONS EMULATION**

In this section, we will detail how sensitive instructions are identified and handled.

#### 5.1 Definition of Sensitive Instructions

Sensitive instructions include both privileged and unprivileged instructions, as there are many unprivileged instructions whose behaviors are different when running in different modes. The adversary may exploit this difference to attack the system. Therefore, we give a unified definition of sensitive instructions from the perspective of instruction behavior here.

**Definition 5.1.** For an instruction $i$, we define its Instruction Behavior $b^m_{ic}$ as the changes in machine state that will be triggered after the instruction $i$ is executed in the mode $m$ with system configuration $c$, where $m \in \{\text{kernel}, \text{user}\}$ and $c \in C$, where $C$ is the power set of all possible permutations of values that system control registers can take. When an instruction’s behavior is UNDEFINED, executing the instruction will raise an Undefined Instruction Exception.

**Definition 5.2.** An instruction is treated as a Sensitive Instruction if and only if it satisfies (1), i.e., *under certain system configurations, its behaviors in different modes are inconsistent*.

\[ \exists c \in C, b^m_{uc} \neq b^m_{ck} \quad (1) \]

**Definition 5.3.** An instruction is an Unconditionally Sensitive Instruction when it satisfies (1) and (2), i.e., *under certain system configurations, its behaviors in different modes are inconsistent, and the behavior in user mode is always UNDEFINED*.

\[ \forall c \in C, b^m_{uc} = \text{UNDEFINED} \quad (2) \]

**Definition 5.4.** An instruction is a Conditionally Sensitive Instruction when it satisfies (1) and (3), i.e., *under certain system configurations, its behaviors in different modes are inconsistent, and neither is UNDEFINED*.

\[ \exists c \in C, b^m_{uc} \neq \text{UNDEFINED} \land b^m_{ck} \neq \text{UNDEFINED} \quad (3) \]
Apart from Definition 5.3 and Definition 5.4, there is another situation of sensitive instructions which is satisfied $∀c ∈ C, k_{kernel} = \text{UNDEFINED}$ $∧ ∃c ∈ C, k_{user} ≠ \text{UNDEFINED}$. However, this situation does not exist in ARMv8-A instructions. We dedicated 8 person-months to meticulously reviewing the 1,529 instructions in the ARMv8-A Architecture Reference Manual [2]. Through careful review and experimentation, we were able to document the behavior of each instruction under different modes (i.e., kernel mode and user mode). We identified 874 sensitive instructions, including 530 unconditionally sensitive instructions and 344 conditionally sensitive instructions. Fig. 4 gives two examples of different types of sensitive instructions. Cache maintenance instruction is an unconditionally sensitive instruction (Fig. 4(a)), while the system registers writing instructions MSR are conditionally sensitive instructions (Fig. 4(b)).

### 5.2 Handling Sensitive Instructions

As shown in Fig. 5, PANIC performs an on-the-fly binary inspection to identify sensitive instructions. PANIC screens sensitive instructions in each new code page (3) and transforms them to instructions whose execution can raise exceptions (2), captures their execution at runtime and then emulates their behavior in user mode (4). In detail, PANIC filters each 4-byte instruction encoding in a soon-to-be-mapped code page to determine if it is an unconditionally or conditionally sensitive instruction. For an unconditionally sensitive instruction, PANIC transforms it in-place to an UDF instruction whose execution can raise an undefined instruction exception; for a conditionally sensitive instruction, PANIC transforms it to a BRK instruction whose execution can raise a breakpoint instruction exception. As shown in Fig. 5 (3), when an unconditionally sensitive instruction is identified, PANIC appends it to an array map and transforms it to BRK instruction whose immediate operand 1 is the current index of the array map. As the value range of the BRK operand is from 0 to 0xFFFF, the size of the array map is set to 0x10000. When the upper limit of the array map is exceeded, the subsequent instruction will be transformed to BRK instruction with the same operand 0xFFFF. The original instruction will be recorded into a hash map which is indexed by using its address.

PANIC emulates both conditionally and unconditionally sensitive instruction once these exceptions are captured. For unconditionally sensitive instructions, it injects an undefined instruction exception. For conditionally sensitive instructions, PANIC locates the original instruction by indexing the array map using ESR_EL1 register which encodes the value of BRK’s operand, and emulates the behaviors of the original instruction accordingly.

