TIDE: An Efficient Kernel-level Isolation Execution Environment on AArch64 via Dynamically Adjusting Output Address Size

Shiyang Zhang
State Key Lab of Processors, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
Zhongguancun Laboratory
Beijing, China
zhangshiyang21s@ict.ac.cn

Chenggang Wu
State Key Lab of Processors, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
Zhongguancun Laboratory
Beijing, China
wucg@ict.ac.cn

Chengxuan Hou
State Key Lab of Processors, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
houchengxuan16@mails.ucas.ac.cn

Jinglin Lv
State Key Lab of Processors, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
Zhongguancun Laboratory
Beijing, China
lvjinglin22@mails.ucas.ac.cn

Yinqian Zhang
Southern University of Science and Technology
Shenzhen, China
yinqianz@acm.org

Qianyu Guo
Zhongguancun Laboratory
Beijing, China
guoqy@zgclab.edu.cn

Yuanming Lai
State Key Lab of Processors, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
laiyuanming@ict.ac.cn

Mengyao Xie
State Key Lab of Processors, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
xiemengyao@ict.ac.cn

Yan Kang
State Key Lab of Processors, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
kangyan@ict.ac.cn

Zhe Wang*
State Key Lab of Processors, Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
Zhongguancun Laboratory
Beijing, China
wangzhe12@ict.ac.cn

Abstract

To enforce the privilege separation in the kernel, kernel-level isolated execution environment (IEE) has become a recent research trend because it can protect critical resources and monitors. Our research found that to isolate the IEE memory, all existing IEEs must act as a reference monitor to isolate page tables and validate their updates, bringing a significant performance overhead. Hence, we propose TIDE, a new kernel-level IEE based on the output address size hardware feature on AArch64, which could offload such checks to the hardware. However, it still faces the flexibility and security challenges. To address them, TIDE presents using the stage-2 translation to expand the physical address range to flexibly map the IEE memory and perform extra access controls on the physical memory; it designs a novel gate to enter (sneak) into the IEE securely by disabling translation temporarily, and ensures it can only be executed at the fixed locations. The experimental results show that TIDE is performant than all existing IEEs on protecting critical kernel structures and security tools.

*Zhe Wang is the corresponding author.

CCS '25, Taipei

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765111

CCS Concepts

• **Security and privacy → Systems security**.

Keywords

IEE, Output Address Size, Translation, Virtualization, AArch64

ACM Reference Format:

Shiyang Zhang, Chenggang Wu, Chengxuan Hou, Jinglin Lv, Yinqian Zhang, Qianyu Guo, Yuanming Lai, Mengyao Xie, Yan Kang, and Zhe Wang. 2025. Tide: An Efficient Kernel-level Isolation Execution Environment on AArch64 via Dynamically Adjusting Output Address Size. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765111

1 Introduction

The security of an operating system's kernel is fundamental to a computing environment's overall integrity and resilience. Since the kernel usually operates with the highest privilege level, a compromise at this level can lead to complete system control by an attacker [8, 25, 30, 48, 60, 62]. However, the kernel's large and complex codebase presents significant security challenges. As of January 2025, the Linux kernel's source code comprises approximately 40 million lines [41], and thousands of vulnerabilities have been exposed in the past five years [12].

Privilege separation, initially introduced in the work of Saltzer and Schroeder [47], is recognized as a fundamental principle in software design to address the concern. To enforce it, a common practice is to protect the kernel's critical parts. This can not only be done by isolating them [9, 14, 26, 31, 35, 57, 63], but also by monitoring their execution [3, 7, 11, 15, 53]. Both methods need an isolated execution environment (IEE) to run them or the monitor.

An intuitive approach is to run the kernel critical parts [4, 34] or the monitor [43, 44, 50, 51] at a more privileged level, e.g., in hypervisor mode. But it will significantly increase the hypervisor's Trusted Computing Base (TCB) and introduce a significant performance overhead because the traps are expensive. Running them in another virtual machine (VM) could avoid affecting TCB but will introduce a higher overhead. To address this problem, the recent research trend is *intra-kernel isolation*, which partially de-privileges the kernel of running at a (logically) lower kernel privilege level and lets the IEE run at the whole kernel privilege level [7, 11, 13, 24, 28, 53, 54, 63].

With ARM processors' dominant position in the mobile market and the increasing popularity of ARM on personal computers and servers, the kernel-level IEE on ARM is necessary for kernel security. It constructs a one-way memory isolation in which the kernel cannot access the IEE's memory, but the IEE can access the entire system's memory. Entering and exiting the IEE must be achieved through a specific gate, which can enable and disable access to the IEE memory. To achieve lower performance overhead, the design of IEE is based on ARM hardware features.

Since performance overhead is the primary factor restricting the deployability of IEE, we analyze the performance overhead of existing IEEs. At first, we systematically summarize three security properties required in a generic IEE (§4.1): 1) *The gate should be secured so it cannot be bypassed*; 2) *The physical memory of the IEE should be isolated* to ensure the IEE memory can only be translated in the IEE; 3) *The kernel $W \oplus X$ should be enforced* to avoid injecting the sensitive privileged instructions to enable the access to the IEE.

For the first property, existing IEEs propose to use different hardware features to switch the access permission to the IEE, i.e., `TTBRn` [7], `TxSZ` [11], `PAN` [28], and `Watchpoint` [24]; for the last two properties, they all isolate page tables, mediate their updates and perform the validation to ensure the IEE memory cannot be mapped to the outside virtual space. Then, we conducted the composition analysis of performance overhead on existing IEEs. The page table protection incurs the primary performance overhead (accounts for 60%~80% of the total overhead) because it results in frequent IEE invocations (§4.2). It motivates us to design an efficient kernel-level IEE, named Tide, to avoid mediating page table updates and directly offload these checks to the hardware.

The key hardware feature used in Tide is *the output address size checking during the page table walk*. The output (physical) address obtained in this process cannot exceed the output address size specified by the `{I}PS` field of the `TCR_ELx` register. Therefore, we could place the IEE memory outside the kernel's allowed output address size. And the hardware ensures that all page tables in the kernel cannot translate the IEE memory. The entry gate can set the `TCR_ELx.{I}PS` field to enlarge the size to enable the translation to the IEE memory. However, leveraging this feature to build Tide is non-trivial; it still faces flexibility and security challenges:

- *Configuring the output address size with more flexibility.*. The output address size has only seven sizes available, i.e., 4GB, 64GB, 1TB, 4TB, 16TB, 256TB and 4PB, and the adjacent sizes vary greatly. This results in an inflexible division of the IEE and kernel physical memory, which in turn wastes physical memory.
- *Kernel $W \oplus X$ still needs mediating page table updates.* The output address size feature only checks the virtual-to-physical mappings, not the access permissions required by the kernel $W \oplus X$.
- *Securely exposing the IEE memory in the gate.* The `TCR_ELx` register contains all address translation controls. Its value set in the gate can be tampered with by launching the jump-in-middle attack, hijacking the subsequent control flow. This allows the IEE memory to be exposed without entering the IEE.

Our key insight is that *the stage-2 translation could provide the opportunity for independent memory mappings and permissions checking in addition to the output address size checking in the stage-1 translation.* Hence, it can address the first two challenges independently of the stage-1 translation. Tide activates the stage-2 translation to introduce the intermediate physical address (IPA) between the virtual address (VA) and the physical address (PA). To address the first challenge, Tide uses the more flexible mapping provided in the stage-2 page table by mapping the physical memory into a larger IPA space. This allows the IEE and regular memory to be placed in different intermediate memory slots, which are split by available output address sizes in the stage-1 translation. Specifically, Tide maps the IEE memory to an arbitrary IPA exceeding the output address size that the kernel can use in the stage-1 translation; To address the second challenge, Tide uses the write-protection and finer-grained execution control provided in the stage-2 page table. Specifically, Tide sets the kernel code's intermediate physical pages (IPPs) as executable but non-writable to prevent modification. Additionally, Tide configures them with UXN (User eXecute Never) and all other IPPs with PXN (Privileged eXecute Never) to prevent attackers from injecting instructions at the privileged level.

To address the third challenge, the TCR_EL1 setting instruction must be moved into the IEE for protection; however, the IEE memory cannot be exposed outside. We observed that *the output address size checking only works during page table walk*. If the translation is disabled, the IEE memory will naturally be exposed. Therefore, Tide sets the SCTRL_EL1 register to disable the translation temporarily, enter (sneak into) the IEE to set the TCR_EL1 register to enlarge the output address size, and finally enable the translation. Unfortunately, the SCTRL_EL1 register also suffers from the jump-in-middle attacks, in which the control flow can be hijacked after its setting. To address this, we proposed two methods to ensure that even if the registers are not set correctly, the subsequent execution cannot be affected: 1) *Code executed during translation disabling will not perform any data accesses* to eliminate the impact of cacheability and data endianness changes; 2) *The virtual memory at the same address as the intermediate physical memory set as privileged executable is exclusively owned by Tide, and its memory mapping is protected*. Even if the IPP with the operation of disabling translation is mapped to another virtual page, the execution of subsequent instructions after the translation is disabled will trigger an exception, which prevents attackers from hijacking the control flow and executing the code page, controlled by the attacker, at the IPA.

We implemented and evaluated a prototype of Tide on Linux kernel v6.9.1, which runs on a Cortex A76 2.6 GHz 64-core processor with 64GB of DRAM. It comes with a thin hypervisor implemented as a kernel module and will be landed to run at EL2 during the kernel initialization. Tide provides a set of APIs for users to run an application in the IEE. To evaluate and compare the performance overhead, we implemented all existing IEEs [7, 11, 24, 28] and two VM-based IEEs, which run alongside the hypervisor and in another VM. And we use them to protect two critical kernel sources and five security tools. We not only conduct the experiments on micro-benchmarks, LMBench [1] and UnixBench [2], but also on real-world Phoronix benchmark [40]. Experimental results show that Tide outperforms other IEEs, and incurs low performance overhead.

In sum, the contributions made by this paper are as follows:

- **A quantitative analysis of existing kernel-level IEEs on AArch64.** We systematically summarize the security properties of IEEs, evaluate the overhead of existing IEEs, and show that the design of mediating all page table updates imposes a significant performance overhead.
- **A novel efficient kernel-level IEE on AArch64.** We propose a novel kernel-level IEE that uses the output address size feature for the first time; it proposes multiple techniques to offload most of the software checks required to build an IEE to hardware to achieve better performance.
- **A prototype implementation and evaluation of Tide.** We implement and evaluate Tide, and show that it outperforms existing approaches on LMBench, UnixBench, and Phronoix when protecting critical kernel sources and security tools.

