# SecureLine: Highly Efficient Intra-process Memory Isolation on AArch64


## ABSTRACT

TAB



## CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

## KEYWORDS

TAB

**ACM Reference Format:**

. 2018. SecureLine: Highly Efficient Intra-process Memory Isolation on AArch64. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX


## 1 INTRODUCTION

## 2 BACKGROUND

### 2.1 Intra-process Memory Isolation on ARM

Sensitive data includes confidential information that users do not want to disclose, such as passwords, or private data that they do not want to be compromised, such as sensitive logs and metadata. In-process memory isolation provides strong protection for sensitive data against memory-corruption attacks from malicious code within the same process. Each in-process memory isolation mechanism provides confidentiality or integrity protection for sensitive data, or both.

Some researches focus on providing in-process isolation protection for ARM processes. Shred[] uses the ARM memory domain hardware mechanism to provide multiple permission-controlled memory domains for processes to prevent malicious code from accessing sensitive data across domains, but the ARM memory domain feature was removed in ARMv8. Hardware debugging is used for memory isolation by monitoring a specific block of memory containing confidential data using watchpoints[]. However, the maximum range of watchpoint isolation is 2GB. SFI is a technique for isolating untrusted code by instrumenting memory range check code to ensure that untrusted code can only access predetermined memory spaces, but this can cause the process code to expand dramatically and also incur significant performance overhead when memory access is intensive. ARMlock provides a fault isolation assisted by the ARM memory domain hardware mechanism, which significantly improves performance compared to SFI. But ARMlock is also limited by hardware and cannot be ported to ARMv8. Isolating data through different privilege levels and preventing untrusted code from accessing it is a viable solution for sensitive data isolation, but normal code also needs to go through privilege level transitions to access isolated data, and the overhead of such transitions cannot be ignored.

### 2.2 Unprivileged Load/Store

Unprivileged Load/Store is a special type of data access instruction. No matter what Exception Level this type of instruction is executed at, it accesses memory with the access permission at EL0. ARM64 supports multiple levels of privilege, called Exception Level(EL), ranging from low to high as EL0 to EL3. The lowest level, EL0, is described as unprivileged, while the other levels are privileged. The access permission of data access instructions usually matches the level at which these instructions are executed, except Unprivileged Load/Store instructions, which ignore this rule and only access memory with EL0's permission.

**Privileged Access Never (PAN)** When PAN is enabled, privileged data access to memory that is accessible at EL0 is blocked. However, since Unprivileged Load/Store instructions access memory with EL0's permission, they are not constrained by the PAN mechanism.

**Unprivileged Access Override control (UAO)** When the UAO mechanism is enabled, it modifies the semantics of Unprivileged Load/Store instructions so that their access permission matches the level at which these instructions are executed, rather than accessing memory with EL0's permission.

**Access Permission** Under ARM64, the permission of each page is determined by the 2-bit AP[2:1] field on the PTE. The AP field has four possible values, each representing the read and write permissions of the page at EL0 and at higher Exception Levels. The Linux operating system uses the values of AP[2:1] and the mapping relationship between permissions at different levels to treat AP[2] as an R/RW indicator and AP[1] as a privilege indicator. We have adopted this Linux identification method, treating pages with AP[1] = 1 as Unprivileged Pages (U-Pages) and pages with AP[1] = 0 as Privileged Pages (P-Pages).

In this work, we focus on a two-layer model consisting of the operating system and user processes. The level at which the operating system runs can be either EL1 or EL2, depending on the specific configuration. For ease of expression, we use ELx to indicate the level at which the operating system runs and EL0 to indicate the level at which user processes run. The ARM64 version of the Linux operating system defaults to enabling PAN and disabling UAO when the operating system is running. This means that except for Unprivileged Load/Store, all other data access instructions in ELx cannot access U-Pages.

### 2.3 Exception Handling

In ARM64 architecture, exceptions are divided into two major categories: synchronous exceptions and asynchronous exceptions. Asynchronous exceptions can be further subdivided into IRQ, FIQ, and SError. The generated exception will be routed to the target


*Conference acronym 'XX, June 03–05, 2018, Woodstock, NY*



ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

*Exception level*, which may be a higher EL or the current EL. Exceptions generated under EL0 must be routed to a higher privileged Exception level.

When an exception occurs, the process running status is saved to the *Saved Program Status Register* (SPSR_ELx). This register will save the Exception level and the stack register used by the process at the time of the exception. Every Exception level has two optional stack pointers SP_ELx and SP_EL0, except EL0 which is forced to use SP_EL0 as its stack pointer. The stack pointer in use is indicated by *SPSel* register. When an exception occurs, the CPU automatically sets the value of SPSel to 1, so that the kernel stack pointed by SP_ELx can be used right after the execution is switched to the kernel context.

After an exception occurs, execution is moved to the target Exception level and forced to start from a specified code address in the Exception Vector. The *Exception Vector* is a vector table composed of sixteen 0x80-length bytes, and its starting address is stored in the *Vector Base Address Register* (VBAR_ELx) associated with the Exception level that handles the exception. 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. Any entry handling the exception originating from the current Exception level and using SP_EL0 is considered an invalid entry in Linux operating systems.

## 2.4 Address Translation

In ARM64, before accessing memory, the CPU needs to translate the virtual address of the memory through the *Memory Management Unit* (MMU) to obtain the physical address and memory attributes. This process requires multiple levels of translation table lookups to complete. *Translation Lookaside Buffers* (TLBs) reduce the average overhead of memory access by caching the results of translation table walks.

A process has a 64-bit address space. The common setting in Linux is to use the upper 256TB as kernel space and the lower 256TB as user space, with 48 bits of virtual address being valid. The kernel space and user space have their own *Translation Table Base Registers*, TTBR1_ELx and TTBR0_ELx respectively. The process switch needs to switch TTBR0_ELx. In the following sections, we will use *TTBRn_ELx* to represent both TTBR0_ELx and TTBR1_ELx.