### 5.3 Optimization

We further perform the following optimizations to reduce performance penalties introduced by sensitive instruction handling.

#### 5.3.1 Reducing Transformation

In fact, the number of conditionally sensitive instructions we need to capture is much less than the 344 mentioned above. This is because we observed that for most default system configurations, many conditionally sensitive instructions behave the same in different modes. Hence, there is no need to capture their execution under the specific system configuration. For example, as shown in Fig. 4 (b), when system configuration $c'$ is satisfied $Cond.4 ∧ Cond.6$, its behaviors are the same in both kernel and user modes. But we should note that if the system configuration changes, these instructions’ behaviors may be not the same at different modes and they should be captured again. To achieve this, PANIC checks whether the system configuration has changed when the control flow returns to the user. If it is, PANIC updates the classification strategy, restores instructions in the array map and the hash map, and re-scans all code pages.

#### 5.3.2 Fast Encoding Filtering

We design a hash-based filtering method to quickly determine if a given instruction encoding is an unconditionally or conditionally sensitive instruction. We found that the encodings of many system instructions, including those accessing system registers/special-purpose registers, performing cache/TLB maintenance and address translation, etc., are continuous [9]. To avoid the handling of hash collisions, we use two separate hash tables to store non-sensitive instructions which are the system instructions and sensitive instructions that are not the system instructions, respectively. PANIC first determines whether an instruction is a system instruction based on the bits 19-31 of the instruction’s encoding, and then searches in the corresponding hash table according to the instruction type.

### Table 2: PANIC APIs.

<table>
<thead>
<tr>
<th>APIs</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>int panic_read(void *dst, void *src, len)</td>
<td>Read data from the isolated region</td>
</tr>
<tr>
<td>int panic_write(void *dst, void *src, len)</td>
<td>Write data to the isolated region</td>
</tr>
<tr>
<td>int panic_emit_code(void *dptr, void *src, len)</td>
<td>Write code to the isolated region</td>
</tr>
<tr>
<td>int panic_register_entry(func_name, ret_type, arg_type, ...)</td>
<td>Register an exit function of IEE</td>
</tr>
<tr>
<td>int panic_register_exit(func_name, ret_type, arg_type, ...)</td>
<td>Register an exit function of IEE</td>
</tr>
<tr>
<td>int panic_alloc_region(int len, int prot, void **cptr)</td>
<td>Allocate an isolated mem. region</td>
</tr>
<tr>
<td>int panic_free_region(void *dptr, void *cptr, int len)</td>
<td>Free an isolated region</td>
</tr>
<tr>
<td>int panic_copy_to_untrust(void *to, void *from, int len)</td>
<td>Copy data from outside memory</td>
</tr>
<tr>
<td>int panic_copy_to_trust(void *to, void *from, int len)</td>
<td>Copy data into outside memory</td>
</tr>
</tbody>
</table>
6 PANIC-BASED MEMORY ISOLATION

PANIC can be used to construct different isolation scenarios. In this section, we outline three scenarios, i.e., protecting CFI (see §6.2), creating isolated execution environments (§6.3), and hardening JITed code (§6.4). We will first outline PANIC APIs with which applications can be built to make use of PANIC (see §6.1).

6.1 PANIC APIs

As shown in Table 2, a set of APIs is provided for users. The APIs are divided into three categories. The first category consists of APIs for managing isolated regions. Users can use the panic_alloc_region() and the panic_free_region() to allocate and de-allocate an isolated memory region. If the executable and writable permissions are specified simultaneously in the argument prot, the return value of the panic_alloc_region() will point to a writable isolated memory region, and the argument cptr will point to the shared regular memory region with the executable permission. The second category consists of APIs for accessing isolated memory regions. Users could read and write isolated memory regions by using the panic_read() and the panic_write(). The panic_emit_code() is used to emit code to isolated memory regions, it will inspect the sensitive instructions in a secure way (see §6.4). The third category of APIs is designed to support IEE. The panic_register_entry() and the panic_register_exit() are used to register the valid entry and exit functions of IEE, and they are only hints for our compilers. The #pragma panic_lee is a hint, which can be placed at the beginning of source files, to specify the file whose code and data need to be placed into the IEE. Two API functions, the panic_copy_from_untrust() and the panic_copy_to_untrust(), are used in IEE to exchange the data between the IEE memory and the outside memory.