2 Background

2.1 Kernel-level IEEs on AArch64

SKEE [7] utilizes the TTBRn, which stores the page table's base address. It maintains two page tables: the regular page table, which is used for the kernel and does not map the physical memory of the IEE, and the dedicated page table, which is used for the IEE and maps all physical memory. Entering and exiting the IEE requires switching between them. Accessing the IEE from the kernel directly will trigger the hardware *translation fault exception*.

Hilps [11] isolates the IEE by adjusting the range of available virtual addresses controlled by the system register TCR_ELx.TxSZ. It adjusts the range to include or exclude the IEE's virtual address space when entering or exiting the IEE. A *translation fault exception* will be raised when translating an address that exceeds the range, which ensures that the kernel cannot access the IEE.

GENESIS [28] uses the PAN (Privileged Access Never) hardware feature, which prevents the kernel code from accessing the user pages. It sets all virtual memory pages in the IEE as the user pages that forbid access from the kernel, and disables/enables the PAN by setting the PSTATE.PAN field when entering/exiting IEE.

We also found a work that uses the hardware debug feature, i.e., the watchpoint, to create the IEE in the hypervisor [24], which could also be applied in the kernel. The watchpoint in AArch64 can watch a contiguous memory region, and accessing this region will trigger the hardware *debug exception*. Therefore, the virtual memory of IEE can be protected by the watchpoint. Entering and exiting IEE only requires temporarily deactivating and activating the watchpoint by setting the PSTATE.D field.

2.2 Hardware Features on AArch64

2.2.1 Exception Levels. AArch64 supports multiple privilege levels, ranging from EL0 to EL3 [5]. EL3 has the highest privileges, followed by EL2, EL1, and EL0. It runs a secure monitor, hypervisor, kernel, and application. Different system registers are provided for different privilege levels with a suffix of ELx in their names. Converting to a higher level is allowed through an exception, actively or passively. An exception vector must exist at the target exception level to handle the exceptions by type. The base address of the exception vector is stored in the Vector Base Address Register (VBAR_ELx).

2.2.2 Virtual Memory Translation. AArch64 provides two virtual address ranges: the higher range is used to map the kernel, and the lower range is used to map the applications [5]. It also provides two registers, i.e., TTBR0_ELx and TTBR1_ELx, to hold the base addresses of the translation tables for the ranges, respectively. In addition, the Translation Control Register (TCR_ELx) is also provided to control the translation process. During the translation table walk, the MMU will check the input (virtual) address size specified by the TCR_ELx.TxSZ field. Every physical address obtained in this process is called the output (physical) address, which should not exceed the output address size specified by the TCR_ELx.{I}PS field. If it exceeds, an *address size fault* exception will be raised. AArch64 also allows programmers to configure the System Control Register (SCTLR_ELx) to achieve top-level control, such as enabling/disabling the translation and the cacheability of instruction or data accesses.

To accelerate the address translation, a Translation Lookaside Buffer (TLB) is provided to cache the recently used address mappings and distinguishes them from different address spaces by associating them with an ASID (Address Space ID). On AArch64, each of TTBR0_EL1 and TTBR1_EL1 contains an ASID field, and the TCR_ELx.A1 field selects which ASID to become the current ASID.

2.2.3 Stage 2 Translation. When running the guest kernel on a hypervisor, the virtual address translation is more complex due to the memory the kernel uses being virtualized. AArch64 provides two stages of translation to achieve that: the stage 1 translation is used to translate a virtual address (VA) to an intermediate physical address (IPA), and the stage 2 translation is used to translate an IPA to a physical address (PA). The stage 2 translation also provides the memory access controls, such as the UXN (User eXecution Never) and the PXN (Privilege eXecution Never), which control whether the user or the kernel can execute an intermediate memory page.

3 Threat Model and Assumptions

Assumptions. We assume the kernel can be loaded and initialized securely before interacting with potential attackers, which can be in user mode, i.e., an un-trusted user process, and in kernel mode, i.e., malicious loadable kernel modules and BPF programs. We assume the kernel contains vulnerabilities that attackers can exploit to compromise the kernel. We also assume that the IEE code, underlying system software, and hardware are trusted and secure.

Attackers' abilities. Attackers could exploit kernel vulnerabilities to do anything allowed in the kernel. Specifically, attackers may gain arbitrary memory read and write capabilities by exploiting vulnerabilities, and they use them to (1) leak secrets, (2) corrupt critical data, (3) hijack control flow, or (4) inject malicious code. Furthermore, they may also gain such capabilities indirectly by launching DMA attacks [36]. For malicious loadable kernel modules, attackers can directly possess the above capabilities.

Security guarantees. The kernel-level IEE can protect critical kernel resources and run security tools to perform kernel inspection. The only way to enter the IEE from the kernel is through the gates provided. The IEE's design ensures the security of the gate — any attempt to access the IEE that bypasses the gate will cause a crash. However, users of the IEE need to add additional security checks to the interface against confused deputy attacks [26].

4 A Quantitative Analysis of IEEs

Here, we summarize the security properties required by a generic IEE (§4.1), analyze the performance overhead of existing IEEs (§4.2), and summarize the design requirements of a low-overhead IEE (§4.3). For ease of introduction, we rename existing IEEs according to the features of the hardware used: TTBRn-based IEE [7], TxSZ-based IEE [11], PAN-based IEE [28], and Watchpoint-based IEE [24].

4.1 Security Properties in Kernel-Level IEEs

Fig. 1 illustrates the three key security properties that any kernel-level IEE must ensure:

- **P-1: The gate should be secured so it cannot be bypassed**. A gate contains instructions to enable or disable the IEE's access permission. As the only way to enter or exit the IEE, it should be designed carefully to avoid control flow hijacking.
- **P-2: The physical memory of the IEE should be isolated**. The IEE has exclusive physical memory, which can only be translated in the IEE. This is required not only to ensure that the IEE's memory mapping cannot be tampered with to (re-)map the regular memory, but also to ensure that the IEE's physical memory cannot be (alias-)mapped to the kernel.
- **P-3: The kernel $W \oplus X$ should be enforced**. Privileged instructions that may destroy the isolation boundary should be moved from the kernel to the IEE to restrict their execution. To prevent attackers from injecting such instructions at runtime, the kernel $W \oplus X$ must be enforced.

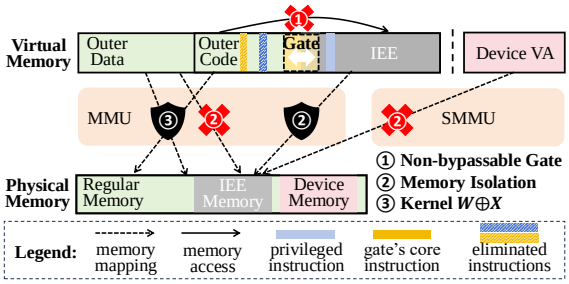

Fig. 1: Security properties of a generic kernel-level IEE.

For **P-1**, all IEEs set the system register to switch accessibility to the IEE, which generally takes two forms: one is through the assignment of immediate values, and the other is through the assignment of a general-purpose register (GPR), whose value is set by the preceding immediate assignment instructions. The PAN-based IEE and Watchpoint-based IEE use the first form (shown in Table 1). However, these two IEEs should have additional prevention so that users may not access them; for example, users could directly access the unprivileged pages in the PAN-based IEE. As such, the PAN-based IEE requires the kernel page table isolation (KPTI) [17], while the Watchpoint-based IEE requires swapping debug registers during the user-kernel switching.

For the second form, the attacker may bypass the GPR's setting instructions and directly jump to the system register setting's instruction to set the register to an arbitrary value (as known as the *jump-in-middle* attack). Since the system registers contain a variety of system controls, incorrect settings will affect the functionality of subsequent gate codes. As such, the TTBRn-based IEE uses the XZR register to eliminate the GPR's setting instructions, which requires the virtualization technique to place the page table root at the zero address; The TxSZ-based IEE adds extra checks for the value immediately after setting the TCR_ELx.

For **P-2**, all IEEs apply for physical pages from the kernel as the physical memory of the IEE and ensure that it can only be translated by the IEE's page table. To this end, all page tables (used by the MMU and the SMMU) are isolated in the IEE, and all updates will be mediated and validated. It is also the same for **P-3**. Since the page tables only require the integrity protection, the IEE also maps them to the kernel space with the read-only permission to avoid unnecessary IEE invocations introduced by the read access.

4.2 Performance Overhead Analysis

We use UnixBench [2] to measure the performance overhead of IEEs and conduct the composition analysis. We implemented existing IEEs in Linux-v6.9, which runs on a Cortex-A76 2.6 GHz 64-core processor with 64GB of DRAM.

Gate Types	Core Privileged Instruction	CPU Cycles
PAN-based	`msr PAN, #0x0[#0x1]`	137
TxSZ-based	`msr TCR_ELx, Xt`	154
TTBRn-based	`msr TTBR1_ELx, Xt`	161
Watchpoint-based	`msr DAIFset[DAIFclear], #0x8`	125
OPT Hypercall	`hvc #0; eret`	685

Table 1: Round trip NULL invoke latency for existing IEEs.

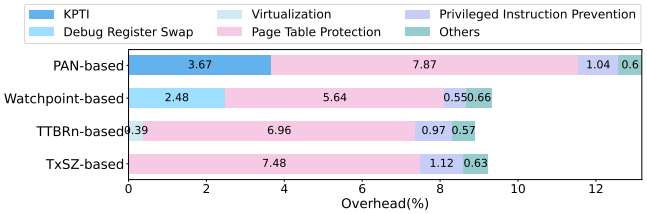

Fig. 2: Overhead of the empty IEEs on UnixBench.

4.2.1 Overhead of Gate-Switching. Since the latency of the gate switching is the key to the performance overhead [29], we measured the round-trip latency of a NULL invocation. As shown in Table 1, the PAN-based and the Watchpoint-based gates have the lowest latencies. This is because the setting of the `PSTATE` register does not require an instruction synchronization barrier, i.e., the `ISB`, while the setting of the `TTBRn_ELx` and `TCR_ELx` needs it. We also compared the gates' latencies of VM-based IEEs — running the IEE alongside a hypervisor or into another VM. Since they both need the hypercall to conduct the gate, we measure the round-trip latency of a NULL hypercall. Note that a complete hypercall usually contains the system registers switching, which is not necessary for IEEs [61]. Therefore, we optimized the hypercall by removing such an operation and reduced the latency from 1,246 to 685. Even so, the latency is still exponentially higher than that of existing gates.

4.2.2 Overhead of the Empty IEEs. To evaluate the baseline performance of existing IEEs, we assessed the performance overhead of the empty IEE, which performs no meaningful protection, except for the basic protection of page tables and privileged instructions used to build the IEE. The results are shown in Fig. 2: the overheads of PAN-based IEE, Watchpoint-based IEE, TTBRn-based IEE, and TxSZ-based IEE are 13.18%, 9.33%, 8.89%, and 9.23%, respectively. This means that an empty IEE incurs significant overhead.

