SFITAG: Efficient Software Fault Isolation with Memory Tagging for ARM Kernel Extensions

Jiwon Seo
jwseo@sor.snu.ac.kr
Seoul National University
Republic of Korea

Junseung You
jsyou@sor.snu.ac.kr
Seoul National University
Republic of Korea

Yungi Cho
ygcho@sor.snu.ac.kr
Seoul National University
Republic of Korea

Yeongpil Cho
ypcho@hanyang.ac.kr
Hanyang University
Republic of Korea

Donghyun Kwon*
kwondh@pusan.ac.kr
Pusan National University
Republic of Korea

Yunheung Paek*
ypaek@snu.ac.kr
Seoul National University
Republic of Korea

ABSTRACT

As ARM is becoming more popular in today's processor market, the OS kernel on ARM is gradually bloated to meet the market demand for more sophisticated services by absorbing diverse kernel extensions. Since this kernel bloating inevitably increases the attack surface, there has been a continuous effort to decrease the surface by dissociating or isolating untrusted extensions from the kernel. One approach in this effort is using *software fault isolation* (SFI) that instruments memory and control-transfer instructions to prevent isolated extensions from having unauthorized accesses to memory regions of the core kernel. Being implementable in pure software has been considered the greatest strength of SFI and thus popularly adopted by engineers to isolate kernel extensions, but software versions of SFI mostly suffer from high performance overhead, which can be a critical drawback for performance-sensitive mobile devices that overwhelmingly use ARM CPUs. The purpose of our work, named as SFITAG, is to make SFI for ARM kernel extensions more efficient by leveraging the hardware support from the latest ARM AArch64 architecture, called the ARM8.5-A *memory tagging extension* (MTE). For efficiency, SFITAG relies on MTE support when it allocates a tag value different from the core kernel for untrusted extensions and enforces extensions to use that value as a tag for pointers and memory objects. Consequently, in SFITAG, accessing the core kernel memory is legitimate only when the tag of a pointer matches the value of the kernel tag, which by means of MTE in effect enables us to safely confine unexpected and buggy behaviors of extensions within the space isolated from the kernel. Through our evaluation, we prove the effectiveness of SFITAG by showing that our MTE-supported SFI efficiently enforces isolation for extensions just with 1% slowdown on the throughput of a network driver and 5.7% on a block device driver.

*Corresponding authors

CCS CONCEPTS

• **Security and privacy** → **Systems security**; **Software security engineering**.

KEYWORDS

Memory safety; Software Fault Isolation; Memory Tagging Extension(MTE);

ACM Reference Format:
Jiwon Seo, Junseung You, Yungi Cho, Yeongpil Cho, Donghyun Kwon, and Yunheung Paek. 2023. SFITAG: Efficient Software Fault Isolation with Memory Tagging for ARM Kernel Extensions. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '23), July 10–14, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3579856.3590341

1 INTRODUCTION

In recent years, ARM has become the dominant architecture due to the huge popularity of ARM-based mobile devices. To control these devices, the ARM operating system (OS) embraces many *kernel extensions*, such as network and device drivers, which provide functionality or hardware support that otherwise would not be a part of the OS kernel. However, kernel extensions have not been considered trustworthy in that they are an abundant source of bugs and vulnerabilities in operating systems [3, 10]. In addition, like most other modern OSes, the ARM OS has a monolithic kernel in which every OS component, including kernel extensions, operates in a single kernel space, holding all privileges to access I/O devices, memory and CPU modules in the system. This implies that even one security bug or vulnerability in a kernel extension would be fatal as such a bug exploited by an adversary might lead with ease to compromising the entire kernel dwelling in the same space and, ultimately, the whole system under control of the kernel as well [15, 32]. In light of all these, it is evident that unless this potential threat posed by such buggy, untrusted extensions is properly tackled, the ARM kernel would always remain at a high risk of being subverted and manipulated by adversaries.

For many years, diverse studies [9, 13, 20, 20–23] have been conducted to combat these constant menacing bugs lurking in the OS kernel. The tactic they opted for is basically the same: isolating untrusted extensions from the rest of the kernel. This is implemented first by constructing an isolated domain where any unauthorized access from inside to outside is strictly banned, and placing an untrusted extension into the domain. One prominent technique for this kernel isolation would be *software fault isolation* (SFI) [30, 33] that establishes an isolated domain (*i.e.*, sandbox) by instrumenting memory and control-transfer instructions in a way to regulate all accesses from an untrusted extension within its domain boundaries. This isolation mechanism of SFI can transform legacy software encompassing existing kernel components by instrumentation to allow security-critical code to run safely in the same memory space as the untrusted party. One attractive merit of SFI is its diversity in granularity at several levels for memory isolation that helps us enjoy the benefits of high precision in access control. Specifically, SFI offers memory isolation not only at page-level granularity but at finer levels such as byte and word. For instance, in previous work [9], by using SFI, the authors were able to accurately regulate data access of every domain at byte-level granularity.

The greatest strength of SFI would be that it is implementable in pure software. But the same strength becomes a weakness in terms of performance as software versions of SFI usually find much harder time meeting performance constraints thus suffering from serious performance degradation, which can be a critical drawback for ARM-based mobile devices that we are targeting in our work. For example, one SFI solution for isolating kernel extensions induces over 30% throughput loss because of the increased time and frequency to look up the permission tables for isolating many memory blocks [9].

Fortunately, a more recent study reveals a glimpse of evidence that SFI can become a lot more efficient with the benefits of hardware performance to a certain extent [16]. Motivated by this empirical study, we in this paper propose SFITAG, a SFI solution that aims to efficiently isolate untrusted extensions from the ARM OS kernel by gaining hardware support of the *memory tagging extension* (MTE), ARM's new architecture feature introduced to the newest generations of AArch64 CPUs. The ARM MTE implements lock and key access to memory. For this, every pointer and memory object is augmented to hold tags (pointer tag and memory tag). Only when the tag values in the pointer and memory object are matched, the memory access to the object via the pointer is permitted. With this hardware-supported memory access policy exerted by MTE, SFITAG is able to efficiently enforce SFI on ARM-based mobile systems in which efficiency is usually of higher priority than in desktop or server systems. SFITAG isolates an untrusted extension from the kernel by compelling all memory objects and pointers in the extension to use different tag values from those used by objects and pointers in the core kernel. Notably, MTE performs tag check in parallel with an ordinary memory operation, thus saving extra cycles that would otherwise be necessary to sequentially execute instructions for tag check in a pure software version of SFI. Our evaluation demonstrates that being aided by MTE hardware tagging, SFITAG is able to isolate a network driver with just 1% slowdown on reception.

2 RELATED WORK

Previous work related to ours can be categorized according to their ways of implementing kernel isolation as follows.

User-level Extensions. One way adopted by researchers [7, 8, 25, 26] is to realize kernel isolation by running an untrusted kernel extension as a user process. Like an ordinary user process, extensions have lower privilege than the kernel, and consequently, they cannot gain enough privilege to access kernel regions. To allow the extension to communicate with a physical device in user space, they have to redesign the interface between user and kernel spaces because executing privileged CPU instructions and handling interrupts in user space are not permitted. As a consequence, isolating some kernel extensions at user-level requires a painstaking effort, which complicates and impedes a quick application of this approach to certain OSes like Linux that have complex and irregular interfaces. On the other hand, SFITAG lets isolated extensions run in kernel space, thus requiring virtually no effort to rewrite kernel interfaces.

Microkernel. For kernel safety, some researchers [6, 14, 22] built microkernels by maintaining only the core functionality in the kernel in the first place, and outsourcing the remainder part of a monolithic kernel. As kernel extensions are mostly excluded from the core part of a kernel, the microkernel design itself plays a role of a natural barrier for the core kernel against threats from malignant extensions. Most recently, there have been efforts to develop microkernels in safe languages, such as RedLeaf [22] written in Rust fortifies with security features that offer SFI. In RedLeaf, the authors have demonstrated that their microkernel is immune from attacks exploiting vulnerabilities in a network driver. Notwithstanding such a security strength, the defense solutions using microkernels have a critical drawback that they require a rewriting of the entire OS kernel. Considering the fact that a vast majority of kernels deployed in the field today are monolithic, this approach has limited applicability to the real world. In comparison, SFITAG has a clear advantage in that it can be applied directly to harden existing monolithic OS kernels by adding a relatively small amount of instrumented code.