Users should adopt our compiler and link the PANIC library to use these APIs. The compiler is implemented using the LLVM framework. It has two main tasks: 1) separating data in the code segment; 2) emitting the IEE code to only use LSU memory access instructions. Separating data in the code segment is essential, because the mixed data may be misidentified as sensitive instructions and the transformation may affect their normal execution. This operation should be performed not only on the protected application, but also on its dependent libraries. Besides handling programs by using our compiler, we also provide a binary rewrite tool to handle binary programs without source code (see Appendix A).

6.2 Hardening CFI Defenses

As shown in Fig. 6 (a), memory-corruption defenses like CFI could protect their metadata, such as the shadow stack, by storing it in the isolated memory region. The normal memory access instructions used in applications cannot access the isolated memory region due to PAN’s protection, while the CFI defense code could still access it by using LSU instructions. LSU instructions cannot exist in application code; they must be transformed to normal load/store instructions. It is simple that they need only flip the 11th bit of the LSU instruction’s encoding.

6.3 Creating Isolated Execution Environment

As shown in Fig. 6 (b), PANIC creates an isolated execution environment (IEE) for holding trusted code and data. PANIC ensures that only the code in IEE can access the data in IEE. All data used in IEE is configured to use U-Pages, including the global data, stack, and heap. During compilation, our compiler identifies and emits all global data into a specific data section. Note that the compiler optimization that promotes the data into the instruction’s immediate operands is disabled when compiling the IEE code. When loading the program, the PANIC module identifies this data section and set it to be U-Pages. Meanwhile, the PANIC module allocates an isolation memory region as the stack of IEE and notify the IEE code by embedding the stack’s location into its data. For the heap, the IEE code must allocate the dynamic memory region by calling the allocation API in PANIC. All IEE code is emitted to only use LSU instructions that ensure the code can access the IEE data.

UAO-based gate. To prevent the outside code from using LSU instructions directly or abuse the LSU instructions in the IEE code indirectly, we propose a UAO-based gate method for IEE isolation (shown in Fig. 7). The entry gate is used to disable UAO, which enables the access of the IEE code to the IEE data, and pivots the stack to use the IEE stack. The exit gate performs the opposite operations. Once the IEE exits, LSU instructions cannot be used to access the IEE data because UAO is enabled again. Hence, only the control flow entering from the entry can access the IEE data. To ensure all control flows out of the IEE must go through the exit gate, the coarse-grained control flow integrity mechanism is enforced on the IEE code. In PANIC, the UAO setting instruction (i.e., MSR UAO, #1mm) is not allowed to be used in the outside code pages by preventing its occurrence when screening code pages. Hence, the outside code cannot set UAO unless invoking the IEE gates.

UAO-based scheme versus PAN-based scheme. It should be noted that switching PANs can achieve a similar but not identical effect. Switching PAN can control the access of LS instructions to isolated regions (U-pages). However, unlike this PAN-based scheme, the UAO-based scheme achieves bi-directional isolation based on the property that LSU instructions can only access U-pages when UAO is turned off, and LSU instructions can only access S-pages when UAO is turned on, preventing possible attack surfaces brought about by accessing memory across isolation boundaries by the access instructions. In contrast, the LS instructions used by the PAN-based scheme to access isolated memory do not distinguish whether the accessed memory is U-pages or not.

Users could specify the entry functions of IEE, which are the valid entry points of an IEE. They are only the entries that can be
called from the outside. Similarly, users could also specify the *exit functions* in the outside, which will be called by the IEE. Fig. 7 gives an example of IEE. Entry_A is an entry function, and Exit_C is an exit function. The entry gate and exit gate are instrumented in the entry and exit points of Entry_A, respectively. When the control flow is transferred from Func_B to Exit_C, the exit gate and entry gate need to be inserted before and after the call sites.

### 6.4 Protecting JITed Code

The hardware enforces the DEP protection on U-Pages in kernel mode. Hence, when using PANIC to protect JITed code, as the JIT compiler now runs in kernel mode, the code cache is DEP-protected. Therefore, the generation and execution of the JITed code need to change the permission of the code cache which will incur high overhead. To address this, we propose a *shared memory-based execution and write separation* method (shown in Fig. 6 (c)). Two shared memory regions are allocated, one is the regular memory region with readable and executable permissions for executing the JITed code, and the other one is the isolated memory region with readable and writable permissions for emitting the JITed code.