The composition analysis shows that implicit invocations of the IEE introduce the primary overhead. This includes two invoking situations: one is introduced by mediating the updates of page tables; the other by mediating the execution of privileged instructions. For the first type of invoking, although we adopted the optimization of group page table updates proposed in SKEE [7] to reduce the number of invocations, the overhead they introduce is still high (ranges from 5.64% to 7.87%); For the second type of invoking, it introduces only about 1% performance overhead. Therefore, the mediation of all page table updates in IEEs is costly.

The second performance overhead mainly comes from the additional design introduced by the secure gate. They include the KPTI, the debug registers swapping, and the virtualization mentioned in §4.1, which bring 3.67%, 2.48%, and 0.39% overhead, respectively. The overhead introduced by virtualization is negligible, since the TTBRn-based IEE avoids expensive hypervisor traps. The other two incur relatively high overhead because they work during user-kernel context switches, and context switches' efficiency significantly impacts performance [27].

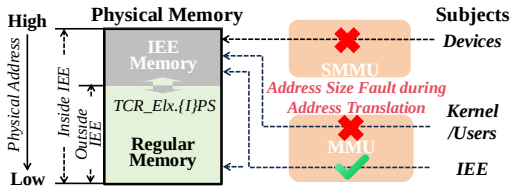

Fig. 3: The core idea of Tide.

4.3 Lessons Learned from Existing IEEs

Based on the above analysis, we summarize the design suggestions in three aspects:

A-1: Invoking the IEE should not be trapped into a higher privilege level. Compared with isolation across privilege levels, the kernel-level IEE has advantages in performance and security. *Trapping is expensive and should be avoided when invoking the IEE, but enhancing isolation with virtualization without trapping or with minimal trapping is feasible.* Isolating the sensitive part into a higher privilege level is not recommended since it will significantly increase the more privileged system's TCB.

A-2: The IEE should use the same hardware features to prevent access from both the kernel and the user. Suppose the hardware feature used in the gate provides only kernel isolation. In that case, additional mechanisms are required to isolate user access to the IEE, often resulting in a significant performance overhead due to the user-kernel switching being performance sensitive.

A-3: Page table updates should avoid being mediated by the IEE. To ensure memory isolation, existing IEEs need to take on the additional role of a *reference monitor* that mediates and validates the updates of all page tables, leading to frequent invocations of the IEE, which results in a significant overhead. Such invocations should be avoided to achieve lower performance overhead.

5 Overview

Following §4.3, we outline the design of Tide in this section.

5.1 Core Idea

The core idea behind Tide is to build an efficient kernel-level IEE that could *avoid mediating page table updates as a reference monitor, and offload the security checks on page tables to the hardware directly.* The key hardware feature used in Tide is the output address size checking during the page table walk. The output (physical) address obtained in this process cannot exceed the output address size specified by the `TCR_ELx.{I}PS`. Therefore, we could place the IEE's physical memory outside the kernel's allowed output address size. And the hardware ensures that all page tables in the kernel cannot translate the IEE memory.

In detail, as shown in Fig. 3, Tide divides physical memory into two regions, according to the high and low addresses. The kernel and IEE are configured to use lower and upper regions, i.e., the regular and IEE memory. The page tables used in the kernel cannot

translate the IEE memory because its address exceeds the kernel's allowed output address size. Legal access to the IEE should be done through a gate that enables page table translation to the IEE memory by enlarging the output address size. The IEE's page table is placed in the IEE memory to prevent the kernel from corrupting it. The SMMU (System Memory Management Unit) also supports this feature. Tide uses a similar method to configure the output address size to prevent DMA from accessing the IEE memory.

5.2 Challenges

It still faces challenges in terms of security and flexibility:

C-1: Configuring the output address size with more flexibility. The granularity of setting the output address size is very coarse, with only seven sizes available: 4GB, 64GB, 1TB, 4TB, 16TB, 256TB, and 4PB. It is challenging to set a reasonable output address size for different physical memory sizes for the kernel to avoid memory waste. For example, given 32GB of physical memory, the kernel-allowed output address size can only be configured to 4GB to reserve 28GB for the IEE. However, this dramatically wastes memory.

C-2: Enforcing kernel $W \oplus X$ still needs mediating page table updates. The output address size feature only checks the memory mappings in page tables, not the access permissions required by the kernel $W \oplus X$. This still requires the IEE to isolate the page tables, mediate, and validate their updates, which cannot reduce the performance overhead raised by such IEE invocations.

C-3: Securely exposing the IEE memory in the gate. Attackers may launch the *jump-in-middle* attack — jumping directly to the `TCR_EL1` setting instruction and setting it to be an illegal value. Unfortunately, any check after this instruction may be bypassed. Since the `TCR_EL1` register contains all the address translation controls, changing page granularity may cause unpredictable behavior for subsequent execution [28]. We found that by mapping the gate code to a specific virtual address and then executing a `TCR_EL1` setting instruction to enlarge the page granularity, it is possible to fetch an attacker-controlled address immediately instead of the next instruction in the program sequence. This allows an attacker to expose the IEE memory but still execute kernel code without entering the IEE.

5.3 Tide Overview

To address the first two challenges, our key insight is that *the stage-2 translation could provide the opportunity for independent memory mappings and permissions checking in addition to the output address size checking in the stage-1 translation.* We could impose the memory control independently of the stage-1 translation. Specifically, Tide activates the stage-2 translation to introduce the intermediate physical address (IPA) between the VA and PA.

To address the **C-1**, Tide uses the more flexible IPA mapping — *the IEE memory and regular memory can be remapped into a larger IPA space and do not have to be contiguous.* Specifically, Tide maps the IEE memory to an arbitrary IPA exceeding the output address size that the kernel can use in the stage-1 translation. In addition to memory mapping, the stage-2 translation can also set the access permissions of an intermediate physical page (IPP). It could provide *the write protection and finer-grained execution control for an IPP*, which can achieve the kernel $W \oplus X$ in stage-2 translation. Specifically, the code pages allowed to be executed in kernel mode are set to be write-protected, and other pages are set to PXN (Privileged eXecute Never) to prevent them from being executed in kernel mode, thus addressing the **C-2**. Note that the stage-2 translation table configuration only needs to be done once, and no additional hypervisor traps are introduced at runtime.

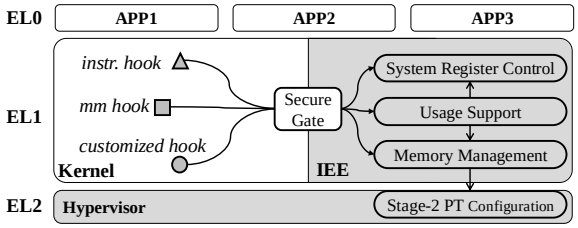

Fig. 4: The architecture overview of Tide.

To address the **C-3**, the `TCR_EL1` setting instruction must be moved into the IEE for protection; however, the IEE memory cannot be exposed outside. We observed that *the output address size checking only works during page table walk.* If the translation is disabled, the IEE memory will naturally be exposed. Therefore, we can set the `SCTRL_EL1` register to disable the translation temporarily, enter (sneak into) the IEE to set the `TCR_EL1` register to enlarge the output address size, and finally enable the translation. However, it still suffers from the jump-in-middle attacks — attackers could change the endianness of data access to affect the correctness of subsequent gate code execution and hijack the control flow after disabling the translation by mapping and executing the setting instruction at other virtual addresses. To this end, Tide has made a more sophisticated design of the gate's assembly code and memory mapping to ensure that the code, which is executed during disabled translation, does not access data and can only be executed at the fixed virtual address that attackers cannot control.

6 System Design

Architecture overview. Fig. 4 gives the architecture overview of Tide. It consists of two parts: one is the thin hypervisor (called *minivisor* in this paper), which is implemented as the kernel module and will be landed to run at EL2 during the kernel initialization, and the other one is the kernel-level IEE at EL1, which will be initialized immediately after the kernel finishes initializing. To avoid hypervisor traps, the minivisor only performs memory virtualization, and all interrupts are directly delivered to EL1 for processing.

IEE mainly contains four components: (1) The *memory management* component manages the stage-1 page table used by the IEE and the stage-2 page table (detailed in §6.1). A *mm agent* is designed in the kernel to apply physical memory for the IEE. The ownership of the applied physical memory will be transferred to the IEE by configuring the stage-2 page table; (2) The *secure gate* is used to enter and exit the IEE, whose assembly code and memory mappings are carefully designed to prevent being bypassed (detailed in §6.2); (3) The *system register control* component is not only responsible for restricting the execution of the privileged instructions that may break the isolation boundary but also securely supports the dynamic loaded/generated kernel code (detailed in §6.3); (4) The *usage*

support component provides a set of APIs for users to program and run an application in the IEE (detailed in §6.4).

6.1 Memory Mangement

In Tide, secure and efficient memory management needs to fulfill the following four requirements:

- **R-1: The IEE's virtual space should be embedded into the existing space**. The IEE should have an independent virtual address space; assigning a page table for the IEE could achieve this. However, it will introduce additional performance overhead of page table switching at the gate. Thus, reserving a virtual address range in existing space as the IEE space is required.
- **R-2: Supporting the integrity-only data protection**. In some cases, sensitive data protected by the IEE may require integrity protection only; the read restriction brings extra performance overhead. The dynamically generated privileged code is a particular case; it requires writable to emit code in the IEE but non-writable to ensure the kernel $W \oplus X$. Thus, the separation of the non-write and write permissions should be supported.
- **R-3: The control flow cannot be hijacked when disabling the translation**. The gate needs to disable the translation temporarily to sneak into the IEE. Attackers may abuse this operation to hijack the control flow after the translation is disabled — they could map the gate to the same virtual address as the physical address of the attacker-controlled code page. When disabling the translation, the processor executes the attacker-controlled code at the physical address instead of the next instruction in the gate. Worse, the IEE memory is now exposed and can be accessed arbitrarily. Therefore, the gate code pages must be executable at the fixed virtual address.
- **R-4: Keeping memory management simple.** To minimize the TCB of Tide, the memory management in the IEE and minivisor should be as simple as possible.