2.1 Page Table Switching

Other researchers tackle the kernel isolation problem by employing the page table (PT) switching mechanism by disabling by default their access permission to the kernel. In their approach, all functions are wrapped to intercept every interaction, and upon function invocation, PTs are examined to validate accesses from extensions to the kernel code and data. The representative studies in this approach are Nooks [29] and SIDE [28]. The techniques in this approach suffer from the performance overhead associated with control transfers between the kernel and isolated extension. For each control transfer, there occurs a context switch accompanying routine procedures, such as loading/unloading PTs and flushing TLB, which all in all adversely affect performance. Some researchers have made effort to minimize the adverse impact on performance of the original PT switching mechanism by utilizing hardware features, such as ARM Memory Domain [1]. ARM Memory Domain assigns memory pages into different domains and the MMU checks its access permission based on the Domain Access Control Register (DACR). For example, ARMLock [35] establishes fault isolation for user applications by assigning different domain IDs to the host application and untrusted modules such as libraries. However, since ARMLock is designed to isolate the user process, it is not directly applicable to kernel extensions. On the other hand, DIKernel [19] enforces isolation by allowing kernel and extensions to use different domains. With hardware support, DIKernel and ARMLock do not need loading PTs or flushing TLB, and only require updating the DACR register. However, their performance number is still disappointing, and DACR is no longer available in recent ARM architecture. In contrast, SFITAG uses SFI that does not require in the first place such burdensome OS-level operations for its domain switching since both the kernel and isolated extensions are running in the same kernel context.

Virtualization. There have been various studies [12, 13, 17, 21, 23] to isolate kernel extensions relying on the virtualization mechanism. For isolating untrusted kernel extensions, it creates a special virtual machine (VM) designated to be an isolated domain that is to hold a kernel extension inside [12] and blocks unvetted accesses to kernel memory regions from inside the VM by configuring the corresponding extended page table (EPT). The downside of this approach, however, is that it often suffers from a considerable amount of performance overhead due to domain switching. To alleviate this performance problem, some proposed optimization methods [21, 23] that leverage hardware features of Intel state-of-the-art x64 architectures. For example, LVDs [23] seeks additional hardware support from `vmfunc` instructions for EPT switching that is intended to lower the overhead of memory isolation and domain switching. Such dextrous use of hardware serves their goal of performance optimization. Most notably, the domain switch overhead of LVDs is merely 396 cycles, which is a lot lower than 834 cycles for page-based context switches [23]. Due to this performance advantage, KSplit [13] relies on LVDs framework to support isolation between kernel and extensions with minimal human involvement. However, unluckily for users who want to protect their kernels running on ARM, these optimization techniques will be of no avail since ARM does not support the `vmfunc` instruction. Instead, for those ARM users, we have designed SFITAG to leverage the ARM MTE architecture for accelerating the performance of a SFI solution implemented to isolate ARM kernel extensions.

Software Fault Isolation. Since SFI can establish a sandbox by instrumenting memory and control-transfer instructions, there are many studies [9, 11, 20, 30] to isolate kernel extensions by putting the extension in the isolated domain in SFI. For example, BGI [9] provides an access control list (ACL) that defines the accessible regions for each extension at the byte level. By referring to ACLs, it can accurately isolate memory blocks variably sized in pages, words and even bytes at every interaction between the kernel and extensions. As another example, LXFI [20] provides an elaborated annotation system that can be used to program access permissions for isolated extensions to the kernel. Developers may set up the rules at their disposal by annotating specific kernel interfaces to an untrusted extension that they want to isolate. Essentially all aforementioned SFI techniques have been implemented and deployed fully in software. As has been said before, being deployable in pure software is a sure strength of SFI, but it can also be a genuine weakness that causes security loopholes. That is, most existing software solutions exclude the application of SFI to untrusted read accesses to the kernel, which would put the kernel at high risk of being exposed to vulnerability exploits via unchecked reads. SFITAG outperforms software-based SFI techniques even though it checks both read/write access.

Tagged Architecture. The memory tagging approach has long been studied previously [27, 31, 34]. To enforce fine-grained memory protection using tagged memory, HDFI [27] and CHERI [31] use a 1-tag bit while Loki [34] uses a longer 32-bit tag. However, a clear limitation of all these techniques is that they require special customized hardware, which hampers their application to real-world kernels running on commodity devices. In other words, a significant hardware redesign, such as the entire memory hierarchy and interfaces is required to implement fault isolation. In contrast, SFITAG is suitable for a wide deployment because it requires no specific processor change or redesign to enforce isolation for extensions, but uses the existing feature of commercial processors.

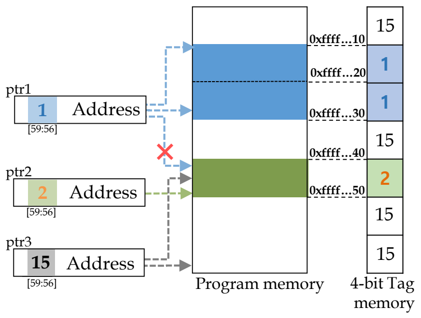

Figure 1: An example of ARM MTE operation

3 BACKGROUND

As mentioned in § 1, SFITAG relies on the hardware support from ARM MTE. Thus, in this section, we describe the architectural structure of MTE and its functionalities [4]. MTE is an architectural extension to improve the performance of memory sanity checks in ARMv8.5-A architecture. In specific, MTE introduces two types of tags: *pointer tags* and *memory tags* for pointers and memory objects, respectively. A pointer tag is a 4-bit ID located at the upper bits (*i.e.*, [59:56] bits of the address), which is ignored in address translation. A memory tag is a 4-bit ID for every aligned 16-byte physical memory. The memory tag is stored in a separate memory and the memory is not accessible from ordinary memory instructions. Instead, MTE provides several special instructions to access memory tags. For example, `ldg` and `stg` are instructions for loading and storing the memory tag, respectively. After MTE is enabled, the ordinary load and store instructions can access the memory if only the pointer tag in the address register and memory tag for the memory address match. When the tag comparison fails, the behavior of the architecture can be configured in two modes: precise mode or imprecise mode. The former mode throws a synchronous exception when a mismatch occurs, and the latter mode asynchronously reports the mismatch by updating the system register TFSR_EL1. In SFITAG, we use the precise mode. Exceptionally, MTE unchecks for memory access if memory instructions that use the stack pointer (SP) as the base register only or with an immediate offset, *e.g.*, `str` `X`$_0$`, [SP, #0x10]`. In the case of memory instructions that use SP as a base register and register offset, *e.g.*, `str X0, [SP, X1]`, MTE performs tag checks. In addition, if the 4-bit ID is `0xF` and the value of TCMA1 in TCR_ELx is `0x1`, all accesses at EL1 are unchecked, which means `0xF` tag value in the kernel space has a special property that can access the memory regardless of the memory tag value. Figure 1 shows an example of MTE operation. Each tag in `ptr1`, `ptr2` and `ptr3` has a unique ID. Two sequential memory regions have been allocated in a granule of 16 bytes. For the `ptr1` or `ptr2`, it is allowed to access the program memory if a pointer tag matches a tag of the memory to be accessed. By contrast, if the tag mismatches, it is deemed to have illegal memory access, thus access is disallowed and the exception alarm is raised. However, any access via `ptr3` with pointer tag `0xF` is permitted without raising an exception on mismatch.