**Ensuring the security of JITed code.** As mentioned before, sensitive instructions may be abused to corrupt the system. The PANIC module does not trust the user code and performs the binary inspection for each code page at runtime. The JITed code also needs to be screened, but using the PANIC module to perform such screening is not an efficient way as every screening request causes the context switch. To address this, we propose to offload such screening into the user context which is inspired by the vDSO in Linux: PANIC maps a PANIC-vDSO library into the user space along with the writable code cache. Only this library can write the code to be issued into the code cache, it will inspect every issued instruction, just like the PANIC module. This library is trusted and protected by the PANIC module. To prevent arbitrary execution of the library code from writing sensitive instructions into the code cache, we use the aforementioned IEE technique (Fig. 6) to hold the whole execution of the PANIC-vDSO. During the execution, if there are breakpoint instruction exceptions coming from the code cache, the PANIC module emulates the execution of sensitive instructions by using the record table in this library. When the system configuration changes, the record table in the library will be updated accordingly. Note that the PANIC module enforces that the UAO setting instructions can only be used in the PANIC-vDSO.

### 7 IMPLEMENTATION

PANIC is designed and implemented on the Linux/AArch64 platform. The PANIC kernel module is implemented with 2,362 lines of C code and 422 lines of assembly code. The PANIC library, which offers APIs, contains 130 lines of C code. And the PANIC-vDSO contains 133 lines of C code. Our compiler is implemented based on LLVM by adding several back-end passes. One back-end pass, used to separate code and data, contains 120 lines of C++ code, and the other passes are related to IEE support which contains 787 lines of C++ code. The binary rewrite tool that separates code and data contains 743 lines of Python code.

In the following paragraphs, we will introduce the implementation details of specific protection scenarios.

**Protecting the shadow stack.** Existing defenses [36, 37] mainly protect return addresses by using PA on AArch64, but they suffer from the pointer reuse attacks [26]. Shadow stack is a powerful method to protect the return addresses by storing them in a separate memory region. Ensuring the integrity of the shadow stack on AArch64 is hard due to the lack of an efficient memory isolation mechanism. PANIC can be used to address this problem by placing the shadow stack into an isolated memory region. Users could use the panic_alloc_region() to allocate an isolated memory region for holding the shadow stack. Users could call the panic_write() to store the return address into the shadow stack after calling a function and then call the panic_read() to restore the return address before the function returns.

**Protecting the session keys in OpenSSL.** OpenSSL is a toolkit for general-purpose cryptography and secure communication that is widely used in web servers. To protect the session keys in the OpenSSL library, we use the method mentioned in CryptoMPK [29] to identify all operations related to the AES session keys, including all functions and relevant data, and move them to a dedicated file which will be compiled into the IEE by using the #pragma panic_iee in the file header. Then, the panic_register_entry() and the panic_register_exit() are used to register the entry functions and exit functions of the IEE. For the memory objects that are accessed by both the IEE code and the outside code, we duplicate them and use two APIs, the panic_copy_from_untrust() and the panic_copy_to_untrust(), to perform the synchronization.

**Protecting the JITed code in JavaScriptCore.** JavaScriptCore (JSC) is a JavaScript engine utilized in the widely-used browser, Safari. On the Linux/AArch64, the code cache in it is configured to have both the executable and the writable permissions by default that can be attacked by injecting the malicious code directly. We deploy PANIC to protect the code cache of the JSC. The JSC is modified to use the panic_alloc_region() with the execution permission in the argument prot to allocate an isolated memory region for the writable code cache and a regular memory region for the executable code cache, and then use the panic_emit_code() to emit the code. The panic_emit_code() in the PANIC-vDSO will inspect the instructions to be emitted and then write the transformed instructions into the isolated code cache.