Overall design. The overall memory mapping of Tide is shown in Fig. 5. *Firstly*, the gate and IEE virtual space are embedded in all user spaces instead of the kernel space (fulfill the **R-1**); this is because the VA of disabling translation cannot be higher than the maximum IPA size, which is much smaller than the VA of the kernel space. *Secondly*, a *trust space* is embedded in user spaces, whose memory mappings are protected, providing the kernel and IEE with trusted access to physical memory other than the IEE memory. A *WP memory* provides a non-writable memory, which is not only used to protect the memory mapping of the trust space, but also used to map part of the IEE memory (fulfill the **R-2**). *Thirdly*, the gate is placed at the junction of the IEE and trust spaces. To avoid it being mapped and executed at the attacker-controllable VA, i.e., the remaining user and kernel spaces, Tide configures the PXN of the stage-2 page table to make the page at the same IPA corresponding to the attacker-controllable VA not have executable permissions at the privileged level (fulfill the **R-3**). *Lastly*, to fulfill the **R-4**, the address ranges of the IEE and trust spaces are aligned with the first-level stage-1 page table entries, which allows us to embed both spaces into the user space by only write-protecting the page table root. Their stage-1 memory mappings are linear; the regular memory maps the entire physical memory through linear mapping,

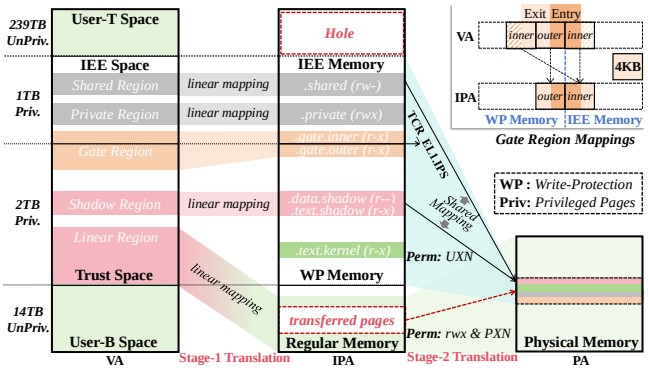

Fig. 5: The stage-1 and stage-2 page table mappings in Tide.

and memory pages used in the IEE and WP memory are transferred from the regular memory by configuring the stage-2 page table.

6.1.1 Stage-1 Page Table Management. Tide embeds 1TB of the IEE space and 2TB of the trust space into user space and sets them as privileged pages. They exclusively occupy two and four first-level stage-1 page table entries, respectively. The user space is split into two regions: the 14TB at the bottom (User-B) and the 239TB at the top (User-T). The IEE space contains two regions: one is the *private region*, which is used to hold the private code and data in the IEE, and the other is the *shared region*, which is used to hold the integrity-only protected kernel data or code. The trust space also contains two regions: one is the *shadow region*, which is shared with the shared region of the IEE space without writable permission, and the other one is the *linear region*, which maps the regular memory. The gate region crosses the trust and IEE spaces.

Memory mappings management. The IEE space is mapped to the IEE memory of the same size in a linear mapping manner, and their starting addresses are the same. The settings for the trusted space and WP memory are the same. The starting IPA of the IEE memory is just over 16TB, which is configured as the output address size allowed for the kernel. The gate region contains three consecutive pages, two in the trust space and the other in the IEE space. The last two pages are identity mapped to two IPPs at the boundary between the WP and IEE memory, which are called the *outer IPP* and the *inner IPP*, respectively; The first page is also mapped to the inner IPP. The gate region has an exit gate and an entry gate, which span the first and second pages and the second and third pages, respectively. Thus, both the entry and exit gates have an outer and inner portion relative to the IEE. The shadow and linear regions in the trust space are mapped linearly into the shadow region of the WP memory and the regular memory, respectively.

Memory mappings protection. To protect the stage-1 page tables, which translate the IEE and trust spaces, Tide uses memory isolation at the IPA. *Firstly*, Tide moves the setting instruction of the `TTBR0_EL1` register into the IEE to monitor the root of all user page tables; *Secondly*, Tide write-protects the user page table root by moving it into the WP memory. Tide mediates all updates of user page table roots to ensure the trust and IEE spaces are exclusive. It should be noted that although the above updates need to enter the IEE, the overhead is negligible; *Thirdly*, the remaining levels of page tables to translate the IEE and trust spaces are isolated by

placing them in the IEE and WP memory, respectively. Since they are linear mappings, they are rarely updated after initialization.

6.1.2 Stage-2 Page Table Management. The kernel still conducts physical memory management instead of the minivisor. To this end, the regular memory maps the entire physical memory through linear mapping. Tide installs a *mm agent* in the kernel to apply for (immediate) physical pages from the buddy system as the IEE and WP memory. Once completed, the kernel no longer owns these pages. It unmaps them from the regular memory by invoking the minivisor to configure the stage-2 page table. Tide transfers 1/32 of the pages from the regular memory in advance to avoid frequent minivisor invoking. The minivisor will perform security checks to ensure the agent cannot be abused (detailed in Appendix A).

Tide protects the memory mappings of the IEE and trust virtual memory spaces, so the intermediate physical memory with the same address as other virtual memory is not executable at the privileged level to ensure that the code page that disables translation in the gate cannot be mapped into other virtual memory. Since the IEE and trust spaces have the same address as the IEE and WP memory, Tide only needs to ensure that the regular memory is not executable at the privileged level. To this end, the regular memory's permissions on the stage-2 page table are set to *rwx* and *PXN*. The kernel text is moved into the WP memory to ensure the kernel code can be executed. The IEE and WP memory permissions are set to *UXN*; the WP memory does not have writable permission. The shadow region in the WP memory is split into two segments: the *.data.shadow* with the read-only permission, which is used to hold the integrity-only protected data, and the *.text.shadow* with the readable and executable permissions, which is used to keep the dynamical-generated kernel code. This region is dual-mapped into the shared region of the IEE memory with the *rw* permission.

Protection against DMA attacks. Since attackers may abuse the DMA functionality to bypass isolation [36], it is necessary to restrict DMAs. If a device supports SMMU, the access will be translated based on SMMU page tables, which also support two translation stages. Hence, Tide arranges the same stage-2 page table for the SMMU and configures the same output address size in the stage-1 page table as the kernel (detailed in Appendix B). For DMA without using SMMU, Tide inspects the update of the memory-mapped registers of the DMA controller via isolating their settings into the IEE to prevent access to the IEE and WP memory.

6.2 Secure Gate

The gate's design must ensure the following two security properties:

- *Atomicity*: The execution of the gate can only start from the entrance. Jumping to the middle of this sequence is not allowed;
- *Determinacy*: The execution result is deterministic and is not affected by the current state of all registers and kernel memory.

6.2.1 Enforcing Atomicity and Determinacy. The entry and exit gates have inner and outer parts; the outer parts are not isolated from the kernel. To ensure the first property, Tide *places as few instructions as possible in the outer part*: the outer part of the entry gate only disables interrupts and translation, and the outer part of the exit gate only enables interrupts. The inner part of the entry gate will *reset the system registers used in the outer part* to prevent attackers from arbitrarily setting system registers by launching jump-in-middle attacks. However, to avoid incorrect settings on the outside, causing the reset of the system registers in the inner part to be interrupted or set to incorrect values, Tide performs additional designs on the interrupt entry and the assembly code in the inner part. To ensure the second property, the assembly code in the gate is carefully designed *not to access the outside kernel memory, and registers must be defined before being used.* Moreover, Tide provides a thread-safe design, such as each core has an IEE stack.

```
.macro iee_switch_stack, sym, base_reg, id_reg, reg1, reg2
    ldr \base_reg, \sym                 ## 1) Load the base address of
    mrs \id_reg, tpidr_el1              ##  ; IEE stack pointer array
    ldr \reg1, [\base_reg, \id_reg]     ## 2) Get the cpu offset
    mov \reg2, sp                       ## 3) Load the new stack
    mov sp, \reg1                       ##  ; pointer and store the old
    str \reg2, [\base_reg, \id_reg]     ##  ; stack pointer
.endm
iee_exit_gate:                          ## IEE inner exit gate
    iee_switch_stack iee_stack_arr, x9, x10, x12, x14 ## 1) SP switch
    ldr x12, kern_tcr_arr               ## 2) Load percpu TCR_EL1 value
    ldr x12, [x12, x10]
    msr tcr_el1, x12                    ## 3) Set TCR_EL1 register
-------------------------PAGE_BOUNDARY-------------------------
    .align PAGE_ALIGN                   ## IEE outer exit gate
    isb                                 ## 4) Instruction barrier
    msr daif, x13                       ## 5) Restore DAIF
    ret                                 ## 6) Return from IEE to kernel
    ...
## @param cmd          cmd index
## @param [arg0-arg5]  parameters for IEE interfaces
iee_entry_gate:                         ## IEE outer entry gate
    mrs x13, daif                       ## 1) Record DAIF status
    msr daifset, 0x3                    ## 2) Disable interrupts
    mrs x10, sctlr_el1                  ## 3) Read SCTLR_EL1
    bic x10, x10, 1                     ## 4) Clear SCTLR_EL1.M
    msr sctlr_el1, x10                  ## 5) Disable MMU
    isb                                 ## 6) Instruction barrier
-------------------------PAGE_BOUNDARY-------------------------
    .align PAGE_ALIGN                   ## IEE inner entry gate
    msr daifset, 0x3                    ## 7) Disable interrupts
    mov x12, 0x3510                     ## 8) Prepare TCR_EL1 value
    movk x12, 0xb550, lsl 16            ##  ; Set TCR_EL1.A1, etc
    movk x12, 0xb5, lsl 32              ##  ; Set TCR_EL1.IPS, etc
    movk x12, 0x0, lsl 48
    msr tcr_el1, x12                    ## 9) Set TCR_EL1 register
    orr x10, x10, 0x1                   ## 10) Enable MMU
    orr x10, x10, 0x4                   ##   ; Enable data cacheability
    orr x10, x10, 0x1000                ##   ; Enable inst cacheability
    and x10, x10, 0xfffffffffdffffff    ##   ; Force little endian
    msr sctlr_el1, x10                  ## 11) Reset SCTLR_EL1 register
    isb                                 ## 12) Instruction barrier
    iee_switch_stack iee_stack_arr, x9, x10, x12, x14 ## 13) SP switch
    stp x13, x30, [sp, -16]!            ## 14) Store DAIF status and LR
    ldr x14, iee_dispatcher_ptr
    blr x14                             ## 15) call IEE dispatcher
    ldp x13, x30, [sp], 16              ## 16) Load DAIF status and LR
    ldr x14, iee_exit_gate_ptr          ## 17) Branch to IEE inner exit
    br x14                              ##   ; gate
iee_dispatcher_ptr: .quad ... ## The address of IEE dispatcher
iee_exit_gate_ptr:  .quad ... ## The address of IEE inner exit gate
iee_stack_arr:     .quad ... ## The address of IEE stack pointer array
kern_tcr_arr:      .quad ... ## The address of kernel tcr value array
```

Listing 1: The Entry and exit gates. Assembly code marked in grey is executed with address translation disabled.