4 THREAT MODEL

In this paper, we design SFITAG based on the following assumptions and attack model. Firstly, we trust the core kernel including components of SFITAG, along with the assumption that it is benign and intact. Thus, we can trust the management of MMU by the kernel, and we assume that W⊕X policy is basically applied to the untrusted extension as well as the core kernel. We assume that the extensions are vulnerable but not malicious. In other words, the developer for the extension does not implement the extension intentionally for malicious purposes but for the original intended purposes. However, we assume that there are memory vulnerabilities in the extension, and an attacker can exploit these to launch various attacks, such as code reuse attacks and data manipulation attacks. The ultimate goal of this potential attacker is to launch an attack on the kernel through vulnerabilities in the extension. Thus, we concentrate on the isolation of the extension to prevent the case that it invades and corrupts the kernel. We do not consider the situation where the hardware device of the extension is malicious. It is known that other security solutions using IOMMU in x86 or System MMU in ARM can thwart such threats [19]. Lastly, we do not consider side-channel attacks and DoS attacks on the extension.

5 SFITAG DESIGN

This section first defines a list of the requirements that SFITAG should satisfy for securely and efficiently isolating untrusted extensions, and explains how SFITAG fulfills those with ARM MTE.

5.1 Design Requirements

- **R1. Comprehensive access control:** Under our threat model, an attacker is able to access the kernel data by exploiting memory vulnerabilities in extensions. Thus, SFITAG should provide a memory access control mechanism to block any illegal accesses to the kernel. Note that, in some SFI works [9, 20], they only regulate memory write operations to avoid excessive runtime checks for read access control. However, this decision for performance optimization may induce security loopholes in that unauthorized reads to the kernel memory also can be a major threat to the system [24]. In addition, SFITAG should enforce that domain switches occur at the predefined entry/exit points, or an attacker can execute the kernel code in an unlawful context.
- **R2. Fine-grained protection:** The monolithic kernels in modern OSes like Linux communicate with kernel extensions through shared data which are stored intermingled with other data in kernel space. Since they usually enforce memory protection at a single granularity (*i.e.*, a page), it is impossible to provide individual access control for shared data differently from other kernel data if both are stored together in the same page. Obviously, in this case, for perfect isolation of the kernel extension, it is somehow necessary to provide a fine-grained memory protection mechanism that first divides a page into smaller byte-sized blocks for access control, and sets permission bits to grant the extension access only to the corresponding blocks for shared data. Not only that, we need to provide another mechanism that prevents temporal safety violations by enabling access permissions to this shared data only when the extension is being executed.
- **R3. Low performance overhead:** In the design of a SFI solution for kernel isolation, the importance of performance cannot be overemphasized. As discussed in § 1, existing solutions, due mainly to their implementations in pure software, suffer from a considerable amount of performance overhead. We have found that the overhead comes from additional instructions inserted into the original code. In our design, we will try to reduce the overhead due to software instructions by replacing the instructions with hardware operations carried out in parallel.

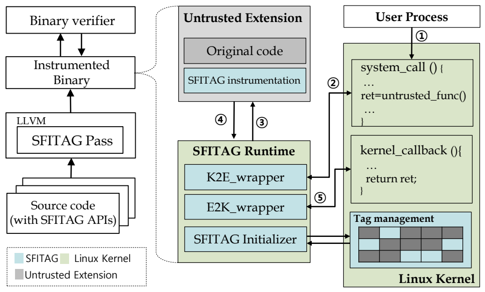

Figure 2: Overall architecture of SFITAG

5.2 Overview

Figure 2 gives an overview of how SFITAG uses MTE to isolate kernel extensions. For our work, kernel extensions have been slightly modified to use SFITAG APIs and recompiled with the SFITAG compiler to generate binaries instrumented following the procedures. Like other SFI solutions, SFITAG provides the SFITAG verifier to check if extension binaries are truly generated by the SFITAG compiler before they are loaded into the kernel. As stated in § 1, SFITAG isolates kernel extensions by forcing them to use MTE tag values different from that of the core kernel. To enforce all-around isolation from initiation to operation, both the SFITAG initializer and runtime collaborate to intervene in the execution of an extension at every step, and assign or update tag values of all memory objects and pointers associated with the extension. We emphasize that SFITAG satisfies all the design requirements specified in § 5.1. First, SFITAG meets **R1** by completely regulating accesses of kernel extensions to all types of kernel memory regions including heap, stack, and MMIO. Second, SFITAG satisfies **R2** with the help of ARM MTE that specializes in enforcing finer-grained memory access policies than conventional techniques for address space masking. Leveraging such an advanced hardware feature substantially decreases the always-on overhead for memory access that SFI solutions commonly impose on applying security policies. In addition, SFITAG achieves **R3** by employing optimization techniques to minimize the amount of code instrumentation.

In the followings, to clarify our MTE-supported SFI mechanism, we elaborate on how SFITAG manages MTE tag values regarding kernel memory objects (§ 5.3), stack (§ 5.4), MMIO (§ 5.5), and control flow (§ 5.6) for secure and efficient isolation of kernel extensions.

All our tag management tactics are executed by the SFITAG components, such as compiler, verifier, initializer and runtime, whose details will be explained in § 6.

5.3 MTE-supported Memory Isolation

As in § 4, we assume adversaries who can exploit any memory vulnerabilities and launch various types of attacks, such as data corruption and control hijacking attacks, within kernel extensions. Under such a strong assumption, SFITAG employs ARM MTE to isolate the extensions, and eventually to prevent the adversaries from accessing the core kernel at their disposal. To be specific, SFITAG enforces an invariant that extensions must be assigned designated tag values that are not assigned to the core kernel. Since ARM MTE prohibits memory accesses upon a mismatch between the memory and pointer tags, this invariant ensures extensions are blocked from touching the kernel data arbitrarily.

SFITAG implements the aforementioned invariant by assigning the predefined tag value, `0xF`, to the core kernel, and all the other different values to extensions. There are two reasons SFITAG adopted this tagging method. First, as explained in § 3, `0xF` is a special pointer tag value designated to allow unrestricted memory access in the kernel. Therefore, by assigning this tag value to the core kernel, SFITAG can enforce a memory access policy biased toward the core kernel over extensions. That is, the kernel can access the memory region for extensions, but not vice versa. The tag assignment method is also beneficial in terms of performance. Since the kernel occupies upper address space in Linux, all kernel pointers naturally hold the tag `0xF`, which implies that SFITAG can avoid explicit tag assignment operations to these pointers. To realize the tag assignment method even in any adversarial situations, SFITAG conducts some code analysis and instrumentation for managing memory and pointer tags, which are as follows.

Memory Tags. Adversaries with full control over kernel extensions may try to arbitrarily manipulate memory tag values, especially to access the core kernel. To thwart this, the SFITAG compiler and verifier ensure that extension binaries do not include any memory tag store instruction (*i.e.*, `stg`). This constraint is imposed for security reasons, but it may hinder extensions from accessing their own or sometimes core kernel's memory objects that are necessary in their operations unless these objects are given the same memory tags as the extensions. The SFITAG runtime addresses this contradictory situation by engaging a set of wrapper functions that manage memory tags on behalf of the extensions. For example, when extensions try to allocate memory objects, they can invoke the wrappers, which then handle the request through the kernel's memory allocators and in turn assign the allocated objects the tag values of the extensions. The wrappers also possess an authority to grant the extensions to access kernel objects. In this case, they temporarily change tag values of the target kernel objects to those of the extensions. Details are explained in § 6.1.