### 8 EVALUATION

In §4 and §5, we identified and resolved security threats of running user code in kernel mode. Therefore, by design, PANIC does not introduce new security issues. So in this section, we focus on the performance evaluation of PANIC. We implemented PANIC on
Linux kernel v5.19.4 that runs on a Mac mini with an 8-core M1 CPU with PA supported and 16 GB RAM. The Apple M1 processor is equipped with four performance cores and four efficiency cores, and all experiments were conducted on performance cores. All benchmarks used in our experiments were compiled using the Clang v14.0.1 at the O2 optimization level.

### 8.1 Evaluating Basic Operations

As mentioned before, PANIC can be used to provide generic intra-process memory isolation. The PANIC-based memory isolation mainly uses two basic operations, i.e., LSU and UAO. The performance of the LSU instruction is crucial to the performance of all PANIC-based memory isolation mechanisms; the performance of the UAO setting instruction is the key to the performance of entering and exiting IEE. In light of this, we evaluated the performance of such two operations in this experiment.

**Measuring the LSU instruction.** As memory access instructions, the LSU instructions (i.e., LDTR and STTR) were compared with the normal load and store instructions, LDR and STR. We assessed the latency of these four instructions under controlled cache hits. The target address of each memory access instruction was cache-line aligned in this experiment. Table 3 gives the experimental results. We can see the LSU instructions are as fast as normal load/store instructions, which is encouraging and implies the high isolation efficiency of the PANIC-based memory isolation.

**Measuring the UAO setting instruction.** The UAO setting instruction is the core of the IEE gate. We chose three traditional implementations of IEE as compared targets: 1) page table manipulation technique that enables and disables the access permission of IEE when entering and exiting IEE; 2) privilege switching technique that isolates IEE at a higher privileged level and switches privilege level when entering and exiting IEE; 3) remote procedure call technique that isolates IEE to other processes. SVC instruction is essential in them because the entry and the exit of IEE need to trap.

### 8.2 Evaluating Kernel Operations

PANIC needs to interpose all control flow between the user and kernel, which changes the way users interact with the kernel which will affect the performance of kernel operations. We used lmbench v3.0-a9 to measure the overhead imposed by PANIC on basic kernel operations. The experimental results are shown in Table 4. The geometric mean of PANIC’s overhead is 45.2%. In particular, it incurs significant overhead in handling lightweight system calls (bold font in the table). This is in fact expected — the lightweight system call tests are mainly used to test the latency of trapping into the kernel. For example, *null call* only invokes the getppid() system call which involves very little kernel operation in a loop. In contrast, PANIC needs additional isolation operations in shim. As a result, system calls with simple kernel operations tend to have higher performance overheads with PANIC.

### 8.3 Protecting Shadow Stack in CFI

We chose the shadow stack as our protected target. The shadow stack was implemented based on the shadow call stack instrumentation pass in LLVM [39]. Its integrity is ensured by using the information hiding technique that places it at a random location. This probability-based pseudo-isolation technique has been proven vulnerable [18, 19, 22, 47, 54, 55]. PANIC is used to conduct strict isolation protection for the shadow stack. For comparison, we also used LLVM compiler’s default return address protection (LLVM-PAC) [40] and the state-of-the-art mechanism PACStack [36] to protect return addresses by using ARM PA [2]. Note that PACStack still suffers from pointer reuse attacks [26], whereas the shadow stack does not.

Fig. 8 illustrates the overhead on SPEC CPU2017 C/C++ benchmarks when using the above four mechanisms to protect return addresses. The results show that when using LLVM-PAC, PACStack, the shadow stack with the probabilistic protection and the shadow
stack with PANIC’s protection to protect return addresses, the geometric mean of the performance overhead is 0.92%, 2.81%, 0.95%, and 1%, respectively. We can see that PANIC incurs negligible overhead compared to the information hiding technique, but provides stronger security guarantees. And the shadow stack mechanism equipped with PANIC performs better than PACStack on average. We can also see that some cases incur higher performance overheads, such as 500.perlbench_r, 502.gcc_r, 600.perlbench_s and 602.gcc_s. This is because the function calls and returns are very frequent in these cases which cause frequent return address signing and authentication in PACStack and frequent accesses to the shadow stack.