Since TLBs cache the results of translation table walks, it is necessary to invalidate the TLB entries of the previous process during a process switch. To avoid invalidating TLB entries every time a switch occurs, the *ASID* mechanism was introduced. Each time a process accesses a Non-Global page, the process's ASID is cached in the TLB entry along with it. There are two storage points for the process's ASID, which are located in the high 16 bits of the page table base registers TTBR1_ELx and TTBR0_ELx respectively. Which storage point of ASID is actually used by the process is determined by TCR_ELx.A1.

The *Translation Control Register* (TCR_ELx) is a control register for the address translation process that includes many fields. In addition to controlling ASID selection, there are also control bits, *EPD1* and *EPD0*, for turning on and off space translation table walks. When EPD1 is 1, kernel space translation table walk is disabled in case of TLB miss and will trigger a Translation fault. When EPD0 is 1, it has similar control over user space.In the following sections, we will use *EPDn* to represent both EPD0 and EPD1.

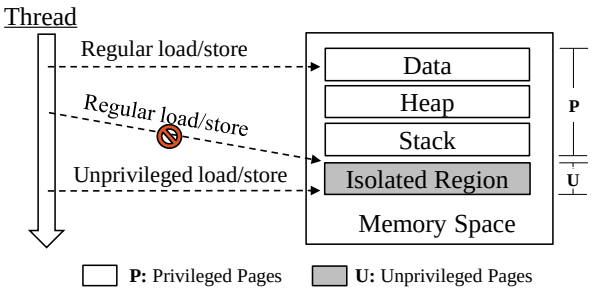


**Fig. 1: The memory layout of the kernel-mode process under SecureLine.**

# 3 OVERVIEW

## 3.1 SecureLine: Unprivileged access-based Isolation Technology

Constrained by the ARM64 Privileged Access Never (PAN) mechanism, regular load/store instructions running in kernel mode do not have permission to access unprivileged pages. However, certain unprivileged load/store instructions operate outside the constraints of this mechanism. Specifically, the *LDTR/STTR* instructions operate as if they are executing at EL0 for permission checking.

We have developed a generic and efficient memory isolation technology SecureLine by leveraging PAN and unprivileged load/store instructions. The application is run in kernel mode, and Fig. 1 illustrates the memory layout of the kernel-mode process. SecureLine sets the isolated memory region as unprivileged pages and leverages the PAN mechanism to prevent regular memory access instructions (e.g., *LDR/STR*) from accessing this region when executed in kernel mode. Trusted code, on the other hand, can access the isolated memory region using unprivileged memory access instructions (e.g., *LDTR/STTR*). Other memory regions, such as heap and stack, are set as privileged pages and can be accessed by regular memory access instructions.

By separating memory access instructions, SecureLine facilitates trusted code to access the isolated memory region using unprivileged memory access instructions, while preventing untrusted code from accessing it using regular memory access instructions. Table 1 presents the CPU clock cycles required for regular and unprivileged memory access instructions running in kernel mode on the Apple M1 chip, averaging over 10,000 tests. The results demonstrate that the CPU cycles for unprivileged instructions are nearly identical to those of regular memory access instructions. It means that ***applying SecureLine's memory isolation technology incurs minimal overhead.***

To run applications in kernel mode and achieve efficient memory isolation through hardware features, SecureLine has the following properties:

- **Security.** Attackers exploiting vulnerabilities in kernel-mode applications may compromise the entire system. Therefore, it is imperative for SecureLine to ensure the security of the system.
- **Usability.** SecureLine offers a set of user-friendly APIs that enable applications to easily achieve efficient isolation in various

**Table 1: Latency of memory accessing instructions**

| Inst. | Cycles | Notes |
|---|---|---|
| **LDR** | 0.531 | Regular memory load instruction |
| **LDTR** | 0.513 | Unprivileged memory load instruction |
| **STR** | 1.016 | Regular memory store instruction |
| **STTR** | 1.004 | Unprivileged memory store instruction |

scenarios, such as protecting metadata of defense mechanisms, sensitive data, and code. Further details will be discussed in §4.

## 3.2 Threat Model

SecureLine assumes the system's operating system and hardware to be both secure and trustworthy, with the DEP mechanism deployed. Given the universal applicability of SecureLine, we have abstracted two usage scenarios and identified the corresponding attacker capabilities.

- **Protecting metadata of the memory corruption defense mechanism,** which is employed to prevent control flow hijacking by attackers. Attackers have arbitrary memory read/write capability and aim to hijack control flow. The protected programs can be server programs, such as Nginx web server, or user programs, such as browsers. By utilizing SecureLine, the metadata of the defense mechanism, such as the shadow stack, can be protected against tampering.
- **Protecting sensitive data and dynamically generated code in software programs,** including libraries such as OpenSSL and software containing just-in-time compilers. In this case, attackers possess arbitrary memory read/write capability and can hijack control flow, with their goal being to leak sensitive data (such as session keys in OpenSSL) or modify dynamically generated code (such as JITed code), thereby breaking code integrity and inserting shellcode. By deploying SecureLine, sensitive data and code can be protected against such attacks.

## 3.3 Key Challenges

To ensure the *security* of the operating system and prevent kernel-mode processes from compromising it, SecureLine needs to address challenges 1 and 2. Additionally, challenge 3 needs to be faced with maintaining *usability*.

**Challenge 1: Preventing kernel-mode processes from tampering with kernel data.** Applications running in kernel mode have permission to access kernel data structures such as translation tables (i.e., page tables) and exception vector tables. A malicious program could manipulate the translation table to control the mapping of memory pages, enabling it to launch attacks on other processes and the operating system.

**Challenge 2: Preventing kernel-mode processes from abusing sensitive instructions.** Applications running in kernel mode have permission to execute sensitive instructions. For example, instruction *MSR <Xt>, TTBR0_EL1* modifies the translation table base register *TTBR0_EL1*, allowing the victim process to use a forged translation table created by the attacker. This leads to the modification of any physical memory, compromising the security of the system.