6.2.2 The Workflow of Gates. Listing 1 gives the whole assembly code in the entry and exit gates. The current IEE design does not support interrupts to prevent executing in the IEE from unsafely switching back to the kernel. They work as follows:

- *Step-1:* Save the DAIF status and disable interrupts (lines 23-24).
- *Step-2:* Set the SCTRL_EL1.M field to disable the stage-1 page table translation to expose the IEE memory (lines 25-28).
- *Step-3:* Enter into the IEE and disable interrupts again (line 31).
- *Step-4:* Reset the TCR_EL1 to enlarge the output address size to expose the IEE memory and switch the ASID (lines 32-36).
- *Step-5:* Set the SCTRL_EL1.M|C|I|EE fields to enable the translation and force enable cache and little-endian (lines 37-42).
- *Step-6:* Switch to the IEE stack of the current core (line 43).
- *Step-7:* Call the dispatcher of internal functions (lines 44-47).
- *Step-8:* Branch to the IEE inner exit gate (lines 48-49).
- *Step-9:* Switch back to the kernel stack (line 10).
- *Step-10:* Reset the TCR_EL1 for the kernel to shrink the output address size to disable access to the IEE Memory (lines 11-16).
- *Step-11:* Restore DAIF and return (lines 17-18).

6.2.3 Security Designs in the Gate. Here we list all security designs.

The necessary to disable interrupts outside. An attacker may bypass the interrupt-disabling operation by jumping directly to line 25, which can hijack the control flow of line 30 when an interrupt happens to arrive. He/she can access the IEE memory in the interrupt handler. To defend against this attack, TIDE makes the exception vector table pointed to by the VBAR_EL1 valid when translation is enabled, and invalid when translation is disabled. Specifically, TIDE makes the VBAR_EL1 always point to the kernel address space, and its address will exceed the maximum intermediate physical address size. And TIDE disallows its updates.

Secure execution during the translation disabling. An attacker may jump directly to line 27 to set the other fields of the SCTLR_EL1, thus affecting all assembly code (lines 30-42) before resetting the register. To this end, we comprehensively analyzed each field of the SCTRL_EL1. Among them, only the EE field affects the security; it determines the endianness of data access, and its change will affect the data access. Therefore, this assembly code is designed not to access memory.

Secure execution after the translation enabling. In addition to the SCTLR_EL1.EE field, the I and C fields determine the cacheability of instruction fetches and data accesses. They may affect the security and performance of code execution after enabling the translation. Therefore, TIDE forces enable cache and little-endian when enabling the translation (lines 37-40). The IEE does not reset all fields but only forces specific fields to be set for performance reasons. This allows the IEE to share the same value with the kernel, thus avoiding resetting the register when exiting the IEE. Note that the force setting of these fields does not affect the kernel's normal function, since they are generally unchanged.

TLB isolation between the kernel and the IEE. Since the output address size checking is performed during page table walk, the mappings loaded into the TLB during the IEE's execution can be used by the kernel. Hence, isolating their TLB entries is needed. To this end, TIDE configures the kernel and the IEE to use different ASIDs, and switches the current ASID when entering and exiting the IEE. In detail, the IEE's unique ASID value (0) is stored in the TTBR1.ASID and the TTBR0.ASID stores the kernel ASID values.

SysReg	Fields Control	Security Policy
TTBR0_EL1	BADDR is in Protected PGD List ASID != 0	The pgtable root cannot be spoofed.
TTBR1_EL1	ASID == 0	The TLB entries used by the kernel and the IEE must be isolated.
TCR_EL1	AS == 0b1 && A1 == 0b0 IPS == 0b100 && TG0 == 0b00 DS == 0b0 && T0SZ == 0b010000	The parsing of the page table structure cannot be changed.
SCTLR_EL1	EE == 0b0	
	M == 0b1	Translation cannot be disabled.
	I == 0b1 && C == 0b1	Cache cannot be disabled.
TPIDR_EL1	Prohibit any change	The core number must be trusted.
VBAR_EL1	bit[55] == 0b1	Interrupts cannot be handled when translation is disabled.

Table 2: The fields control in system registers.

Switching the current ASID can be done together with adjusting the output address size by flipping the TCR_EL1.A1 bit (lines 33-34).

Concurrent execution securely. To ensure that it is safe for multiple cores to invoke the IEE concurrently, the context for switching in and out of the IEE is private to each core. The context is designed as an array structure indexed by the core number. The core number is recorded in the TPDIR_EL1 register, which is initialized by the kernel, and then TIDE does not allow the kernel to modify it. The context stores the stack pointer of the per-core IEE stack and the kernel TCR_EL1 value in the current core.

6.3 System Register Control

The settings of the memory-related system registers (i.e., TTBR0_EL1, TTBR1_EL1, TCR_EL1, and SCTLR_EL1) and the system registers used in the gate (i.e., TPIDR_EL1 and VBAR_EL1) should be controlled — their settings are moved into the IEE, and updates to specific fields will be restricted to avoid being set to arbitrary values. The detailed control on different fields is listed in Table 2. It involves the security of page table roots, the parsing of page table structures, the TLB isolation, the memory consistency, and the gate's execution flow. Undoubtedly, the system register control cannot affect the normal kernel functionality. Our research found that except for the SCTLR_EL1.M, normal kernel execution does not modify the above fields. Therefore, control of other fields does not affect normal functionality. A normal kernel may set the SCTLR_EL1.M to disable the translation. Fortunately, it is only a temporary operation, and the code executed during this period is very short (only a few dozen assembly instructions). Thus, TIDE only needs to move and execute these codes in the IEE.

Supporting the dynamically generated code. To support the dynamically generated/loaded kernel code, such as eBPF JITed code and loadable kernel modules, a shadow code region in TIDE is provided (mentioned in §6.1). The kernel should invoke the IEE to emit the code into this region. All encodes will be inspected, and the setting instructions of the above registers will be replaced by BL instructions to invoke the IEE to complete the updates. Since instructions on AArch64 are fixed length and aligned, TIDE can obtain all instructions accurately from the emitted encoding.

6.4 Usage Support

As shown in Table 3, TIDE provides a set of APIs for users/programmers. The internal APIs are used for applications run in the IEE; the external APIs are used for the kernel invocations.

APIs (*Names with _* are internal APIs*)	**Descriptions**
int tide_load_ko (void *elf_addr, int len, char *name) int tide_find_func (int ko_descriptor, char *func_name) long tide_call_func (int func_id, ulong arg0, ulong arg1, \\ ulong arg2, ulong arg3, ulong arg4, ulong arg5) void *tide_emit_code (void *dst, void *src, int len)	Load a module, find the function, and call it from the kernel. JIT or Load code.
void *_alloc (int len, int align, bool shared, bool code) void *_free (void *mem_ptr)	Allocate and free memory in IEE.
void *_copy_from_outside (void *dst, void *src, int len) void *_copy_to_outside (void *dst, void *src, int len)	IEE copies mem. with the kernel.

Table 3: Tide APIs.

External APIs. All codes are loaded into the private region of the IEE space as kernel modules, which we call IEE applications. A user can invoke the `tide_load_ko()` to load a kernel module and obtain its descriptor. The IEE maintains a function pointer array internally to store the functions exported by the kernel module. Users can use the function name and the module's descriptor to which it belongs to obtain the index number of the function pointer array by invoking the `tide_find_func()` and then use this number to invoke the target function via the `tide_call_func()`. The IEE uses this array to achieve the dispatch of applications' function calls. It supports dynamically generated kernel code. Users could invoke the `tide_emit_code()` to emit code in the shadow region of the trust space, and the IEE will perform inspection.

Internal APIs. The IEE has a built-in memory allocator to manage the shared and private memory regions in the IEE memory. Users can dynamically allocate memory from the *shared* region or *private* region of the IEE space via specifying the *shared* parameter of the `_alloc()`. They can also specify the address alignment requirements of the memory and whether it is code when the *shared* parameter is specified. The memory can be released using the `_free()`. Tide provides the memory copy functions between the IEE and the kernel, similar to the `copy_to/from_user()` in the kernel. Since page tables are untrusted, to safely access kernel memory, the kernel addresses must be converted into the linear mapping region of the trust space before accessing.

7 Implementation

Support for virtualization. In Tide's design, it uses the Type-1.5 virtualization method, which runs as a *minivisor* in EL2 when the kernel boots up. The implementation of the *minivisor* is simple – with only 585 lines of C code and 115 lines of assembly code. Nevertheless, Tide can be implemented on any type of virtualization, such as the KVM and bare-metal hypervisors. We only need to add a small amount of code to configure the stage-2 translation table. To demonstrate that, we implemented the core logic of the *minivisor* into the KVM and found that it only required modifying (deleting and adding) 98 lines of C code in the KVM.

Avoiding user memory allocation in the IEE and trust spaces. The memory areas management in the user space is conducted through a virtual memory area structure (i.e., `vm_area_struct`). Tide has created the `vma_area_struct` structures corresponding to the trust space and IEE space for each process in advance. In addition, Tide also adjusts the starting positions of the stack, the heap, and the mmap areas to avoid these two spaces.

Support for dynamically loaded kernel modules. Due to the kernel $W \oplus X$, the dynamically loaded kernel modules cannot be granted executable permissions. To address this, we modified the loader module of the kernel to invoke the `tide_emit_code()` API to securely load the code segment of kernel modules into the shadow region of the trust space. Then, we mapped these code pages at the loaded virtual address of the module in the kernel space.

We have implemented two typical scenarios: one is the protection of critical kernel structures, and the other is running security tools:

Protecting the credentials. The `cred` records the permissions of a file or a process. Attackers usually corrupt it to achieve privilege escalation [31]; isolating it is required for kernel security [9, 26, 35]. To this end, we created a kernel module to manage these structures and loaded it into the IEE. The kernel module uses the `_alloc()` and `_free()` to allocate and free memory in the shared region of the IEE space to ensure their integrity. Their pointers will be relocated to the shadow memory region in the trust space before being returned to the kernel. To avoid corruption of the pointers, we adopt the ownership method proposed in DOPE [35]. All updates of `cred` will be mediated to invoke the IEE to complete.

Protecting the page tables. The page tables are a sensitive system structure; isolating them could thwart various attacks [14]. Like protecting credentials above, we created a kernel module in the IEE to manage page tables. In addition, we also performed some security checks on the updates of page tables to defend against potential confused deputy attacks. In addition to the security check of the page table root pointers, we also validated all page table entries to achieve a stronger kernel $W \oplus X$, which additionally enforces the mapping of kernel code (against the page-oriented programming [21]) and the integrity of the read-only data.