8.4 Protecting Session Keys in Nginx+OpenSSL

We used the PANIC-based JEE to protect SSL session keys (and relevant operations) of OpenSSL v1.1.1 in a high performance web server, Nginx v1.12.1. Nginx was configured to only use ECDHE-RSA-AES128-GCM-SHA256 cipher and AES encryption for sessions. It started 4 worker processes, and each was pinned to a different CPU performance core. We used 4 concurrent ApacheBench (ab) instances to simulate 200 concurrent clients constantly sending 100,000 requests to transfer a file remotely. Similar to the existing work [51], we varied the size of the requested file, i.e., 1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K.

As shown in Fig. 9, the performance overhead (geo_mean) of Nginx under the protection of PANIC is 4.94%. When the file is small, such as 1KB, the performance overhead is 10.92%, and when the size reaches 128KB, the performance overhead is 0.98%. We can see that as the requested file size increases, the overheads decline.

8.5 Protecting JITed Code of JavaScriptCore

To evaluate the practicality and performance of PANIC to protect JITed code, we applied PANIC to protect the code cache of JavaScriptCore-v2.38.3 (JSC). For comparison, we also implemented the mprotect() system call based protection that enables and disables the writable permission when emitting the code. We evaluated their performance overheads with the Octane benchmark [20], which is the JIT-heavy benchmark at runtime. Each JavaScript program in the benchmarks was executed 50 times, and we calculated the average score.

Fig. 10 shows the performance overhead on the Octane benchmark. The baseline is the unmodified JSC on the Octane which does not apply any memory protection on the code cache (has both the writable and the executable permissions). We can see that the mprotect-based protection incurs a geometric average overhead of 2.72%, and PANIC incurs a geometric mean overhead of 0.83%. The mprotect-based protection incurs higher overheads on some test cases, such as DeltaBlue, SplayLatency and Typescript. This is due to these cases triggering more frequent JIT operations that cause the frequent invoking of the mprotect(). PANIC incurs higher overheads on MandrellLatency and Typescript than other cases. It is because they trigger more code to be emitted that causes more frequent instruction inspections in the PANIC-vDSO library.

8.6 Statistics of Sensitive Instructions

As mentioned in §5.1, sensitive instructions can be conditionally or unconditionally. And conditionally sensitive instructions can be further subdivided according to the system configuration. In this experiment, we counted the classification results of sensitive instructions and the number of sensitive instructions encountered when running benchmarks. Fig. 11 gives the classification results. Cond.(ignore) means conditionally sensitive instructions that do not require handling since their behavior is the same in different modes under the default system configuration in our experiments; Cond.(emu) means conditionally sensitive instructions that require capturing and emulating. We can see that among 1,529 instructions, there are 530 unconditionally sensitive instructions and 344 conditionally sensitive instructions. There are 274 conditionally sensitive instructions that do not require handling, such as DC CIVAC, and only 70 conditionally sensitive instructions need to be handled, such as MRS <Xt>, CTR_EL0.

Table 5 shows the number of sensitive instructions encountered when running all benchmarks used in our experiments. We can see that no unconditionally sensitive instructions appeared, all Cond.(ignore) instructions do not need to be handled, and only a few sensitive instructions list in row Cond.(emu) need to be transformed. We also found that all Cond.(emu) instructions occurred in the libraries. We only transform Uncond. and Cond.(emu) instructions during the runtime screening, and the screening speed is 691.37 KB/ms.
To protect the OS kernel on AArch64, ILDI \[14\] isolates the sensitive code to ensure system security. \[41, 53\] that run code at privileged level are all implemented on X86. Besides, new security challenge — how to run unprivileged code at privileged level securely. Besides, PANIC also provides more generic protection abilities by using UAO, such as protecting JITed code and creating an isolated execution environment.

### Intra-address space isolation on ARM

Apart from the works \[3, 14, 34, 59\] that use LSU to isolate data, there are other works exploring other hardware features on ARM to conduct the isolation. As for intra-process isolation, Shred \[11\] uses ARM memory domain \[1\] that implements a domain isolation mechanism, but memory domain has been removed in AArch64. Another work \[27\] proposes a method to isolate memory within a process using hardware debugging, specifically by using a watchpoint to monitor a particular memory region containing secret data. However, both setting a watchpoint register and switching a memory domain require system register settings, which require trapping into the kernel. Sealer \[13\] encodes the encryption key that needs protection into the instruction and puts these instructions in an execution-only memory of the user mode provided by AArch64, where only trusted code can execute this part of the code to obtain the data. However, this method is not suitable for generic data protection, such as data that can be modified dynamically. The works based on ARM Memory Tagging Extension (MTE) \[35, 42\] color memory objects and restrict code access to these objects by sanitizing pointer-dereference operations, which accelerates the bounded acquiring \[42\] and the bounded checking \[35\]. However, they require instrumenting all memory access operations, leading to code bloating and performance overhead.