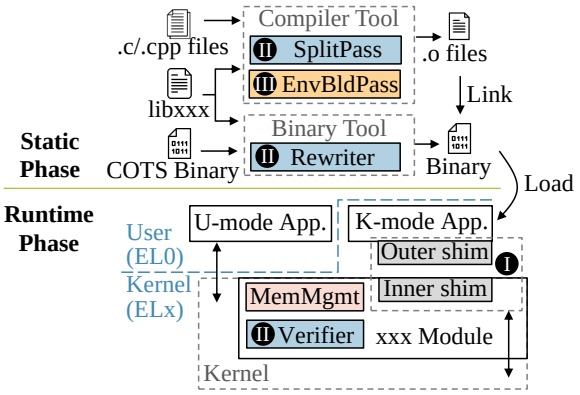


**Fig. 2: SecureLine framework.**

**Challenge 3: Leveraging SecureLine to achieve more general intra-process isolation.** While using SecureLine to protect metadata in defense mechanisms is relatively straightforward, further design is required to protect other types of sensitive data and code. For instance, it may be necessary to create a secure and trusted isolation execution environment for sensitive data. There is also a challenge in protecting JITed code, as code running in the ELx cannot execute unprivileged writable pages due to hardware limitations.

## 3.4 Approach Overview

SecureLine addresses the three challenges mentioned in §3.3 by using three tasks. Fig. 2 depicts SecureLine's software framework, where components belonging to Tasks I, II, and III are identified. During the static phase, SecureLine processes input source code or binary using a compilation or binary tool to generate a binary that does not contain sensitive instructions. During the runtime phase, SecureLine provides a set of components to run the input binary in kernel mode, ensuring security and usability.

**Task I: Rebuild bidirectional memory isolation between processes and the kernel.** To address Challenge 1, we implemented a set of shims that use EPDx (x=0,1) and ASID in AArch64 to achieve memory isolation between kernel-mode processes and the kernel. The core idea is to set the kernel space as inaccessible in the Inner shim before switching to the user context, and set the user space as inaccessible in the Outer shim before switching to the kernel context.

**Task II: Restrict the execution of sensitive instructions.** To address Challenge 2, we comprehensively collect and restrict all sensitive instructions using multiple techniques. In particular, SecureLine intercepts the execution of all sensitive instructions in ELx by (1) filtering the instruction encodings and stopping the execution, (2) configuring instructions to make them insensitive, and (3) raising processor exceptions and simulating the execution.

**Task III: Adapt to general memory isolation scenarios.** SecureLine constructs an isolated execution environment for sensitive data, using the UAO mechanism to prevent untrusted code from jumping into trusted code in the middle, and storing intermediate variables in user-space pages to prevent them from being tampered with. When isolating JITed code, we propose a method to separate write/execution based on shared memory: when allocating the isolated memory

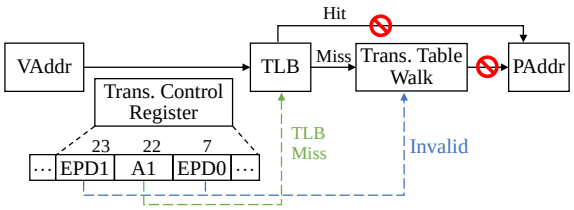


**Fig. 3: XXX.**

region for the JITed code, we allocate two virtual memory regions for the same physical memory region; one is configured as U-pages that can be read and written, and the other is set to be U-pages that can be read and executed.

## 4 SECURELINE FRAMEWORK

As depicted in Fig. 2, the *MemMgmt* component sets the privilege level of the process to kernel mode and converts the process's regular memory pages into privileged pages during both the process's startup and runtime, allowing the process to execute in kernel mode. SecureLine is compatible with Linux operating systems and POSIX APIs. The change in the process's privilege level is transparent to the operating system, which still treats it as a user-mode process. To ensure *security* and *usability*, SecureLine comprises three tasks: rebuilding bidirectional memory isolation (§4.1), restricting sensitive instructions (§4.2), and applying generic isolation (§4.3).

### 4.1 Preventing the operating system from being corrupted

**Access control to memory spaces.** The memory of the operating system and processes are bi-directionally isolated: processes cannot access the memory of the operating system, modify privileged data or execute privileged code; the operating system code cannot arbitrarily access the memory of processes to prevent ret_to_user attacks. However, if processes and operating systems run at the same privilege level, the memory isolation between them does not exist. In order to protect the security of the operating system, SecureLine uses the EPDn and the A1 to reconstruct the memory of the kernel-mode processes isolation.

Translation Table Walks and TLB construct two paths for the address translation process, as shown in Figure 3: Slow path, when TLB miss occurs, the address translation requires translation table Walks to obtain the physical address; Fast path, when TLB hit occurs, the virtual-to-physical address mapping can be obtained directly from the TLB. EPDn prohibits translation table Walks of TTBRn_ELx pointing to space, and blocks the Slow Path. However, EPDn cannot restrict obtaining the virtual-to-physical mapping that already exists in the TLB, so when accessing memory in different execution contexts, SecureLine force kernel-mode process to use different ASIDs, thereby blocking the Fast Path. The two ASIDs are placed in TTBR0_ELx and TTBR1_ELx registers, so SecureLine only needs to switch A1 to complete the switching of different ASIDs.

**Security of TCR_ELx setting instructions.** The EPDn and A1 switching can be completed by one TCR_ELx setting instruction which exists in both the outer shim and the inner shim. The outer shim is located in the user space and consists of two routines, entry routine and exit routine. The entry routine is used to open the kernel space when the kernel-mode process switches from the user context to the kernel context, and the exit routine is used to close the kernel space when switching back to the user context. The inner shim is located in the kernel space, and is responsible for opening and closing the user space when the kernel-mode process runs in the kernel context. As we know from Overview, all sensitive instructions in the user space of the kernel-mode process will be eliminated, but the TCR_ELx setting instruction must be reserved in the user space for exception handling. However, this instruction may be exploited by the attacker.