Pointer Tags. Adversaries who cannot control memory tags anymore may alternatively try to manipulate pointers' tag values for accessing out of the isolation boundary. An intuitive way to fend off this threat is to deprive the adversaries of control over pointer tags as with the case of memory tags. Unfortunately, this intuitive solution is not feasible in reality because there are many easy paths to modify pointer tags that are stored in the, publicly known, upper part of pointers. For example, pointers stored in memory can be overwritten by memory store instructions, and their values can be changed by pointer arithmetic instructions. Instead of this method, therefore, SFITAG employs a masking method to force the use of the designated pointer tags in all pointer dereferences within kernel extensions. The cases 1 and 2 of Table 1 show how the masking method is realized by instrumenting memory instructions in the kernel extension code. SFITAG first reserves a dedicated register ($\texttt{X}_{rsv}$) that the designated pointer tags of the extensions have already been written in the topmost bits. It then adds before every memory instruction a *bitfield move* instruction, which moves specific bits from the source register to the destination while leaving the other bits unchanged. The added bitfield move instructions copy the memory target address stored in a register ($\texttt{X}_m$) to the dedicated register while preserving its pointer tag. Lastly, SFITAG changes memory instructions to use the dedicated register as a target address register. To sum up, this masking method ensures that the dedicated register always holds the designated pointer tag and memory instructions only use this register for referring to a target address. As a result, in no way can adversaries perform a memory access using other than the designated pointer tag. Compared to the conventional address masking method of the previous SFI solutions, our method brings two major benefits. First, as shown in Table 1, our method minimizes the number of added instructions to just one, thereby reducing a lot of performance overheads. Second, our method provides finer-grained memory access policy thanks to ARM MTE so that it can even be applied to kernel extensions in which associated memory objects are highly scattered.

5.4 Extension Stack Isolation

As said in § 3, ARM MTE does not check memory instructions using the stack pointer (`SP`) as a base register. Therefore, if these stack memory instructions are abused through stack buffer overflow or code reuse attacks, our memory isolation policy described in § 5.3 between the core kernel and extensions can be bypassed. To tackle this problem, we opt for a simple but expensive strategy that applies the address masking method of previous SFI solutions to every stack memory instruction. For more efficiency, we add an optimization technique as follows when this method is applied. First, when an extension starts to run in the kernel, we have designed the SFITAG runtime to allocate a separate memory region for the extension's stack. Then we have modified the extension code to invoke the SFITAG runtime to check every instruction that sets the `SP` value (*e.g.*, `mov SP,` $\texttt{X}_0$). Since the extension and the kernel are now having their own stacks, the SFITAG runtime can ensure the integrity of the extension stack simply by checking bounds of the stack's pointer. Adversaries also may manipulate the `SP` with memory instructions (*e.g.*, `ldr` $\texttt{X}_0$`, [SP, #0x20]`) to access the kernel memory adjacent to the extension stack. To detect malicious attempts to misuse these `SP`-based instructions, SFITAG places two inaccessible redzone pages adjacent to the both sides of the execution stack. As the immediate field of the `SP`-related instructions are less than or equal to 12-bits, the maximum immediate value is 32760 which is the same with the redzone pages size. Therefore, any attempts of accessing outside the stack using the `SP`-based

Table 1: Instrumented instructions by the SFITAG compiler

Case	Original Instruction	Converted Instructions
1	ldr X_n, [X_m]	bfxil X_{rsv}, X_m, #0, #38 ldr X_n, [X_{rsv}]
2	str X_n, [X_m]	bfxil X_{rsv}, X_m, #0, #38 str X_n, [X_{rsv}]
3	ret	bfxil X_{rsv}, lr, #0, #38 ldr XZR, [X_{rsv}] br X_{rsv}
4	blr(br) X_n	bfxil X_{rsv}, X_n, #0, #38 ldr XZR, [X_{rsv}] blr(br) X_{rsv}

instructions are trapped by redzones. But adversaries still may be able to bypass redzones by exploiting *SP*-based arithmetic instructions that add/subtract a immediate value to/from the `SP`, *e.g.*, `sub SP, SP, #0x20`. If the immediate value is larger than the size of redzone pages, we insert check operations for the `SP`. Even more, adversaries who are able to subvert control flows can (1) modify the `SP` outside the extension stack by executing the arithmetic instructions repeatedly, and (2) execute a `SP`-based memory instruction. In this case, the malicious memory access will not be trapped by the redzone pages. To prevent this problem, the SFITAG compiler finds all code patterns that there are indirect jumps between `SP`-based arithmetic instructions and `SP`-based memory instructions. The compiler then adds bounds check instructions for the `SP` before the indirect jumps, detecting abnormal modification of the `SP`.

5.5 MMIO Access Control

The most common use case of kernel extensions is device drivers to handle interactions between devices and the kernel. Unlike many kernel components which only need to access main memory (*i.e.*, DRAM), device drivers need to have access to device memory as well. For such access, ARM provides the Direct Memory Access (DMA) or the memory-mapped I/O (MMIO) mechanism to allow drivers to access device memory. While DMA allows device drivers to access memory without using virtual memory, MMIO allows device memory to be mapped to kernel virtual address space, making device memory accessible with the same memory instructions as main memory. This means that SFITAG should be able to intervene in kernel extensions accessing device memory via memory instructions. Unfortunately, SFITAG cannot make use of ARM MTE for this purpose because MTE only provides access control for main memory. To overcome this limitation, SFITAG inevitably applies to memory instructions the conventional address masking method of the previous SFI solutions. Specifically, SFITAG performs the address masking operation for device memory as well as the preserving operation for pointer tags before every memory instruction in the extensions. However, it is worth noting that SFITAG can carry out these two operations with only a single bitfield move instruction. For this, we adjust the virtual address space for the kernel as follows. First, we map all device memory, which is by default mapped to a `vmalloc` region through `ioremap` function as in Figure 3-(a), to the high address of the kernel virtual address space by adjusting its base offset (Figure 3-(b)). Second, we change the linear mapping region to be placed within the bottom half of the kernel virtual address space by resizing the `vmalloc` region as in Figure 3-(b). Now, we

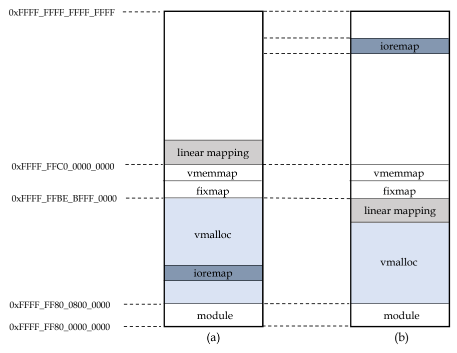

Figure 3: Kernel virtual address space layout. (a) shows the default layout, and (b) shows the changed layout

can distinguish the memory access for device memory and kernel memory by comparing the 38th bit in the address ('0' for the kernel memory and '1' for the device memory). As a consequence, by preserving upper 26 bits (including pointer tags in the bits 56 to 59) of the memory target address, we can enforce the memory instructions for the main memory not to access the device memory. On the other hand, we also regulate the memory operations for device memory to not to access the other device. In Linux kernel, the access to the device memory is carried out through invoking specific kernel functions (*e.g.*, readl and writel), so we check the boundaries of the device memory which are defined by the user using SFITAG APIs § 6.2 against the memory address passing to those kernel functions.

5.6 MTE-based Control Flow Isolation

Adversaries are capable of altering the control flow at runtime by manipulating the target address of indirect jumps and calls (*e.g.*, function pointers or return addresses) in extensions. With such a capability, they may attempt to execute the kernel code and the SFITAG runtime code. Since the code includes sensitive instructions, such as privileged instructions for memory management or memory tagging instructions, the adversaries can neutralize SFITAG by executing these instructions. To cope with this threat, SFITAG again utilizes ARM MTE as follows. SFITAG assigns designated tag values of extensions to legitimate indirect jump/call target addresses, such as entries of wrapper functions and extension functions, during the initialization of extensions. At run-time, SFITAG checks the validity of the tag value assigned to target addresses by executing a memory load instruction before indirect jumps/calls as shown in the case 3 and 4 of Table 1. Because code other than those legitimate ones is not assigned to the accurate tag values, any attempts of malicious indirect jumps/calls to subvert control flow will be caught. It is noteworthy that the load instructions for tag checking use the `XZR` register whose value is always zero as a destination register so that SFITAG can avoid possible side effects caused by the load instructions. However, as ARM MTE provides a memory tag at a 16-byte granularity, we should note that adversaries can execute arbitrary instructions within 16-byte code blocks of the legitimate target addresses. To mitigate this problem, SFITAG aligns instructions in the legitimate target addresses at a 16-byte boundary. Note that SFITAG does not apply this alignment policy to the extension, just to the SFITAG runtime that intervenes all control transfers between the extension and the kernel. Surely, the adversaries can also attempt to bypass the tag validity checks by skipping the execution of the added load instruction via any control-hijacking attacks in the extension. To overcome it, SFITAG again uses the method of reserving a dedicated register similar to § 5.3. Specifically, as described in Table 1, SFITAG makes only the dedicated register hold target addresses, thereby thwarting any attempts of arbitrarily modifying the target addresses to execute indirect jumps/calls bypassing the tag checks.