For in-kernel memory isolation, HAKC \[42\] proposes a kernel compartmentalization enforcement mechanism by using PAC and MTE, which needs lots of instrumentation on the kernel. SKEE \[5\] chooses to configure different page tables for different components and switches page tables during the component switching; it relies on the hypervisor to achieve the secure page table switching. Hilps \[15\] uses the method of adjusting TxSZ to control the accessible virtual memory area to achieve in-kernel memory isolation. TZ-RKP \[4\] proposes to use TrustZone on ARM to achieve isolation.

### Intra-process memory isolation on X86

Existing works can be divided into three categories: 1) The address-based method by using Intel Memory Protection Extensions (MPX) \[25\] to accelerate the bound-checking \[7, 31\]; 2) The domain-based method by using various hardware features to accelerate the switching of access permissions, such as using Intel Memory Protection Keys (MPK) \[8, 23, 24, 29, 48, 49, 51\], using VMFUNC \[24, 31, 38, 50\], and using Intel Supervisor Memory Access Prevention (SMAP) \[53\]; 3) The specialized move method, similar to PANIC, uses different memory access instructions to access isolated memory regions. IMIX \[17\] and MicroStache \[43\] add new memory access instructions for sensitive memory pages by extending the X86 ISA. CETIS \[57\] achieves this method on commercial processors based on Intel Control-flow Enforcement Technology (CET) \[25\]. However, CETIS only ensures the integrity of isolated memory regions.

### 9 RELATED WORK

#### Running user code at privileged level

Existing works \[6, 21, 32, 41, 53\] that run code at privileged level are all implemented on X86. Similar to PANIC, SEIMI \[53\] runs an untrusted process in ring 0 of the VMX non-root mode based on Intel VT-x \[25\], and leverages Intel Supervisor Memory Access Prevention (SMAP) to provide the intra-process memory isolation for sensitive data. SEIMI relies on virtualization, which incurs additional performance overhead. Compared with SEIMI, PANIC does not need to rely on virtualization, and avoids frequent domain switching. Running user code at privileged level can also be applied in other scenarios. For example, Dune \[6\] supports a secure and trusted process running in ring 0 of the VMX non-root mode, allowing the process to manage the exceptions and page table. DangZero \[21\] proposes an efficient use-after-free detection method. Dune and DangZero ensure system security based on virtualization. KML \[41\] runs a program at privileged level, which is written in a typed assembly language and passed the kernel’s security checking, thus achieving the faster system call invocation. Similar to KML, Privbox \[32\] proposes a faster system call invocation method, and it uses instrumented (sandboxed) code to ensure system security.

#### Use of Load/Store Unprivileged Instructions

The LSU instructions have already been explored for security \[3, 14, 34, 59\]. However, these works differ from PANIC in both application and implementation. To protect embedded systems on ARM Cortex-M processors, Silhouette \[59\] proposes a shadow stack mechanism based on the LSU instructions against the control flow hijacking attacks. uSFI \[3\] proposes to use the LSU instruction to isolate untrusted modules, and uXOM \[34\] transforms normal load instructions to the LSU instructions to implement execute-only memory. To protect the OS kernel on AArch64, ILDI \[14\] isolates the sensitive data, such as page tables, into a safe region by using PAN and LSU. Different from previous work to protect privileged software, PANIC is proposed to use privileged hardware for protection by running unprivileged software at a privileged level. That raises a new security challenge — how to run unprivileged code at privileged level securely. Besides, PANIC also provides more generic protection abilities by using UAO, such as protecting JITed code and creating an isolated execution environment.