Attackers have two optional attack targets, one is to access the kernel space arbitrarily by setting the TCR_ELx register to open the kernel space and then jumping back to attacker's code and the other is to cause the system errors through just setting critical fields of the TCR_ELx register. Both targets require to execute the TCR_ELx setting instruction. SecureLine should ensure that the routines has 1) **Atomicity**, the execution of the routines can only start from the entrance and any behavior of jumping into the routines from the middle will trigger an exception and 2) **Determinism**, the value of all registers in the outer shim must be a kernel-controlled determined value.

```
1  begin:
2      msr tpidrro_el0, x30            ## save x30.
3      msr dbgbvr0_el1, xzr            ## clear breakpoint at set_tcr
4      isb
5      mrs x30, tcr_el1                ## get current tcr_el1
6      movk x30, #0x7550, lsl 16       ## enable EPD1 and switch A1
7  set_tcr:
8      msr tcr_el1, x30                ## set tcr_el1
9      isb
10     adr x30, set_tcr                ## get the address of set_tcr
11     msr dbgbvr0_el1, x30            ## set breakpoint at set_tcr
12     isb
13 check_spsel:
14     mrs x30, SPSel                  ## get current SPSel
15     eor x30, x30, #1                ## x30 = x30 xor 1
16     cbz x30, jmp_inner              ## check if SPSel == 1
17     brk #0                          ## break if checking is failed
18 jmp_inner:
19     ldr x30, [sp, -16]              ## get the address of inner shim
20     add x30, x30, #(begin - userspace_vector_base) ## adjust to the
   ↪  corresponding entry
21     ret                             ## jmp to inner shim
```

**Listing 1: Entry Routine of Outer Shim**

**Defenses in routines.** We take the entry routine 1 as an example. The lines without background color is functional code, Lines 5-9 sets the value of TCR_EL1, and Lines 18-21 gets the address of the inner shim saved on the kernel stack, and then jump into the entry corresponding to the exception type from the inner shim. The code with a gray background is security code used to ensure the correct execution of the functional code.

We use two key technologies to protect the code of the outer shim. The first key technology is the breakpoint monitor. Breakpoint registers are debug registers in ARMv8 that can be used to monitor the execution of a certain instruction, and trigger a break exception if executed. We set the breakpoint on the instruction of TCR_ELx setting, which means that only when executing from the entrance

of the entry routine can the setting of the breakpoint register be cleared in Lines 2-3. After setting TCR_ELx, the setting of the breakpoint register is restored in Lines 10-12.

Any execution flow that executes the TCR_ELx setting instruction without clearing the breakpoint will trigger a break exception, which also ensures that TCR_ELx will not be maliciously tampered with. Although this technology can ensure the atomicity of the execution flow, it cannot prevent attackers from opening the kernel space by executing the entry routine without triggering an exception. Therefore, we introduce the second key technology, SPSel checking.

When an exception occurs, the CPU automatically sets the value of SPSel to 1. However, if the attacker directly jumps into the outer shim, SPSel will not change its value. The instruction of setting SPSel is a sensitive instruction that will not appear in the user space. Therefore, we can check the value of SPSel to determine whether the entry routine is executed in the kernel context. Lines 14-17 check whether the SPSel is equal to 1. If not, SecureLine detect a malicious attack and directly triggers a break exception.

When an exception is triggered, the interrupt is automatically turned off, and then the entry routine is executed. But if the attacker illegally executes the entry routine without turning off the interrupt, the execution flow of the entry routine may be interrupted before checking SPSel after setting TCR_ELx. SecureLine will ensure that each thread of the kernel-mode process will execute the exit routine and set TCR_ELx to close the kernel space when returning from the kernel context to the user context, so whether it is re-executed through interrupted entry routine or pre-empted by other threads, it is not feasible to open the kernel space and return to the attacker-controlled code.

## 4.2 Preventing the sensitive instructions from being abused

### 4.2.1 Definition of Sensitive Instruction.

*Definition 4.1.* For an instruction $i$, we define its **Instruction Behavior** $b^e_{i,c}$ as the changes in machine state that will be triggered after the instruction $i$ is executed in the execution context $e$ with system configuration $c$, where $e \in \{kernel, user\}$ and $c \in \mathbf{C}$, where $\mathbf{C}$ is the power set of all possible permutations of values that system control registers can take.

*Definition 4.2.* An instruction is treated as a **Sensitive Instruction** if and only if it is satisfied:

$$\exists c \in \mathbf{C}, b^{kernel}_{i,c} \neq b^{user}_{i,c} \tag{1}$$

*Definition 4.3.* A sensitive instruction is an **Unconditionally Sensitive Instruction** when it is satisfied:

$$\forall c \in \mathbf{C}, b^{user}_{i,c} = \text{UNDEFINED} \tag{2}$$

*Definition 4.4.* A sensitive instruction is a **Conditionally Sensitive Instruction** when it is satisfied:

$$\exists c \in \mathbf{C}, b^{user}_{i,c} \neq b^{kernel}_{i,c} \wedge b^{user}_{i,c} \neq \text{UNDEFINED} \wedge b^{kernel}_{i,c} \neq \text{UNDEFINED} \tag{3}$$

Apart from D3 and D4, there is another situation of sensitive instructions which is satisfied $\forall c \in \mathbf{C}, b^{kernel}_{i,c} = \text{UNDEFINED} \wedge \exists c \in \mathbf{C}, b^{user}_{i,c} \neq \text{UNDEFINED}$. However, this situation does not exist in ARMv8-A instructions.