6 IMPLEMENTATION

As described in § 5.2, SFITAG consists of several components. In this section, we explain how these components are implemented and work.

6.1 SFITAG Runtime

The SFITAG runtime has wrapper functions to supervise all control transfers between the kernel and extensions. It also has utility functions for tagging operations and tag checking operations (*i.e.*, TAG and CHECK). The wrappers can be divided into two categories. One includes functions for intervening when the kernel enters the isolated domain of an extension (K2E wrappers). The other includes those for when an extension invokes kernel functions (E2K wrappers). Figure 4 shows the workflow of these wrappers.

The role of these wrappers is to support domain switches in SFITAG. Specifically, K2E wrapper assigns a tag value for the called extension to X_{rsv} register exclusively reserved for holding a tag value and changes SP register to point to the execution stack. Then, K2E wrapper performs memory tagging operations, thereby allowing an extension to access passed arguments while running extension functions. Note that how to assign memory tags to memory objects that are used in the extension function like arguments is configured by the developers using SFITAG APIs (see § 6.2). After that, the extension function is called in K2E wrapper. Once the extension function is returned, memory tags for arguments are changed to hold `0xF` that is used for the kernel. And X_{rsv} register is cleared and SP register is changed to point the kernel stack.

In the case of E2K wrapper, similar to K2E wrapper, the values of X_{rsv} register and SP register are configured depending on whether the extension code is executed or the kernel code is executed. However, memory tags for the arguments of the kernel function are not changed before calling the kernel functions. Because the kernel is working with `0xF` as the value of pointer tag and this pointer tag can access the any memory objects regardless of the value of the memory tag of them. Instead, the E2K wrapper performs an operation that checks whether the extension function passes arguments having the tag value dedicated for the extension. Note that, when the extension tries to run kernel functions for MMIO accesses, the wrappers check if the passed pointers are within legal boundaries, instead of checking tags. On the other hand, the tag operation is performed to the memory object returned to the extension because this object should be accessible in the extension.

6.2 SFITAG Compiler

We provide four APIs for developers to relieve their efforts of integrating the extension with our system, as follows:

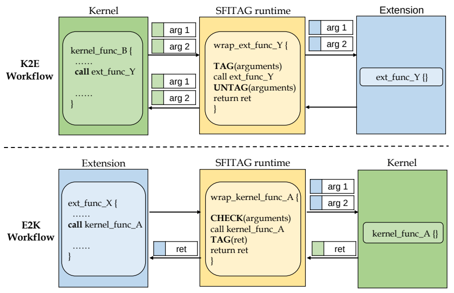

Figure 4: Workflow between the kernel and an extension

setTag(ptr, size, tag, depth). This API allows developers to assign a memory tag, *tag*, to the memory region [`ptr, ptr + size`), granting or revoking access to objects used in extension functions. For example, in Listing 1, Lines 7-8 and 23-25 show the code that grants and revokes access to the arguments (*i.e.*, skb, dev) of the stmmac_xmit function, respectively. Note that they use different tag values as arguments of the API call (`0x1` at Lines 7-8 and `0xF` at Lines 23-25). This API is also used when the extension invokes a kernel function. At Line 11, the extension invokes `netdev_priv` function and receives the pointer variable (*i.e.*, `struct stmmac_priv`) as a return value. Before invoking the function, SFITAG checks that dev has the correct tags (Line 10). To grant access to the memory object that the pointer points to in the extension function, `setTag` is used in Line 12. On the other hand, in Line 18, as `des` is a local variable and is stored in the extension stack, `setTag` is not called. SFITAG also provides `depth` as a parameter that enables our tagging mechanism to conveniently assign a memory tag to a multi-level object which contains a set of member objects aggregated in a hierarchical structure with one root member. Given the `depth` value *n*, SFITAG gives tags to all member objects down to level *n* in the hierarchy from the root at level 0. This is surely convenient for developers as they can tag a group of objects at once without individually annotating each object in the group. For example, Line 7, 12, 23 and 25 show that the developer tags objects with `depth` 1.

checkTag(ptr, size, tag, depth). This API checks if a memory tag of the memory region [`ptr, ptr + size`) matches `tag`. For example, arguments are checked to have appropriate access rights before invoking a kernel function in Line 10 and Lines 16-17. Likewise `setTag`, this API also has a `depth` parameter, so the developer can check the tags of multiple member objects in the multi-level object at once.

setBound(addr, size). This API sets the bounds of MMIO region for the extension to [`addr, addr+size`) as explained in § 5.5.

checkBound(addr). This API checks if `addr` falls within the bounds established by `sfitag_set_bounds`. Developers can insert APIs before MMIO access functions (*e.g.*, readl, writel) as in Line 33. These annotations are used to generate wrapper functions by adding appropriate operations to the prologue and epilogue of each function according to the developer's directions conveyed in them. Those additional instructions are mostly for tagging operations which will be executed by wrappers in the SFITAG runtime. For example, the wrapper for function `netdev_priv` is augmented with tagging operations of the returned object `priv`. Likewise, tag-checking operations are added before the call site for the kernel function in the wrapper of `wrap_dma_map_single` due to the annotations in Lines 16-17.

6.3 Extension Loading Procedure

When an extension is installed in the kernel, the following procedures are carried out for the SFITAG to operate normally at runtime. First, the tag manager in the kernel allocates a tag value for the extension and registers the value to its own table. Through this tag registration process, we can prevent newly installed extensions from either accidentally or maliciously sharing the same tag value with running extensions. Next, while loading the extension, the SFITAG runtime performs a series of tagging operations as follows; (1) allocating tagged memory pages for the extension stack and making all adjacent pages around it inaccessible, (2) assigning memory tags for valid jump targets to enforce control flow isolation at runtime, and (3) assigning the memory tag to the extension's private objects to reduce tagging operations at runtime.

6.4 SFITAG Verifier

We provide the SFITAG verifier to assure that the extension is correctly isolated. The verifier scans the binary file of the extension to check if the following invariants are enforced. The first one is that there are no unsafe instructions, such as privilege instructions and memory tagging instructions which may deprive the protection of SFITAG. One such instruction is a privileged instruction that can disable MTE in the system. Another is an MTE instruction that can be used to bypass SFITAG by changing the memory tags in the extension. The second invariant is that the SFITAG instrumentation is correctly applied to all memory and indirect jump instructions. Note hereby that the verifier should ensure no instruction to take X_{rsv} as its operands except the instrumented memory and indirect jump instructions. The next invariant is that there is no instruction assigning arbitrary values to SP register without sanitization check. Finally, the last is that the target addresses of all direct jumps remain within the code region for the isolated extension or the address of wrapper functions.