Running security tools. Tide supports running security tools to inspect the kernel to infer whether the kernel is being attacked. Security tools generally have two execution modes: callback-based and periodic. For the first one, Tide installs hooks in the kernel code to invoke the IEE; for the second one, Tide intercepts all interrupts and invokes the IEE to run the security tool before the interrupt returns. We run a callback-based tool, namely LKRG [3], and four periodic tools, which include three rootkit detection tools ModTracer [37], Rootkit Spotter [33], and Rootkit Breaker [32], and a retrofitted Virtual Machine Introspection (VMI) tool nitro [45]. Since the nitro inspects physical memory, it can access the trust space directly. For the remaining four tools, most of the critical structures they check are located in the linear mapping region; they can convert the address of the structure into an address in the trust space, which is also linearly mapped, for access. To access other regions, these tools must obtain the physical address using the `AT` instruction and then access it from the trusted space.

8 Evaluation

In this section, we focus on the performance evaluation of Tide. The prototype is implemented on Linux Kernel v6.9.1, which runs on a Cortex-A76 2.6 GHz 64-core processor with 64GB of DRAM.

Microbenchmarks. We evaluate and analyze the CPU cycles of the gates, and run UnixBench [2] and LMBench [1] to evaluate the impact on the basic kernel operations. UnixBench is designed to

	Benchmarks	TTBRn-based			TxSZ-based			PAN-based			Watchpoint-based			Alongside Hypervisor			Isolate in a VM			TIDE		
		empty	+cred	+pgtable	empty	+cred	+pgtable	empty	+cred	+pgtable	empty	+cred	+pgtable	empty	+cred	+pgtable	empty	+cred	+pgtable	empty	+cred	+pgtable
UnixBench	Dhrystone 2	-0.03%	-0.07%	-0.17%	0.26%	0.07%	0.34%	0.11%	0.18%	0.11%	0.06%	-0.09%	-0.18%	-0.05%	0.01%	-0.10%	-0.04%	-0.15%	-0.04%	-0.10%	0.10%	0.15%
	DP Whetstone	0.02%	-0.01%	0.02%	0.01%	0.00%	-0.03%	0.01%	0.01%	-0.01%	0.01%	0.00%	-0.01%	0.01%	0.00%	0.01%	-0.01%	0.00%	-0.01%	-0.01%	0.01%	0.01%
	Execl T/P	17.53%	20.52%	22.90%	17.88%	21.52%	21.87%	18.40%	20.63%	21.80%	16.46%	18.24%	19.99%	0.38%	4.82%	37.87%	0.82%	6.76%	53.86%	1.18%	4.13%	18.38%
	FC 1024/2000	5.92%	7.22%	9.88%	8.46%	14.48%	15.05%	14.13%	17.17%	17.36%	6.70%	13.34%	13.03%	3.82%	6.70%	6.32%	3.65%	7.38%	9.20%	1.49%	3.51%	5.65%
	FC 256/500	7.21%	9.01%	11.73%	8.24%	12.85%	13.54%	14.74%	18.70%	18.81%	8.72%	12.27%	16.70%	3.22%	4.55%	6.65%	3.84%	6.56%	7.38%	2.01%	3.97%	6.12%
	FC 4096/8000	8.40%	9.94%	10.13%	5.11%	8.75%	14.96%	9.47%	13.64%	14.26%	6.24%	10.04%	12.02%	5.80%	12.48%	15.45%	6.05%	17.14%	17.50%	3.34%	6.93%	7.93%
	Pipe T/P	2.55%	3.25%	3.13%	3.60%	3.97%	4.42%	14.11%	14.47%	18.67%	6.98%	7.39%	7.19%	1.53%	2.56%	2.78%	2.10%	3.17%	5.83%	0.18%	1.39%	1.98%
	Pipe Switch	0.49%	3.21%	0.62%	1.63%	3.67%	2.31%	4.55%	5.36%	5.73%	1.23%	5.39%	2.58%	0.31%	3.27%	0.33%	0.37%	2.77%	0.80%	0.38%	2.13%	1.52%
	Proc Creation	21.34%	22.01%	23.04%	20.24%	21.29%	21.43%	19.22%	20.42%	20.23%	20.21%	23.11%	23.80%	4.12%	10.99%	34.69%	4.90%	15.55%	48.11%	5.32%	6.35%	17.39%
	Shell Scripts(1)	8.78%	13.86%	16.01%	9.38%	9.44%	9.78%	5.30%	6.31%	11.55%	6.81%	10.50%	11.63%	-0.05%	3.36%	28.29%	0.52%	5.46%	43.54%	0.77%	1.48%	9.66%
	Shell Scripts(8)	15.49%	16.47%	17.32%	9.09%	11.13%	11.12%	11.12%	12.66%	15.76%	7.42%	11.95%	11.25%	0.14%	2.88%	49.31%	0.13%	4.01%	69.23%	-0.16%	0.25%	10.76%
	Syscall	6.47%	7.77%	8.32%	6.90%	6.99%	7.16%	26.69%	27.29%	31.01%	11.68%	12.14%	12.13%	5.68%	5.67%	5.93%	6.11%	6.59%	7.15%	4.83%	5.05%	5.17%
	Index Score	**8.89%**	**9.47%**	**10.98%**	**9.23%**	**9.88%**	**11.10%**	**13.18%**	**13.69%**	**14.80%**	**9.33%**	**9.67%**	**11.67%**	**1.66%**	**3.98%**	**17.18%**	**1.77%**	**7.11%**	**24.59%**	**1.64%**	**1.94%**	**6.72%**
LMBench	null call	1.19%	2.65%	2.13%	0.89%	1.23%	1.13%	56.68%	57.34%	59.65%	25.87%	26.09%	26.05%	0.67%	1.33%	1.33%	1.33%	1.53%	1.80%	-0.66%	0.67%	0.66%
	null I/O	1.78%	2.61%	2.25%	1.30%	1.49%	1.40%	31.66%	31.80%	33.15%	12.64%	12.80%	12.63%	0.00%	0.01%	0.45%	0.00%	0.03%	0.61%	0.00%	0.06%	0.57%
	stat	11.96%	15.42%	15.78%	11.54%	13.72%	13.00%	24.09%	28.53%	30.09%	7.92%	10.97%	12.93%	0.29%	4.29%	4.65%	0.15%	5.81%	6.10%	0.78%	5.52%	6.40%
	open/close	5.40%	11.41%	6.59%	5.51%	12.45%	4.51%	14.81%	18.48%	18.54%	8.78%	12.58%	8.64%	1.58%	51.41%	2.86%	1.29%	76.94%	3.17%	1.92%	21.63%	2.49%
	slct TCP	0.40%	1.23%	0.73%	0.20%	0.35%	0.38%	3.07%	3.98%	3.94%	0.33%	0.54%	0.52%	1.87%	1.80%	1.84%	1.84%	1.76%	1.88%	0.34%	0.73%	0.33%
	sig inst	2.44%	4.48%	4.11%	2.42%	3.15%	2.60%	32.95%	33.31%	35.41%	8.29%	8.68%	8.31%	0.00%	4.26%	4.35%	0.00%	4.47%	4.55%	0.17%	1.74%	1.45%
	fork proc	32.58%	33.71%	35.23%	29.24%	34.43%	36.49%	30.27%	30.62%	31.59%	21.61%	21.01%	25.80%	1.94%	8.15%	59.91%	1.78%	16.18%	98.85%	1.12%	2.07%	25.25%
	exec proc	20.35%	26.82%	28.69%	29.28%	30.14%	31.61%	21.45%	23.76%	25.55%	16.20%	18.86%	21.20%	0.76%	5.95%	50.09%	1.32%	10.70%	89.91%	1.70%	2.01%	20.07%
	Mmap L/T	34.46%	37.38%	39.84%	37.32%	38.06%	39.44%	35.80%	41.09%	42.47%	33.68%	35.21%	35.50%	1.93%	5.17%	61.05%	1.75%	7.70%	101.40%	1.40%	5.56%	32.82%
	Page Fault	42.83%	52.25%	56.67%	53.26%	53.76%	54.36%	51.87%	52.42%	66.86%	41.83%	43.05%	44.67%	0.27%	3.87%	164.46%	0.35%	4.02%	282.81%	0.02%	0.65%	44.61%
	Geo_mean	**4.59%**	**4.79%**	**5.60%**	**3.94%**	**4.39%**	**4.92%**	**7.54%**	**7.80%**	**8.66%**	**3.80%**	**4.26%**	**4.39%**	**0.29%**	**5.24%**	**9.40%**	**0.38%**	**6.24%**	**12.32%**	**0.20%**	**0.69%**	**3.30%**

Table 4: Performance overhead of IEEs when protecting credentials and page tables on UnixBench and LMBench.

measure various system aspects through a series of tests, including CPU speed, memory performance, file system operations, system calls, and so on. LMBench mainly focuses on experimenting with the reaction time of some events and the bandwidth.

Real-world applications. Phoronix [40] integrates performance testing of many real-world applications, and we select some representative ones usually chosen by existing literature [20, 46]. They include Nginx, interpreters of PHP and phython, OpenSSL, etc, testing their latency and throughput.

Comparison targets. We chose existing IEEs mentioned in §4, i.e., TTBRn-based [7], TxSZ-based [11], PAN-based [28], and Watchpoint-based [24], as comparison targets. In addition, we also implemented and compared two common VM-based IEEs: running the IEE alongside the hypervisor and in another VM. These two IEEs also used TIDE's minivisor, and the hypercall implementation adopts the optimized version (shown in Table 1). The IEE that runs in another VM does not completely switch the VM, but only traps the hypervisor to switch the stage-2 page table to achieve better performance.

8.1 Overhead of Gate-Switching

We measured the round-trip latency of a NULL invocation, resulting in 177 CPU cycles. The entry and exit gates introduce a latency of 113 and 64 CPU cycles, respectively. The latency at the entry gate is higher because of the two extra settings of the `SCTLR_EL1`. This is also why the gate's latency of TIDE is higher than that of existing IEEs (shown in Table 1), but it is still much more efficient than hypervisor traps. The latency mainly results from setting system registers in the gate, which requires an instruction synchronization barrier (ISB) that costs 22 cycles. Setting a system register with an `ISB` usually requires 25~30 CPU cycles. To optimize performance, we reduce the use of ISB — continuous system register settings only require one ISB, which means that although the entry gate has five system register settings, it only requires two ISB instructions.

8.2 Micro-Benchmarks

We evaluated the overhead of IEEs in protecting kernel critical structures and running security tools on UnixBench and LMBench. We ran the benchmarks on the Linux kernel equipped with IEEs.

8.2.1 Protecting critical structures. We used IEEs to protect the credentials and page tables, and Table 4 gives the results.