**Table 2: Sensitive instructions of ARMv8-A**

| Line | Type | Instructions | Count | Method |
|---|---|---|---|---|
| 1 | Unco. | DC CGDSW, DC CSW, ... | 543 | ① Filter |
| 2 | | AT S12E0R, AT S1E2R... | | |
| 3 | | TLBI ALLE1, TLBI ASIDE1... | | |
| 4 | | MRS <Xt>,ELR_EL1… | | |
| 5 | | ERET, HVC... | | |
| 6 | Cond. | DC CIVAC, DC CVAC… | 284 | ② Configure |
| 7 | | MSR [FPCR/FPSR…],<Xt>, … | | |
| 8 | | CASB, LDADDB, … | | |
| 9 | | ADDG, IRG, LDG, STG, … | | |
| 10 | | DGH | | |
| 11 | | LD64B, ST64B, … | | |
| 12 | | MRS <Xt>, [RNDR/RNDRRS] | | |
| 13 | | CFP RCTX, <Xt>, CPP RCTX, <Xt>, ... | 32 | ① Filter |
| 14 | | MRS <Xt>,CNTP_CTL_EL0... | | |
| 15 | | MRS <Xt>,CTR_EL0… | 10 | ③ Trap |

### 4.2.2 Identifying and Restricting Sensitive Instructions.

We dedicated 8 person-months to meticulously reviewing the 1453 instructions in the ARMv8-A Architecture Reference Manual[]. Through careful review and experimentation, we were able to document the behavior of each instruction under different execution contexts (i.e., kernel and user). As shown in Table 2, we identified 869 sensitive instructions, including 543 unconditional sensitive instructions and 326 conditional sensitive instructions.

To restrict the unconditional sensitive instructions (Lines 1-5), we employed a filtering method to scan the instruction encoding, taking advantage of the A64 instruction set's RISC architecture and four-byte alignment. This approach proved efficient in filtering these sensitive instructions.

For the conditional sensitive instructions (Lines 6-15), we manually configured the system configuration $c$ based on a MacMini M1 processor. The configuring principles are as follows: 1) conciseness. Minimizing the number of instructions with inconsistent behavior under different execution contexts; 2) compatibility. Ensuring compatibility with both the system configuration $c$ and mainstream system configurations. Once the system configuration was configured, we classified the conditional sensitive instructions into three categories based on their behaviors:

**(1) $b^{kernel}_{i,c} = b^{user}_{i,c}$**. As shown in Lines 6-12, the behavior of 284 conditional sensitive instructions remains consistent across different execution contexts given the system configuration $c$. These instructions were classified as non-sensitive and did not require restriction.

**(2) $b^{kernel}_{i,c} \neq b^{user}_{i,c} \wedge b^{user}_{i,c} = \text{UNDEFINED} \wedge b^{kernel}_{i,c} \neq \text{UNDEFINED}$**. We employed a filtering method to prevent the execution of the instructions listed in Lines 13-14.

**(3) $b^{kernel}_{i,c} \neq b^{user}_{i,c} \wedge b^{user}_{i,c} \neq \text{UNDEFINED} \wedge b^{kernel}_{i,c} \neq \text{UNDEFINED}$**. The instruction in Line 15 can be executed in different execution contexts, for example, the semantics of the *MRS <Xt>, CTR_EL0* instruction is to read the value of the system register *CTR_EL0*, which will trigger the *MRS Instruction Execution Exception*

in EL0 and then read the sanitized value. However, when executed in ELx, it will read the real value. To address this inconsistency, XXX rewrote the instruction as an exception-triggering instruction (e.g., *BRK* instruction), intercepted the exception, and simulated the instruction's behavior.

*4.2.3 Handling Executable Data.* Embedded data. Executable sections.

*4.2.4 Runtime Verifying.*

## 4.3 Applying SecureLine for intra-process memory isolation protection

```c
/* type for function arguments that should be isolated */
typedef struct {
    void *buf;
    int len;
} sl_inbuf;

/* attribuites for sensitive data and trusted functions */
char __attribute__((sensitive)) secret[size];
__attribute__((trusted))
void trusted_function(sl_inbuf *secret_input) {...}

/* alloc and free function for isolated data area */
void * sl_alloc_data(int len, int prot);
err_t sl_free_data(void *ptr, int len);

/* type for pointers pointing to isolated code cache */
typedef struct {
    void *exec_ptr;
    void *write_ptr;
} sl_code_ptr;

/* alloc, write and free functions for isolated code cache */
code_ptr * sl_alloc_code(int len);
err_t sl_write_verified(void *dst, void *src, int len);
err_t sl_free_code(code_ptr *ptr, int len);
```

**Listing 2: API Defines**

*4.3.1 SecureLine APIs Defines.* We have defined two sets of APIs for two types of protection scenarios. For protecting sensitive data, SecureLine offers two attributes, sensitive and trusted, to label sensitive data and trusted functions, respectively. If non-register parameters of a trusted function need to be isolated and protected, they can be declared as type `sl_inbuf`, with the parameter `sl_inbuf.buf` assigned the original value, and `len` assigned the length of `sl_inbuf.buf`. If users want to manage isolated data areas, they can use two functions, `sl_alloc_data()` and `sl_free_data()`. `sl_alloc_data()` allocates an isolation area with a length of len and permissions of prot, and returns a pointer to the isolation area; prot can only be read-only (`RO`) or read-write (`RW`). `sl_free_data()` is used to release the isolation area pointed to by `ptr` with a length of `len`.

For protecting code cache, SecureLine first offers two functions, `sl_alloc_code()` and `sl_free_code()` to allocate and free isolated code cache. `sl_alloc_code()` only accepts an integer parameter to allocate a protected code area, and returns a pointer to the structure `sl_code_ptr`, which contains two pointers. The pointer `sl_code_ptr.exec_ptr` points to a privileged readable and executable page, while the pointer `sl_code_ptr.write_ptr` points to an unprivileged readable and writable page. Both pointers point to the same physical page, namely, the code cache area. To write to the code cache, the function `sl_write_verified()` must be used, with the writable pointer of the code cache as the `dst` parameter, the pointer pointing to the content to be copied to the `dst` pointer area as `src`, and the length to be copied as `len`.

```c
/* a sensitive global variable accessed by trusted functions */
__attribute__(sensitive)
char *secret = "...";

__attribute__(trusted)
int thusted_func(sl_inbuf *input, char * others){
    char *buf = input->buf;
    int result = 0;

    /* check input and others */
    check(input, others);

    for (int i; i < buf.len; i++) {
        /* read three buffers and process */
        result += compute(secret[i], buf[i], others[i]);
        ...
    }
    return result;
}
```