6.5 Multithreading Support

Today, most OS kernels support multi-threading, so we designed SFITAG to support the multi-thread as well. First, since there is one memory tag per physical memory not per thread, we have to consider the race condition that multiple threads try to access the same memory tag. For example, assume that there are two functions, A and B, and they use different tag values, *i.e.*, *tag_a* and *tag_b*, to access the shared kernel data K. When function A runs first, kernel data K is tagged by *tag_a*. Then, if function B is invoked on the different cores simultaneously before function A is over, function B would try to assign the kernel data K with *tag_b*. Consequently, this change makes function A unable to access the kernel data K since the memory tag and pointer tag are not matched. To deal with this, we have relied on the coding patterns in the extension for the shared kernel data. When developers implement a kernel extension, they apply the synchronization mechanism of OS kernel to the memory access which can cause a race condition. In other words, we put the code that checks the memory tag and performs the tagging operation only to the memory operations in the critical sections defined by the synchronization mechanism. Also, in order to handle the case when multiple threads are working with one kernel extension at the same time, we assign a separate extension stack for each thread. In specific, SFITAG allocates the extension stack in the thread local storage of each thread. And, when the thread invokes the extension function, SFITAG wrapper makes the stack pointer point to its extension stack. After the function is over, the stack pointer is changed to point to the original interrupt stack. We disable the nested interrupt feature to prevent that the execution stack is corrupted by executing multiple extensions in one thread simultaneously.

```
1  static netdev_tx_t stmmac_xmit
2        (struct sk_buff *skb, struct net_device *dev) {
3
4   unsigned int des;
5   unsigned int len = skb_headlen(skb);
6
7   setTag(skb, sizeof(struct sk_buff), 0x1, 1);
8   setTag(dev, sizeof(struct net_device), 0x1, 0);
9
10  checkTag(dev, sizeof(struct net_device), 0x1, 0);
11  struct stmmac_priv *priv = netdev_priv(dev);
12  setTag(priv, sizeof(struct stmmac_priv), 0x1, 1);
13
14  /* code execution */
15
16  checkTag(priv->device, sizeof(struct net_device), 0x1, 1);
17  checkTag(skb->data, sizeof(struct sk_buff), 0x1, 1);
18  des = wrap_dma_map_single(priv->device, skb->data,
19                                  len, DMA_TO_DEVICE);
20
21  /* code execution */
22
23  setTag(skb, sizeof(struct sk_buff), 0xf, 1);
24  setTag(dev, sizeof(struct net_device), 0xf, 0);
25  setTag(priv, sizeof(struct stmmac_priv), 0xf, 1);
26
27  return NETDEV_TX_OK;
28 }
29
30 void dwmac_enable_dma_transmission
31                            (void __iomem *ioaddr)
32 {
33     checkBound(ioaddr + DMA_XMT_POLL_DEMAND);
34     writel(1, ioaddr + DMA_XMT_POLL_DEMAND);
35     return;
36 }
```

Listing 1: An example of the execution code using SFITAG APIs

7 EVALUATION

We conduct all experiments on the ODROID-C4 [5] development board with four 2.0GHz quad-core ARMv8-A based Cortex-A55 processors and 4GB RAM. We use 64-bit Ubuntu 20.04 Linux with kernel version 4.9.236 as an OS. All instrumentations are applied using the LLVM 9.0 compiler framework [18].

7.1 MTE-Analogue

At the time of writing, there has been no line of ARM processors that are equipped with MTE. We verified the functional correctness of SFITAG by implementing a prototype on ARM Fast Models [2], a software emulator including MTE. However, the emulator does not provide cycle-accurate execution. Thus, to estimate the performance overhead introduced by using MTE, we have devised

```
mov    X_rsv ,   XZR
bfxil  X_rsv ,   X_src,     #0,     #38        ; memory offset masking
movk   X_t ,     #tag mem0,   lsl #0
movk   X_t ,     #tag mem1,   lsl #16
movk   X_t ,     #tag mem2,   lsl #32
movk   X_t ,     #tag mem3,   lsl #48
add    X_t ,     X_t ,  X_rsv,   lsr #4
ldr    X_rsv ,   [X_t , #8]                    ; tag load
str    X_src ,   [X_rsv , #8]
```

Figure 5: Tag Load Instrumentation. X_{rsv} is the reserved register for memory address with fixed tag, and X_t is a register for tag memory address

Table 2: Code instrumentation overhead

Drivers	*wrappers*	*MI*	*CFI*	X_{rsv}	Δ *codesize*
nullnet	356	100	22	0	546
stmicro	8809	4574	455	3	13841
nullblk	694	374	36	0	1176

MTE-analogue. Most of the performance overhead in MTE comes from (1) memory tagging operation and (2) tag checking operation. Therefore, we mimic these two types of operations in software to estimate the MTE overhead. Therefore, we first prepare a reserved memory region in the kernel address space to emulate the tag memory and estimate tagging overheads by executing substitutive memory instructions that write single bytes to the reserved memory. Next, we emulate MTE instruction which includes loading and comparing tags. Fortunately, the tag comparing operation is performed by separate hardware logic that runs concurrently with the CPU core, resulting in negligible overhead [4]. To estimate tag loading overhead, SFITAG inserts an extra load instruction before every load and store, it performs comparing the pointer tag value with the target memory in 16-byte granularity. As a result, SFITAG enables to emulate the precise mode which supports immediate detection of tag mismatch.

7.2 Code Instrumentation Overhead

To measure such impact on code size, we collect the following statistics from the execution code, as seen in Table 2. *MI* shows the number of *bitfield move* instructions before every memory instruction in the extension. Note that, in *MI*, we do not include instructions described in MTE-analogue since these instructions are not required in the MTE-enabled platform. *CFI* denotes the total number of added instructions to every indirect jump. X_{rsv} shows the code size overhead when one general-purpose register is not used in the extension. As a result, *code size* shows the code size overhead when SFITAG is applied to the extension. Note that, in this case, we do not include the code size for wrapper functions since it can vary according to the developers. Instead, we count the number of wrapper functions (*i.e.*, K2E and E2K wrappers) for each extension, as shown in *wrappers* in Table 2.

7.3 Load Time Overhead

As explained in § 6.3, when loading an extension, SFITAG performs memory tagging operations to bring valid isolation of the extension at runtime. Although the number of tagging operations would differ from extensions and how SFITAG APIs are applied, under our experimental setting, the one that most affects the performance is the tagging operations for the extension stack (8KB). Nevertheless, the loading time is slowed down to less than 0.1s in all extensions we used in the evaluation.

Table 3: Cost of domain crossing for SFITAG

Micro operations	X_{rsv} *setting*	*stack switching*	*setTag*	*checkTag*
Cycles	130	122	82	202

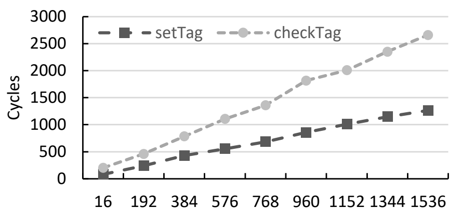

Figure 6: Tag overhead according to the size of object (Bytes)

7.4 Micro Operations Overhead

To analyze the performance impact of SFITAG in detail, we measure the cycles separately which are measured by using `perf_event_open` system call to read PMU event counters. Specifically, the system call is configured to count hardware CPU cycles by setting `HW_CPU_CYCLES` attribute. All reported cycles are averaged over 100,000 runs. Table 3 shows the minimum cost of micro-operations that should be performed each time domain crossing occurs. X_{rsv} *setting* is the operation performed to ensure that a dedicated register always holds the specified pointer tag, which takes 130 cycles. *Stack switching* is always performed in the function prologue and epilogue, and represents the cost of the kernel and each extension stack transitions. Since the MTE assigns or checks the tag at 16-byte granularity, *setTag* and *checkTag* show the number of cycles about the tag granule. *setTag* is the operation that assigns a tag to a memory location by adding 4 bits of metadata to every 16 bytes of physical memory, taking a total of 82 cycles. *checkTag* takes 202 cycles because it loads the corresponding memory tag from dummy tag memory and performs the tag comparison before each memory access. As shown in Figure 6, *setTag* and *checkTag* indicate that cycles increase linearly with the size of the shared object. Fortunately, it can be optimized by tagging only the fields of the structure that are actually shared with the API provided by SFITAG, and our experiments have confirmed that performance is not significantly affected, as will be explained later.