Empty IEEs. The index score in UnixBench and geometric mean in LMBench of performance overhead show that TIDE outperforms existing kernel-level IEEs when empty. In detail, for *Execl Throughput*, *Process Creation*, and *Shell Scripts* in UnixBench, and *fork*, *exec*, *mmap*, and *page fault* in LMBench, the performance advantage of TIDE is more prominent because they cause frequent page table updates, which requires frequent IEE invocations. However, the performance overhead of TIDE and VM-based IEEs is similar because they do not require the page table protection. For *Pipe Throughput* and *Syscall* in UnixBench, and *null call*, *null I/O*, and *signal install* in LMBench, the performance overhead of PAN-based and Watchpoint-based IEE is worse due to the frequent context switches between the user and the kernel triggered by them.

Protecting credentials. TIDE performs much better than the others, whose overheads are only 1.94% and 0.69%. Compared with existing kernel-level IEEs, although the latency of TIDE's gate is higher, the performance overhead of TIDE is still lower. This is because page table updates are much more frequent than credential updates, thereby bringing more IEE invocations, which is the main reason for the higher overhead. Among them, the *open/close* in LMBench is a special case that TIDE performs worse than existing kernel-level IEEs because credentials are updated more frequently, and thus the latency of the gate is the main reason that affects performance. Compared to VM-based IEEs, TIDE has lower performance overhead because the latency of its gate is lower.

Protecting page tables. Although the gate's latency is higher, TIDE is performant than kernel-level IEEs. This is because TIDE does not need a reverse mapping table of each physical page, which validates three security invariants: 1) IEE memory pages cannot be mapped

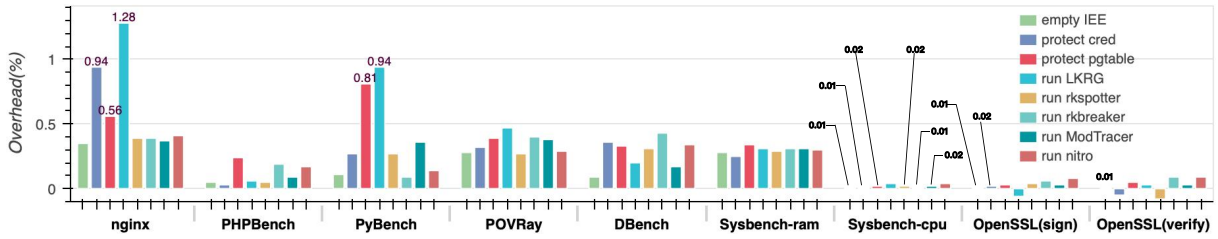

Fig. 6: Performance overhead of TIDE on Phoronix.

UnixBench						LMBench					
TIDE	LKRG	rkspt	rkbrk	MdTrc	nitro	TIDE	LKRG	rkspt	rkbrk	MdTrc	nitro
Dhrystone 2	-0.06%	0.02%	-0.11%	-0.08%	0.07%	null call	0.78%	0.78%	1.30%	1.90%	0.02%
DP WS	0.00%	0.00%	-0.01%	0.01%	0.00%	null I/O	1.57%	2.01%	1.88%	1.97%	0.04%
Execl T/P	2.77%	1.69%	1.87%	1.72%	1.68%	stat	74.37%	4.97%	6.39%	4.90%	5.86%
FC 1024	3.46%	2.04%	2.24%	2.40%	2.52%	open/close	51.15%	3.43%	3.45%	3.27%	3.82%
FC 256	3.22%	2.38%	2.30%	2.19%	2.74%	slct TCP	0.66%	1.91%	1.18%	1.53%	0.37%
FC 4096	4.00%	3.31%	3.47%	3.86%	3.81%	sig inst	3.68%	1.95%	1.61%	1.69%	0.28%
Pipe T/P	1.72%	1.73%	1.85%	1.75%	1.91%	fork proc	1.14%	2.22%	2.26%	1.64%	1.14%
Pipe Switch	2.22%	1.96%	1.15%	1.22%	1.98%	exec proc	4.25%	1.73%	2.15%	2.00%	1.84%
Proc Creat	5.73%	5.69%	5.42%	5.44%	5.74%	Mmap L/T	3.76%	2.27%	2.17%	1.92%	3.20%
Shell(1)	5.49%	0.78%	0.73%	0.76%	0.76%	Page Fault	3.62%	2.55%	5.28%	4.75%	0.84%
Shell(8)	5.26%	0.72%	0.87%	1.33%	0.29%	/					
Syscall	5.56%	5.46%	6.24%	6.46%	5.83%	/					
Index Score	**3.47%**	**1.78%**	**1.84%**	**1.74%**	**1.82%**	**Geo_mean**	**4.77%**	**0.99%**	**0.91%**	**0.98%**	**0.97%**

Base	UnixBench Index Score					/	LMBench Geometric Mean				
TTBRn	**10.68%**	**9.04%**	**9.12%**	**9.03%**	**9.21%**	/	**8.62%**	**5.28%**	**5.18%**	**5.24%**	**5.25%**
TxSZ	**10.97%**	**9.31%**	**9.48%**	**9.31%**	**9.45%**	/	**7.99%**	**4.64%**	**4.50%**	**4.59%**	**4.58%**
PAN	**14.80%**	**13.45%**	**13.50%**	**13.44%**	**13.59%**	/	**11.02%**	**8.16%**	**8.01%**	**8.12%**	**8.13%**
Watchpoint	**10.80%**	**9.64%**	**9.74%**	**9.65%**	**9.74%**	/	**7.43%**	**4.55%**	**4.41%**	**4.45%**	**4.49%**

Table 5: Performance overhead of IEEs when running security tools on UnixBench and LMBench.

to the kernel; 2) privileged executable pages cannot be mapped as writable; 3) kernel read-only data pages cannot be mapped as writable. Thanks to the PXN and write-protection provided by the stage-2 translation, most of the work is done by it. TIDE only needs to protect their regular mappings, which could avoid a large software checking overhead. Similar to the protection of credentials, compared to VM-based IEEs, TIDE has lower performance overhead because the latency of its gate is much lower.

8.2.2 Running security tools. As mentioned in §7, we retrofitted and ran five security tools in kernel-level IEEs. We did not run them on VM-based IEEs because their IEEs are too different, and retrofitting these tools is much harder. The experimental method is to run UnixBench and LMBench to evaluate the performance overhead while running security tools. Table 5 gives the results. We can see that TIDE is performant than other IEEs because the page table protection is not required.

Compared with periodic detecting tools, TIDE incurs more performance overhead when running the callback-based tool, i.e., LKRG. This is because LKRG installs a lot of hooks into the kernel code, which brings more frequent IEE invocations, especially in two cases, i.e., *stat* and *open/close*, of LMBench. Compared with other IEEs, TIDE is very performant because the extra IEE invocations are brought from the page table protection. Like TIDE, other IEEs incur more performance overhead when running LKRG, and the reason is the same.

8.3 Real-world Applications

To evaluate the performance overhead of TIDE on real-world applications, we ran Phoronix on the kernel equipped with TIDE, which protects the credentials and page tables and runs security tools. The results are shown in Fig. 6. The performance overhead incurred by TIDE is very small, and most are less than 1%. This shows that TIDE performs very well on real-world applications.

Among them, the performance overhead on *Sysbench-cpu* and *OpenSSL* is negligible. For *Sysbench-ram*, the performance overhead introduced by TIDE is the same regardless of what is protected and run. This is because the main overhead is caused by the stage-2 translation. This application introduces many TLB misses, which trigger many page table walks, and the stage-2 translation will bring more memory footprints in this process. For other applications, the performance overhead introduced by TIDE positively correlates with the frequency of IEE invocations. Overall, when protecting credentials and page tables and running LKRG, TIDE will result in more IEE invocations, introducing more performance overhead. Among them, for *nginx* and *PyBench*, the performance overhead introduced by TIDE is more significant, but at most it is 1.28%.

9 Discussion

Relying on virtualization. Using virtualization to improve kernel security is common. Even when building IEE, industry and academia have proposed and deployed virtualization-based solutions. For example, Samsung and Huawei phones have built-in RKP [4] and HKIP [34] mechanisms, respectively, to isolate critical kernel structures into the hypervisor for protection; SKEE [7] is a kernel-level IEE based on virtualization. Although TIDE also conducts IEE based on virtualization, it does not introduce traps, which brings very low performance overhead. Since the physical memory access control brought by stage-2 translation could replace some software validations on the updates of page tables, TIDE even has performance advantages over other IEEs when protecting page tables. This further highlights the performance advantage of using virtualization technology to enhance kernel security.

Page-oriented programming. POP [21] attacks can bypass the CFI mechanism deployed in the kernel by maliciously adjusting the mapping order of code pages based on the kernel code integrity protection provided by two-dimensional page table technology, namely MBEC (Mode-based Execute Control) of X86 and PXN of ARM. TIDE relies on the PXN mechanism on stage-2 translation to achieve kernel $W \oplus X$ to prevent attackers from injecting sensitive privileged instructions. Although POP attacks could modify code mappings to construct gadgets at code page boundaries in the ARM architecture, they cannot construct/inject new instructions because ARM64 instructions are fixed-length and address-aligned.

User space occupation problem. TIDE embeds the trust and IEE spaces into the user space, which not only consumes part of the

user space, but also causes the user space to be discontinuous. However, it does not affect the normal memory allocation of the application because this space is pre-allocated and the kernel is informed. Under memory allocation, without specifying a specific location (that is, `mmap` with the `MAP_FIXED` flag), the kernel will not allocate memory at these two spaces. To ensure thread safety and cross-platform compatibility, the kernel requires applications to use `mmap` with the `MAP_FIXED` flag only on memory regions previously occupied/reserved via their memory allocations. Since Linux 4.17, the kernel provides the `MAP_FIXED_NOREPLACE` flag to address the above issue; it never clobbers a preexisting mapped range. In short, as long as the application follows the kernel's requirements, Tide's exclusive use of part of the user space will not affect its normal function. Our experiments also show that when the user space of all Linux processes (including the real application under test) is occupied partially by Tide, the system runs normally.

10 Related Works

Achieving isolation on AArch64. Applications, kernel, and hypervisor running on different exception levels all need isolation to ensure security. For user applications, intra-process isolation is used to restrict least privilege in a process. PANIC [58] leverages PAN to achieve intra-process isolation, while Lightzone [61] runs processes at EL1 and switches by traps into EL2. For the OS kernel, SKEE [7] and Hilps [11] isolate the trusted components by constructing an IEE as mentioned in §2. ILDI [10] just restricts access to the sensitive data by PAN. Granule Protection Check (GPC) introduced by Armv9 could also be used to achieve isolation [64]. It requires traps to EL3 to switch the granule protection table (GPT), which is even slower than traps into the hypervisor. In contrast, DriverJar [56] and KSplit [23] isolate the drivers based on Watchpoint and software. HAKC [38] constructed a multi-domain isolation by MTE (Memory Tagging Extension) and PA (Pointer Authentication). For the hypervisors, SelMon [24] leverages the watchpoint to protect the monitor, and any access outside will trigger an exception.