**Listing 3: Programming Routine of Secure Execution Environment**

*4.3.2 SecureLine Secure Execution Environment.* Sensitive data often have requirements for confidentiality and integrity. We can place sensitive data in unprivileged pages, add Unprivileged Load/Store instructions in trusted code, and ensure that only trusted code can access sensitive data. However, in practical scenarios, temporary buffers are often used to store intermediate results during the processing of sensitive data. If only access to sensitive data is protected, it cannot prevent attackers from indirectly tampering with or leaking sensitive data through intermediate buffers. Therefore, we provide a secure isolation execution environment solution based on SecureLine and combined with the UAO mechanism to handle sensitive data. Users only need to follow certain programming guidelines to run trusted code and access sensitive data in a secure isolated execution environment.

**API usage.** As shown in Listing 3, if a user has sensitive data `secret` that needs to be processed in `trusted_func`, the user first marks the sensitive data as `sensitive` (Line 2) and marks the trusted function as `trusted` (Line 5). If the trusted function has other pointer parameters that point to non-isolated memory, the user can choose to define the parameter using type `sl_inbuf`.

Code structured in this way, when compiled with our provided compiler options, will load sensitive data into an isolated area and dynamically copy parameters of type `sl_inbuf` used by trusted code into a secure isolated stack. Access to data in the isolated area will use Unprivileged Load/Store instructions. When returning from trusted code to untrusted code, all registers except for the return value register will be cleared to prevent leakage of sensitive data.

If a trusted function wants to use an isolated heap area to store data, it can use `sl_alloc_data()` to allocate memory. In addition, to ensure data security, trusted functions should not call untrusted functions to prevent untrusted functions from leaking private data from trusted functions through parameters.

**Secure Isolated Stack.** During the execution of trusted functions, stack space is often used to cache sensitive data. Therefore, we pre-allocate a secure isolated stack space for all threads and switch stacks when entering and exiting trusted functions. Local variables and `sl_inbuf` type parameters used during execution are placed on the secure stack and cannot be accessed by untrusted code using regular access instructions. The stack switching and parameter copying instructions are also added by the compiler.

**Runtime protection.** Although Unprivileged Load/Store instructions will not appear in untrusted functions, they will certainly exist in trusted functions. If left unprotected, attackers can reuse Unprivileged Load/Store instructions in trusted functions by tampering with code pointers. In SecureLine, we use the UAO feature that can change the semantics of Unprivileged Load/Store instructions to add an access protection switch for trusted functions. Each trusted function entry adds code to turn off UAO, while the exit adds code to turn on UAO. This ensures that only when entering from a trusted code entry can the isolated area be accessed, and direct jumps to Unprivileged Load/Store instructions will trigger an exception.

```
1  void jit_engine() {
2    ...
3    /* alloc isolated code cache */
4    sl_code_ptr *p = sl_alloc_code(code_len);
5    if(!p)
6          exit_error();
7    ...
8    /* write code by write_ptr pointing to the unprivileged page */
9    if (is_err(sl_write_verified(p->write_ptr, src, src_len)))
10         exit_error();
11   ...
12   /* execute code by exec_ptr pointing to the privileged page */
13   jmp_to_code(p->exec_ptr);
14 }
```

**Listing 4: Programming Routine of Isolated Code Cache**

*4.3.3 SecureLine Isolated Code Cache.* The Just-In-Time (JIT) engine exists in the runtime of many interpreted languages, aiming to transform high-level languages into low-level assembly instructions written into the JITed code cache, which accelerates the execution process of interpreted languages.

JITed code can be regarded as a special type of "sensitive data" that is dynamically generated code. It is necessary to prevent attackers from mixing their attack logic into the JITed code cache, and therefore, limiting the write operation of JITed code cache is required. However, JITed code cannot be considered entirely as "sensitive data" because the JITed code cache is the code page(s) with executable permissions, and there is a possibility of dynamically generating sensitive instructions. Furthermore, there are multiple modifications of JIT code, which means that the write permission of JIT code cache cannot be entirely prohibited after writing once.

We need a different protection plan for this type of isolated memory that requires both writable and executable permissions and may contain sensitive instructions.

**Shared physical pages.** Switching page table permissions to implement Data Execution Prevention (DEP) in the presence of a JITed code cache with multiple self-modifications is not appropriate. This not only incurs additional overhead due to system trapping, but also exposes the JIT code cache to all threads, including those that may be controlled by attackers, due to the write permission being enabled.

We aim to simultaneously enable write and execute permissions for the JITed code cache and restrict access to only trusted code. However, this cannot be achieved in AArch64, where privileged level instructions cannot execute unprivileged pages, even if those pages are marked as writable and executable under privileged level. Therefore, we adopt the approach of sharing the same physical page between a privileged page with readable and executable permissions and an unprivileged page with readable and writable permissions. This approach enables the JITed code cache to have both write and execute permissions, without requiring a switch of page table permissions. The modification of the JITed code and execution of the JITed code can be achieved using different pointers.

**Mandatory instruction validation before write.** Due to the dynamic code generation nature of JITed code, all JITed code must undergo sensitive instruction validation before execution. The instruction validation is responsible for by the SecureLine kernel module and can only be entrusted to the kernel module. SecureLine cannot allow privileged processes that can dynamically generate code to bypass the instruction validation process, generate sensitive instructions, and execute them. However, the exception-triggered instruction validation approach mentioned in §4.2 depends on modification of page table permissions, which is not as applicable for JITed code that undergoes multiple dynamic modifications, because it would frequently trigger the process to trap into the kernel.