7.5 Device Drivers

To evaluate SFITAG, we instrument three device drivers in the Linux kernel to be isolated. We choose the network and block device drivers since they utilize kernel subsystems with the tightest performance budgets. `nullnet` and `nullblk` emulate infinitely fast devices in software. While they fail to fully reflect the complex nature of real device drivers due to the lack of hardware device related interfaces, their simplicity allows us to stress the default overheads (*e.g.*, SFI overhead, wrapper overhead, tag-related operations overhead) of SFITAG without any artificial hardware limits. On the other hand, `stmicro` driver allows us to explore several

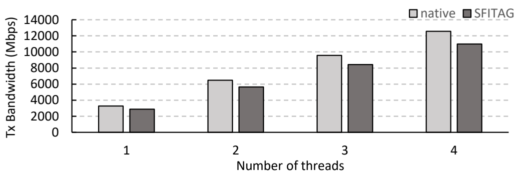

Figure 7: Performance of the `nullnet` driver

optimization techniques for SFITAG when applied to real-world drivers due to its complex interfaces for the kernel and the hardware device.

7.5.1 Dummy Network Driver (nullnet). We employ the `nullnet` device driver as a representative example, which represents an infinitely fast device and is accessed through a highly optimized I/O submission path in the kernel network stack. We use the *iperf3* benchmark, a network traffic tool for measuring transmit and receive bandwidth for different payload sizes. We configure the `nullnet` driver with a varying number of threads from 1 to 4, and report the UDP transmit bandwidth on the maximum transmission unit size packets averaged across ten runs. In our experiments, we configure `nullnet` to annotate objects specific to its use rather than leveraging the `depth` parameter in our provided APIs. While `depth` parameter provides simple mechanism to tag subobjects in a comprehensive way when it is difficult to track down numerous subobjects to annotate them respectively, we observe that `nullnet` only consists of few lines of code that merely updates several statistic related fields of objects, allowing us to annotate each of the specific use-cases of objects within the `nullnet` driver. This configuration allows us to analyze the isolation overheads of SFITAG in the ideal scenario where no instructions are wasted to tag (and un-tag) unused subobjects. Figure 7 shows the results. With one application thread, the non-isolated driver (native) achieves 3281 Mbps. The SFITAG-based isolated driver (SFITAG) achieves 2880 Mbps (87.7% of the native performance). Performance scales linearly as we increase the number of threads for all drivers: SFITAG-based isolated drivers achieve on average 87.69% of the native driver across 1-4 threads.

7.5.2 Stmicro Network Driver. We use an *iperf3* test for TCP transmit and receive bandwidth, varying the number of iperf threads ranging from 1 to 4 (Figure 8). We configure SFITAG to run several configurations to highlight performance gain from different optimizations that SFITAG provides: 1) *SFITAG-naive*: the driver is annotated with a maximum number of dereferences for every root object utilized within a driver function. If the maximum number of dereferences (depth) of an object is 4, every sub-object accessed through 1 to 4 dereferences of root object (with a depth of 1 to 4) is tagged regardless of its actual use case. As explained in § 6, this approach of implementation illustrates simple, one-lined annotations for root objects without having to go through the whole function to track down sub-objects. 2) *SFITAG-opt1*: the driver is annotated specifically, going through the functions and only annotating the sub-objects in use by the driver. 3) *SFITAG-opt2*: the annotations related to driver private objects are removed. This optimization comes from our observation that these objects must always be accessible by the driver (*i.e.*, not requiring changes in access permission at runtime) and are allocated by the driver functions during initialization. With such characteristics, after being tagged on the initial

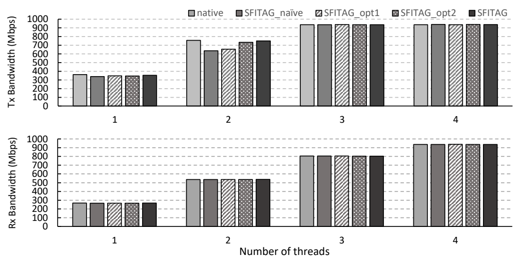

Figure 8: `stmicro` network driver Tx/Rx bandwidth

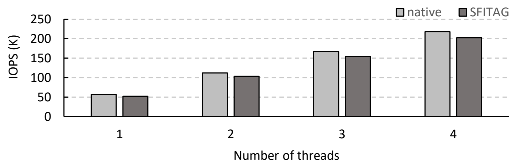

Figure 9: `nullblk` driver for packet size of 512B

allocation, driver private objects can avoid excessive (re)tagging and untagging on its use. For example, driver private objects are allocated to a private heap. 4) *SFITAG*: the driver is annotated with full utilization of optimizations stated above. On SFITAG, a small number of application threads saturates a 1Gbps network adapter. For all experiments, we use the 64KB packet size to maximize the domain crossing overhead. On the transmit path, the isolated device driver before optimizations (*SFITAG-naive*) suffers from an average of 5.6% degradation in throughput compared to the native driver. In the case of *SFITAG-opt1* and *SFITAG-opt2*, the throughput drops by 4.4% and 1.9%, respectively. When both optimizations are applied (*SFITAG*), the isolated driver saturates the network interface, so it causes about 1% drop in the throughput of native driver. On the receive path, isolated drivers without optimizations (*SFITAG-naive* record 3% lower performance than the native driver in terms of throughput. In the case of *SFITAG-opt1* and *SFITAG-opt2*, the throughput drops by 1.9% and 1.1%, respectively. After both optimizations are applied, there is a 0.8% performance degradation in throughput compared to the native driver. According to our analysis, the reason for more throughput reduction for transmit paths than receive paths in our technique is that SFITAG APIs (*i.e.*, setTag and checkTag) are called more during the transmit path than the receive path. In addition to throughput, we also measure CPU utilization when packets are transmitted and received. When saturating a 1Gbps network adapter for TCP connections, SFITAG consumes 12% additional CPU usage which is the average of 16% for the transmit path and 9% for the receive path. As described above, we need more tagging operations for the transmit path than for the receive path, and consequently, SFITAG consumes more CPU cycles for the former case than the latter. All in all, our experimental results give us a general rule of thumb that the more tagging operations, the higher CPU utilization as well as the lower throughput.

7.5.3 Null Block Device Driver (nullblk). Similar to the `nullnet` driver, the `nullblk` does not interact with any real physical device(NVMe), thus, it is possible to emulate with pure-software. Also, we can annotate each object specifically rather than using the `depth` parameter for the same reason as the `nullnet` network driver. In `nullblk`, I/O requests are generated using the fio benchmark. To evaluate our experiments, we set the optimal baseline by configuring parameters that can provide the lowest latency path to the extension. In addition, we vary the number of threads from 1 to 8 and used two block sizes, 512B and 1MB. Since the `nullblk` driver does not communicate with the real hardware device, we only employ read I/O requests. For the packet size of 512B and a single thread, the native driver achieves 57K IOPS and for SFITAG it achieves 52.6K IOPS, which is 92.28% of performance compared to native (Figure 9). Also, the native driver with the packet size of 1MB has 7427 IOPS, whereas the SFITAG achieves 7141 IOPS, resulting in 96.15% performance compared to the native driver (Figure 10).

7.6 Security Analysis

Data manipulation attacks. As noted in § 4, an adversary may try to make any arbitrary access to the kernel data by exploiting memory vulnerabilities in extensions. However, in SFITAG, none of memory instructions within extensions can be used to achieve this purpose. First, in the case of the memory instructions for the main memory, SFITAG forces the upper 26-bits of memory target address including the pointer tag to hold dedicated value, so these instructions only can access the memory region with the same memory tag value. In other words, the adversary cannot access kernel data that holds a different memory tag value from the one for the extension. Moreover, SFITAG does not allow the extension code includes any special instructions that can assign or update memory tag values. The adversary may exploit memory instructions for the device memory. For this, SFITAG performs bounds-checking operations on the target memory address of these operations to prevent to access main memory for device memory for other extensions. Therefore, an attacker cannot perform data manipulation attacks beyond the isolation boundary.