### 10 CONCLUSION

Intra-process memory isolation is a classical technique to improve security within a process. In this paper, we propose PANIC, an efficient and generic memory isolation technology based on PAN and LSU in AArch64. To leverage such two features, securely running the user process in kernel mode is needed. To this end, PANIC propose two new techniques: shim-based memory isolation

### Table 5: The number of sensitive instructions that appeared when running benchmarks.

<table>
<thead>
<tr>
<th>Type</th>
<th>Cond.(emu)</th>
<th>Cond.(ignore)</th>
<th>Cond.</th>
<th>Uncond.</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncond.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Cond.(gua)</td>
<td>98K</td>
<td>247K</td>
<td>54K</td>
<td>106K</td>
<td>413K</td>
</tr>
<tr>
<td>Cond.(emu)</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Type</td>
<td>531</td>
<td>538</td>
<td>541</td>
<td>544</td>
<td>557</td>
</tr>
<tr>
<td>Uncond.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Cond.(gua)</td>
<td>76K</td>
<td>99K</td>
<td>44K</td>
<td>50K</td>
<td>54K</td>
</tr>
<tr>
<td>Cond.(emu)</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Type</td>
<td>620</td>
<td>623</td>
<td>625</td>
<td>631</td>
<td>638</td>
</tr>
<tr>
<td>Uncond.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Cond.(gua)</td>
<td>121K</td>
<td>219K</td>
<td>83K</td>
<td>82K</td>
<td>110K</td>
</tr>
<tr>
<td>Cond.(emu)</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

DUNE CPU 2012-2017 & N/A
and sensitive instruction emulation. PANIC provides a generic and efficient isolation primitive that can be applied in three different isolation scenarios. Experiments show that PANIC is more efficient than existing methods.

ACKNOWLEDGMENTS

We thank Mingfan Peng for his help with the experiment and the anonymous reviewers for their insightful suggestions and comments. This research was supported by the National Natural Science Foundation of China (NSFC) under Grants 61902374, 62272442, U1736208, and the Innovation Funding of ICT, CAS under Grant No.E161040.

REFERENCES


Ioan Gras, Kaveh Raezari, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In NDSS.


Ben Gras, Kaveh Raezari, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In NDSS.


Ioan Gras, Kaveh Raezari, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In NDSS.

Ioan Gras, Kaveh Raezari, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In NDSS.

Ioan Gras, Kaveh Raezari, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In NDSS.

Ioan Gras, Kaveh Raezari, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In NDSS.

Ioan Gras, Kaveh Raezari, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In NDSS.
A SEPARATING OF CODE AND DATA

Since PANIC treats all encodes in code pages as instructions, the data mixed in the code segment may be misidentified to be a sensitive instruction, the transformation on it will affect the program’s normal execution. Therefore, the data need to be separated from the code segment.

A.1 Data Mixed With Code

We analyzed the source code of compiler/linker and all binaries stored in the /usr directory in our experimental environment to figure out all types of data that could be mixed in the code segment:

- **Embedded data.** It includes two data types: padding and literals. Padding is used to make the code alignment which is never referenced. Literals are read-only data (i.e., constant values and addresses) that are often placed near the instruction to which they are referred, which can reduce the number of emitted instructions.

- **ELF sections.** During linking, the static linker usually bundles the sections with compatible access permissions to form a segment. For example, the read-only data sections (such as .rodata and .gnu.hash sections) with the code section (e.g., .text section) into the code segment.

A.2 Tools to Conduct the Separation

To separate code and data, we provide two tools to process the program with and without source code, respectively.

**Compiling and linking tool.** The compiling tool is used to avoid emitting the embedded data: 1) Padding need not to be handled due to its encodes are only the NOP instruction or zero whose encodings are not sensitive; 2) For constant literals, PANIC chooses to embed literals into the immediate operations of instructions. This is done by modifying the AsmParser pass to transform the literals referencing instructions to a semantically equivalent MOV and MOVK instruction sequences; 3) For address literals, PANIC redirects them to a newly created data section, which has only read permission. The linking tool is used to avoid bundling other ELF sections into code segment. This is achieved by adding the _Z separate-code option to linker during linking.

**Binary rewriting tool.** We mainly use existing techniques [12, 16] to separate code and data. Here, we only briefly introduce our method. For embedded data, we collect the reference sites within the code which are treated as the storing locations of embedded data by performing the static binary analysis. Then, embedded data is copied to a new read-only section and all relevant references will be updated; For ELF sections, we identify and move them to a new read-only segment, all references are also updated. The old embedded data and ELF sections in code segment will be cleared to zero.