We design a method where SecureLine takes control of all JITed code write operations, which involves mapping the JITed code write function in the process space. The function defines that instruction validation must be performed before writing, and that writing to the code cache is only permitted after a successful validation execution. The writing function also includes the User UAO mechanism for dynamically protecting the Unprivileged Load/Store instructions. This mechanism requires that the JITed Code write function is the only trusted function in the process that writes to the code cache, to avoid other functions that include Unprivileged Load/Store instructions from writing to the code cache. Therefore, the proposal for an isolated JITed Code Cache and an isolated sensitive data area are not compatible. Once an isolated JITed code cache has been requested, all code pages containing sensitive instructions will be scanned, and the Unprivileged Load/Store instructions in the process will be replaced.

**API usage.** Listing 4 demonstrates the code for protecting JITed code cache using our API. The JIT engine requests the code cache by using the API `sl_allo_code()` and receives a pointer to `code_ptr`, which is returned as p (Line 4). When code needs to be written to the code cache, the JIT engine should use `p->write_ptr` as one of the parameters for the function `sl_write_verified()` (Line 9). `sl_write_verified()` verifies whether all the code in the memory pointed to by `src` contains sensitive instructions; only then can it be written into the code. The JIT engine should utilize `p->exec_ptr` as the target address when executing the code cache.

# 5 USER CASES

# 6 IMPLEMENTATION

# 7 EVALUATION

This section focuses on evaluating the performance of SecureLine. All experiments were conducted on a Mac mini with an 8-core M1 CPU and 16 GB of RAM, running Asahi Linux (Linux kernel v5.19.4 with PAC enabled). In Section §7.1, we described the experimental configuration and preparation. In Section §7.2, we used microbenchmarks to evaluate the overhead of running SecureLine for basic kernel operations and accessing isolated memory regions. In Section §7.3, we evaluated the overhead of the isolation defense mechanism implemented in real-world applications based on SecureLine.

## 7.1 Configuration

**Microbenchmarks** SecureLine requires hooking certain kernel functions to complete the operation of running user applications in kernel mode. We used LMBench v3.0-a9 to measure the overhead of SecureLine on basic kernel operations. To avoid the overhead of both SecureLine and isolation protection mechanisms, all test cases of LMBench were run under SecureLine, and the isolation protection mechanism was disabled for only basic kernel operation overhead testing.

In order to compare the cost of accessing isolated areas of memory, we conducted tests on the time consumed by the `mprotect()` function in setting page table permissions and accessing protected memory regions, as well as the time consumed in switching privilege levels and accessing isolated data. Finally, we compared the time cost of these two modes of accessing isolated regions with with the time cost of unprivileged load/store operations.

**Real-world Applications** We evaluated the performance of the three use cases mentioned in Section §5 to test the effectiveness of the inner- isolation mechanism implemented with SecureLine. All programs were compiled using the LLVM/Clang v14.0.1 compiler with the `-O2` optimization parameter.

For the evaluation experiment of metadata protection using Shadow Stack, we first improved the runtime mechanism of LLVM's built-in Shadow Call Stack (SCS) to support the storage allocation for Shadow Stack and other features, such as multi-threading and exception handling. Then we deployed the SCS mechanism on the SPEC CPU2017 test suite and evaluated the overhead of SCS. We further isolated and protected the Shadow Stack region using the isolation mechanism of SecureLine and accessed the Shadow Stack using unprivileged load/store instructions to evaluate the overhead of the protected SCS. To compare with the existing state-of-the-art return address protection mechanism on AArch64, we fully ported PACStack from LLVM 9.0.0 to LLVM 14.0.1 and used the `-full` parameter, which provides the most complete protection, provided by PACStack to compile SPEC CPU2017.

For the experiment of protecting sensitive data, we chose to isolate and protect For the experiment of protecting sensitive data, we isolated and protected the session key in OpenSSL. Functions for allocating, releasing, and using the session key in OpenSSL were tagged as trusted functions, and the key was marked as sensitive data to build an isolated execution environment for the key. We evaluated the performance of the session key protection using the web server Nginx v1.22.1 and OpenSSL v1.1.1, selected the ECDHE-RSA-AES128-GCM-SHA256 cipher, and measured the overhead of the server's network requests and response processes during runtime protection for OpenSSL.

For the evaluation experiment of protecting the JIT code cache, we used webkitgtk-2.38.3 as the test subject. We modified the code in JavaScriptCore (JSC) of webkitgtk for allocating, writing, and releasing the JIT Code Cache to use our API to manage the isolated Code Cache region. We evaluated the performance of the modified JSC on JavaScript benchmarks Octane and compared its performance overhead with that of the unprotected JSC.

**Table 3: Latency on process-related kernel operations (in $\mu s$) - smaller is better**

| Config | null call | null I/O | stat | open close | slct TCP | sig inst | sig hndl | fork proc | exec proc | sh proc |
|---|---|---|---|---|---|---|---|---|---|---|
| **Native** | 0.13 | 0.18 | 2.37 | 6.01 | 3.22 | 0.23 | 4.18 | 450 | 1537 | 4573 |
| **SL** | 0.61 | 0.93 | 6.40 | 11.3 | 5.78 | 2.23 | 73.9 | 606 | 703 | 1166 |
| **Delay** | 3.6X | 4.1X | 1.7X | 89% | 79% | 8.7X | 17X | 35% | ?? | ?? |

**Table 4: Context-switching latency (in $\mu s$) - smaller is better**

| Config | 2p/0K | 2p/16K | 2p/64K | 8p/16K | 8p/64K | 16p/16K | 16p/64K |
|---|---|---|---|---|---|---|---|
| **Native** | 2.48 | 2.97 | 4.74 | 3.42 | 4.79 | 3.21 | 4.93 |
| **SL** | 4.31 | 4.96 | 5.73 | 5.96 | 6.53 | 6.48 | 7.22 |
| **Delay** | 73.9% | 67% | 21% | 74.2% | 36.2% | 102% | 46.2% |