Control hijacking attacks. An adversary also can launch control hijacking attacks while executing the extension code because the adversary is capable of modifying code pointers dwelling in the extension. However, even if the adversary manipulates the control flow of the extension, only limited code gadgets within the extension can be exploitable since SFITAG only allows control transfers to legitimate targets, such as function entries and basic block entries in the extension code. Also, the adversary can try to invoke wrapper functions of SFITAG runtime which are responsible for security-critical operations such as memory tagging. To prevent abuse of these wrappers, SFITAG prevents malicious jumps to the middle of the wrapper function even if an attacker manipulates the control flow of the extension. In addition, this enforcement allows us to verify whether the arguments are benign or not by putting this check code at the beginning of the function. For example, SFITAG can check if the passed pointer argument actually points to the memory region for the extension by confirming the memory tag value. Additionally, during running extension code and wrapper functions, SFIKE disables subsequent interrupts to counteract attacks that maliciously trigger interrupts.

8 DISCUSSION

Limitation on the number of tags. As noticed earlier, ARM MTE provides just four unused bits to form the tag section, implying that SFITAG in principle is only allowed to have at maximum 14 isolated extensions assigned distinct tags, in addition to two remaining tags reserved. This means that if we try to isolate one more extension here, we have no more free tags left for the extension. A naive design that we can consider to handle this tag shortage problem would be assigning a newly installed extension the same tag already given to running extensions. Although this may solve the problem, it can cause a security loophole that allows a misbehaving extension to make arbitrary access to others assigned the same tag. As a quick remedy, we may reduce the loophole by following the design suggested by BGI [9] where mutually trusted extensions, such as those made by the same manufacturer, are placed into one domain, which effectively saves the use of tags by sharing one tag among multiple extensions in a safer way. An alternative design that we may consider is to opt for a hybrid approach where we combine SFITAG and BGI. In this design, we basically use the BGI scheme for kernel isolation because it enables us to isolate as many extensions as we need to. However, as described in § 2, BGI cannot prevent the read access to the kernel, so we may apply SFITAG to security critical extensions (up to 13) and BGI to the others. SFITAG and BGI can work independently in the system at the same time because both the schemes operate respectively on separate data structures, the pointer tags and ACL.

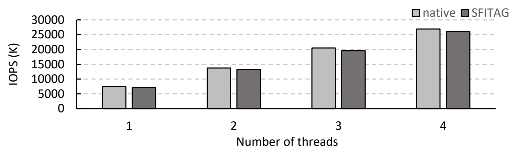

Figure 10: `nullblk` driver for packet size of 1MB

Limitation on minimum alignment granularity. When having kernel objects allocated for isolated extensions, SFITAG should pay special attention to a case where the size of an object for tagging is smaller than Tag Granule, 16-bytes. Since MTE provides a tag for each 16-byte granule in the physical address space, memory objects smaller than 16 bytes will receive the same tag value if they both fit inside the same granule. A simple cure for this would be aligning small objects to 16 bytes. But, this is impractical in that the kernel code must be modified and compiled every time an extension is added. Alternatively, we handle this case by selectively enforcing bounds checking when accessing certain objects. In our method, SFITAG allows a developer to select and annotate a specific object for bound checking. Later, just before the object is accessed, SFITAG checks if the access falls within the legal bounds. From our experiments, we have observed that this method is a practical solution for implementing fine-grained isolation of small objects of any size.

9 CONCLUSION

SFITAG seeks the memory tagging hardware support to deliver boosts in the performance of traditional software implementations of SFI for kernel extension isolation on ARM. With the aid of ARM MTE, isolated extensions are assigned different tag values from the kernel, and all their accesses are checked dynamically by hardware to prohibit illegal reads/writes to the memory regions belonging to the core kernel as well as other isolated extensions with different tag values. As evinced in our experiments, the MTE-supported runtime access checks are done fast with low overhead, which helps

SFITAG to enhance the overall performance. As another advantage of SFITAG, we have shown that our solution offers kernel isolation with higher precision by providing finer-grained protection.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. This work was supported by the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2023, and Inter-University Semiconductor Research Center (ISRC). Also, the research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT (Ministry of Science and ICT)) (No. NRF-2022R1A4A1032361), Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone). Finally, this work was supported by the MSIT, Korea, under the Convergence security core talent training business (Pusan National University) support program (IITP-2023-2022-0-01201) supervised by the IITP, and IITP grant funded by the Korea government (MSIT) (No.2021-0-00724, RISC-V based Secure CPU Architecture Design for Embedded System Malware Detection and Response).

REFERENCES

[1] 2001 (accessed May 20, 2021). *ARM domain access control*. https://developer.arm.com/documentation/ddi0198/e/memory-management-unit/domain-access-control
[2] 2019 (accessed January 12, 2021). *Fast Models*. https://developer.arm.com/tools-and-software/simulation-models/fast-models
[3] 2019 (accessed January 12, 2021). *Linux Kernel : Security Vulnerabilities Published In 2019*. https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2019/Linux-Linux-Kernel.html
[4] 2019 (accessed January 12, 2021). *Memory Tagging Extension: Enhancing memory safety through architecture*. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf
[5] 2019 (accessed January 12, 2021). *ODROID C4*. https://www.odroid.co.uk/index.php?route=product/product&product_id=1027
[6] Godmar Back and Wilson C Hsieh. 2005. The kaffeos java runtime system. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 4 (2005), 583–630.
[7] Silas Boyd-Wickizer and Nickolai Zeldovich. 2010. Tolerating Malicious Device Drivers in Linux.. In *USENIX annual technical conference*. Boston.
[8] Shakeel Butt, Vinod Ganapathy, Michael M Swift, and Chih-Cheng Chang. 2009. Protecting commodity operating system kernels from vulnerable device drivers. In *2009 Annual Computer Security Applications Conference*. IEEE, 301–310.
[9] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. 2009. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 45–58.
[10] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*. 1–5.
[11] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. 2006. XFI: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 75–88.
[12] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, Mark Williamson, et al. 2004. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*. Boston, USA;, 1–1.
[13] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2022. {KSplit}: Automating Device Driver Isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 613–631.
[14] Galen C Hunt and James R Larus. 2007. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review* 41, 2 (2007), 37–49.
[15] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1868–1882.
[16] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*. 437–452.
[17] Igor Korkin. 2018. Divide et Impera: MemoryRanger Runs Drivers in Isolated Kernel Spaces. *arXiv preprint arXiv:1812.09920* (2018).
[18] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86.
[19] Valentin JM Manès, Daehee Jang, Chanho Ryu, and Brent Byunghoon Kang. 2018. Domain Isolated Kernel: A lightweight sandbox for untrusted kernel extensions. *computers & security* 74 (2018), 130–143.
[20] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 115–128.
[21] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, et al. 2019. LXDs: Towards isolation of kernel subsystems. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 269–284.
[22] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 21–39.
[23] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2020. Lightweight kernel isolation with virtualization and VM functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 157–171.
[24] Olatunji Ruwase, Michael A Kozuch, Phillip B Gibbons, and Todd C Mowry. 2014. Guardrail: A high fidelity approach to protecting hardware devices from buggy drivers. *ACM SIGPLAN Notices* 49, 4 (2014), 655–670.
[25] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. 2009. Dingo: Taming device drivers. In *Proceedings of the 4th ACM European conference on Computer systems*. 275–288.
[26] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. 2009. Automatic device driver synthesis with Termite. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 73–86.
[27] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. 2016. HDFI: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–17.
[28] Yifeng Sun and Tzi-cker Chiueh. 2013. SIDE: Isolated and efficient execution of unmodified device drivers. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 1–12.
[29] Michael M Swift, Brian N Bershad, and Henry M Levy. 2003. Improving the reliability of commodity operating systems. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 207–222.
[30] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*. 203–216.
[31] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. 2015. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 20–37.
[32] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. {FUZE}: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 781–797.
[33] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 79–93.
[34] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2008. Hardware Enforcement of Application Security Policies Using Tagged Memory.. In *OSDI*, Vol. 8. 225–240.
[35] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 558–569.