**Table 5: File and VM system latency (in $\mu s$) - smaller is better**

| Config | OK Create | File Delete | 10K Create | File Delete | Mmap Latency | Prot Fault | Page Fault | 100fd selct |
|---|---|---|---|---|---|---|---|---|
| **Native** | 12.4 | 9.26 | 23.9 | 14.7 | 135.3 | 0.24 | 0.25 | 2.26 |
| **SL** | 22.2 | 14.6 | 36.6 | 20.2 | 235.7 | 11.5 | 0.64 | 5.21 |
| **Delay** | 79.5% | 57.4% | 52.71% | 37.10% | 74.1% | 47X | 156% | 131% |

## 7.2 Microbenchmarks

*7.2.1 LMBench.*

*7.2.2 Cycle comparison.*

## 7.3 Real-world Applications

*7.3.1 Protecting shadow stack.* Figure 4 illustrates performance overhead of different return-address protection mechanisms on all C and C++ test cases in SPEC CPU2017. SCS, built in LLVM, is a Shadow Stack Call mechanism deployed on SPEC CPU2017. SCS-SL is built on SCS and places the shadow stack of SCS in an isolated region. PACStack implements the return address protection mechanism based on ARMv8's PAC mechanism, which avoids context collision by constructing context based on the call chain and maintains the integrity of the return address.

The geometric mean values of the overhead imposed by the three mechanisms, SCS, SCS-SL, and PACStack, on SPEC CPU2017 are 0.85%, 0.94%, and 2.8%, respectively. The experimental results

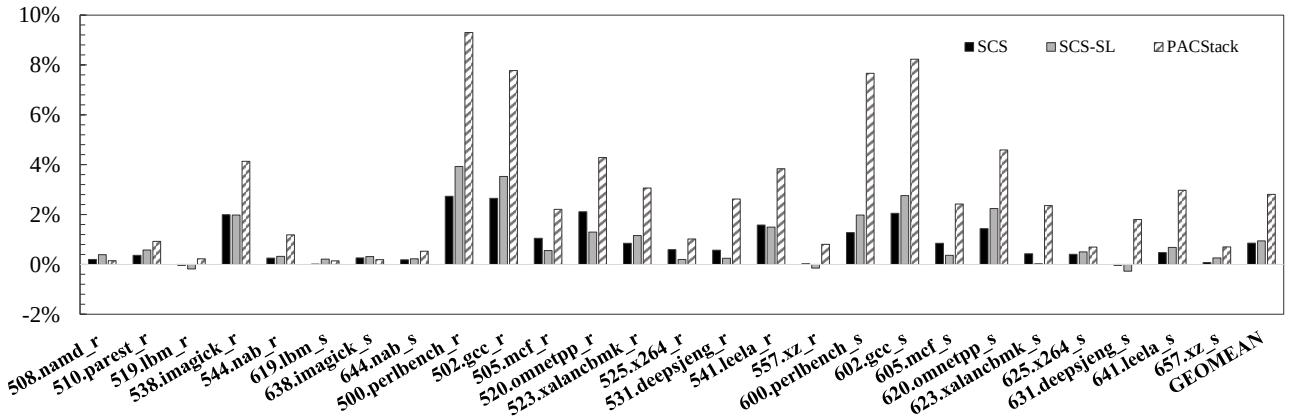


**Fig. 4: Performance overhead on the SPEC CPU2017 benchmarks incurred by defense when using SCS/SCS-SL/PACStack to protect return address. All overheads are normalized to the unprotected benchmarks.**

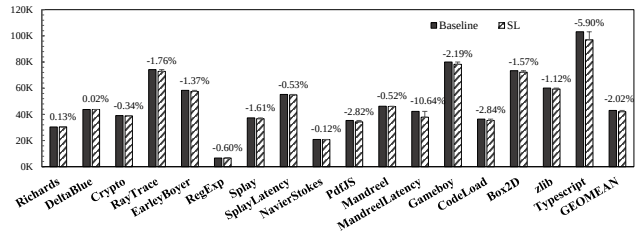


**Fig. 5: Performance score on the Octane benchmark**

show that SCS-SL incurs almost no additional overhead compared to SCS and performs better than PACStack when protecting the return address. In all test cases, except for 508 and 619, SCS-SL performs better than PACStack. Figure 4 shows that when a test case incurs a higher overhead in the PACStack experiment, the overhead in SCS-SL is relatively higher. Several test cases in the PACStack experiment, including 500, 502, 520, 523, 538, 541, 600, 602, and 620 incur high overhead ranging from 3.84% to 9.3%, with a geometric mean of XXX. The corresponding overhead ranges for SCS-SL are 1.16% to 3.93%, with a geometric mean of XXX. The overhead imposed by all protection mechanisms in some test cases, such as 508, 519, 619, 638, and 644, is close to zero because these test cases involve significantly fewer function calls than other test cases.

#### 7.3.2 *Protecting session keys in Nginx.*

*7.3.3 Protecting JITed code of JavaScriptCore.* Figure 5 illustrates the performance overhead of testing the JITed code cache protection mechanism on the Octane JavaScript benchmarks. Baseline represents the unmodified JSC experiment on Octane, while SL represents the experiment on Octane after deploying the SecureLine's JITed Code Cache isolation protection mechanism on JSC. The horizontal axis in Figure 5 represents the test cases in Octane, and the vertical axis represents the score values of each test case executing on different JSCs. A higher score value indicates better performance. It is worth noting that the original JSC on Linux does not apply any memory protection to the JITed code cache, and the JITed code cache maintains read, write, and execute permissions throughout its entire lifespan.

As shown in Figure 5, SL caused a geometric mean score decrease of 2.02% compared to the Baseline. SL produced varying degrees of score decreases in all test cases except for Richards and DeltaBlue. The score decrease ratios in most examples were not more than 2.82%, except for MandreelLatency (10.65%) and TypeScript (5.9%), which were caused by XXXX.

## 8 DISCUSSION

## 9 RELATED WORK

## 10 CONCLUSION

## ACKNOWLEDGMENTS

To Robert, for the bagels and explaining CMYK and color spaces.

## REFERENCES

## A RESEARCH METHODS

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009