Achieving isolation on X86. Similar to AArch64, there are works achieving intra-process isolation on X86. Hodor [22] and libmpk [42] use MPK (Memory Protection Keys) to restrict the access permission. SEIMI [55] leverages Supervisor Mode Access Prevention (SMAP), similar to PAN, to achieve intra-process isolation. For isolation in the kernel, GENESIS [28] also uses SMAP, while another common method is to switch page tables. To optimize the switch, Secpod [53] leverages the CR3 target-list to achieve the switch without traps, and XMP [46] uses VMFUNC, which is much more efficient than traps. MPK could also be used for isolation in the kernel, divided into 2 categories: PKU for the user and PKS for the supervisor. BULKHEAD [20] uses the PKS to achieve isolation, while PKU could be used for microkernel components [19]. Furthermore, EPK [18] extends the usable domains isolated by MPK with the help of virtualization.

Enhancing the kernel security. To ensure the security of the kernel, one basic method is to compartmentalize the kernel. Both the protection of sensitive resources and the prevention of vulnerable components could be a choice. For protection, data structures crucial to access control and page tables are mentioned before [9, 26, 28]. Aspects of separation [7, 10, 11], randomization can also be used to protect them. PTRand [14] is not committed to making the resources inaccessible. Instead, it makes the virtual address of the resources randomized. Therefore, the attacker can not get the virtual address of a sensitive resource for access. For prevention from vulnerabilities, the drivers are usually selected [56]. SFI [49, 59] could also be used to limit the fault in one region, preventing it from affecting other ones. Another common practice is to add a security layer for monitoring. It usually relies on hardware supports such as hypervisor [52], Intel SGX [39], or TrustZone [16].

11 Conclusion

Intra-kernel isolation is widely used to enhance kernel security. In this paper, we propose Tide, a new kernel-level IEE leveraging the output address size feature on AArch64 to disallow the page table translations to the IEE memory in the kernel. Tide addresses flexibility and security challenges in using this feature by adopting the stage-2 translation to extend physical address space and enforce access controls, and by introducing a novel entry gate that disables translation to sneak into the IEE securely. The goal of these designs is to offload most software checks, that is required by a generic IEE, to hardware for better performance. Experiments show that Tide outperforms existing kernel-level IEEs and VM-based IEEs in protecting critical kernel structures and running security tools.

Acknowledgments

We want to thank our anonymous reviewers for their valuable comments. This research was supported by the National Natural Science Foundation of China (NSFC) under Grants 62272442, 61902374, U1736208, the CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118), and the Innovation Funding of ICT, CAS under Grant No.E161040.

References

[1] 2013. LMBench Test Suite. (2013). https://lmbench.sourceforge.net.
[2] 2023. UnixBench Test Suite. (2023). https://github.com/kdlucas/byte-unixbench.
[3] Solar Binarly Frank RageLtMan Jakub Vitaly Mikhail Adam, Mariusz. 2024. LKRG. (2024). https://lkrg.org.
[4] Alexandre Adamski. [n. d.]. A Samsung RKP Compendium. https://blog.longterm.io/samsung_rkp.html.
[5] ARM 2024. *Arm Architecture Reference Manual for A-profile architecture.* ARM. https://developer.arm.com/documentation/ddi0487/ka/
[6] ARM 2024. *Arm System Memory Management Unit Architecture Specification.* ARM. https://developer.arm.com/documentation/ihi0070/ga/
[7] Ahmed M. Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. 2016. SKEE: A lightweight Secure Kernel-level Execution Environment for ARM. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016.* The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/skee-lightweight-secure-kernel-level-execution-environment-for-arm.pdf
[8] Shuangpeng Bai, Zhechang Zhang, and Hong Hu. 2024. CountDown: Refcount-guided Fuzzing for Exposing Temporal Memory Errors in Linux Kernel. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security.* ACM, Salt Lake City UT USA, 1315–1329. https://doi.org/10.1145/3658644.3690320
[9] Quan Chen, Ahmed M. Azab, Guruprasad Ganesh, and Peng Ning. 2017. PrivWatcher: Non-bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi (Eds.). ACM, 167–178. https://doi.org/10.1145/3052973.3053029

[10] Yeongpil Cho, Donghyun Kwon, and Yunheung Paek. 2017. Instruction-Level Data Isolation for the Kernel on ARM. In *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*. ACM, 26:1–26:6. https://doi.org/10.1145/3061639.3062267

[11] Yeongpil Cho, Donghyun Kwon, Hayoon Yi, and Yunheung Paek. 2017. Dynamic Virtual Address Range Adjustment for Intra-Level Privilege Separation on ARM. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/dynamic-virtual-address-range-adjustment-intra-level-privilege-separation-arm/

[12] CVE. [n. d.]. Linux CVE details. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33. Accessed: 2025-03-31.

[13] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram S. Adve. 2015. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 191–206. https://doi.org/10.1145/2694344.2694386

[14] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/pt-rand-practical-mitigation-data-only-attacks-against-page-tables/

[15] Tal Garfinkel, Mendel Rosenblum, et al. 2003. A virtual machine introspection based architecture for intrusion detection.. In *Ndss*, Vol. 3. San Diega, CA, 191–206.

[16] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. 2014. Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture. *CoRR* abs/1410.7747 (2014). arXiv:1410.7747 http://arxiv.org/abs/1410.7747

[17] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10379)*, Eric Bodden, Mathias Payer, and Elias Athanasopoulos (Eds.). Springer, 161–176. https://doi.org/10.1007/978-3-319-62105-0_11

[18] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. EPK: Scalable and Efficient Memory Protection Keys. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 609–624. https://www.usenix.org/conference/atc22/presentation/gu-jinyu

[19] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. 2020. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication. In *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 401–417. https://www.usenix.org/conference/atc20/presentation/gu

[20] Yinggang Guo, Zicheng Wang, Weiheng Bai, Qingkai Zeng, and Kangjie Lu. 2025. BULKHEAD: Secure, Scalable, and Efficient Kernel Compartmentalization with PKS. In *Annual Network and Distributed System Security Symposium, NDSS*.

[21] Seunghun Han, Seong-Joong Kim, Wook Shin, Byung Joon Kim, and Jae-Cheol Ryou. 2024. Page-Oriented Programming: Subverting Control-Flow Integrity of Commodity Operating System Kernels with Non-Writable Code Pages. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/han-seunghun

[22] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 489–504. https://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[23] 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 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 613–631. https://www.usenix.org/conference/osdi22/presentation/huang-yongzhe

[24] Jinsoo Jang and Brent Byunghoon Kang. 2020. SelMon: reinforcing mobile device security with self-protected trust anchor. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services* (Toronto, Ontario, Canada) *(MobiSys '20)*. Association for Computing Machinery, New York, NY, USA, 135–147. https://doi.org/10.1145/3386901.3389023

[25] Zheyue Jiang, Yuan Zhang, Jun Xu, Xinqian Sun, Zhuang Liu, and Min Yang. 2023. AEM: Facilitating Cross-Version Exploitability Assessment of Linux Kernel Vulnerabilities. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2122–2137. https://doi.org/10.1109/SP46215.2023.10179305

[26] Juhee Kim, Jinbum Park, Yoochan Lee, Chengyu Song, Taesoo Kim, and Byoungyoung Lee. 2024. PeTAL: Ensuring Access Control Integrity against Data-only Attacks on Linux. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*. ACM. https://doi.org/10.1145/3658644.3690184

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

[28] Seongman Lee, Seoye Kim, Chihyun Song, Byeongsu Woo, Eunyeong Ahn, Junsu Lee, Yeongjin Jang, Jinsoo Jang, Hojoon Lee, and Brent ByungHoon Kang. 2024. GENESIS: A Generalizable, Efficient, and Secure Intra-kernel Privilege Separation. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, SAC 2024, Avila, Spain, April 8-12, 2024*, Jiman Hong and Juw Won Park (Eds.). ACM, 1366–1375. https://doi.org/10.1145/3605098.3635951

[29] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. 2025. SoK: Software Compartmentalization . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 75–75. https://doi.org/10.1109/SP61157.2025.00075

[30] Zhengchuan Liang, Xiaochen Zou, Chengyu Song, and Zhiyun Qian. 2024. K-LEAK: Towards Automating the Generation of Multi-Step Infoleak Exploits against the Linux Kernel. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/k-leak-towards-automating-the-generation-of-multi-step-infoleak-exploits-against-the-linux-kernel/

[31] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. 2022. DirtyCred: Escalating Privilege in Linux Kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 1963–1976. https://doi.org/10.1145/3548606.3560585

[32] linuxthor. 2020. Rootkit Breaker. (2020). https://github.com/linuxthor/rkbreaker.

[33] linuxthor. 2020. Rootkit Spotter. (2020). https://github.com/linuxthor/rkspotter.

[34] lmpalabs. [n. d.]. Shedding Light on Huawei's Security Hypervisor. https://blog.impalabs.com/2212_huawei-security-hypervisor.html#kernel-page-table-writes.

[35] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. 2023. DOPE: DOmain Protection Enforcement with PKS. In *Annual Computer Security Applications Conference, ACSAC 2023, Austin, TX, USA, December 4-8, 2023*. ACM, 662–676. https://doi.org/10.1145/3627106.3627113

[36] Alex Markuze, Shay Vargaftik, Gil Kupfer, Boris Pismenny, Nadav Amit, Adam Morrison, and Dan Tsafrir. 2021. Characterizing, exploiting, and detecting DMA code injection vulnerabilities in the presence of an IOMMU. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 395–409. https://doi.org/10.1145/3447786.3456249

[37] Matheuz. 2025. ModTracer. (2025). https://github.com/MatheuZSecurity/ModTracer.

[38] Derrick Paul McKee, Yianni Giannaris, Carolina Ortega, Howard E. Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. 2022. Preventing Kernel Hacks with HAKCs. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/auto-draft-257/

[39] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*, Ruby B. Lee and Weidong Shi (Eds.). ACM, 10. https://doi.org/10.1145/2487726.2488368

[40] Matthew Tippett Michael Larabel. 2022. Phoronix Test Suite. (2022). http://www.phoronix-test-suite.com.

[41] OSTechNix. [n. d.]. Linux Kernel Source Code Surpasses 40 Million Lines. https://ostechnix.com/linux-kernel-source-code-surpasses-40-million-lines/. Accessed: 2025-01-27.

[42] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 241–254. https://www.usenix.org/conference/atc19/presentation/park-soyeon

[43] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. 2008. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. 233–247. https://doi.org/10.1109/SP.2008.24

[44] Nick L. Petroni and Michael Hicks. 2007. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 103–115. https://doi.org/10.1145/

1315245.1315260
[45] pfohjo. 2013. nitro-ng. (2013). https://github.com/pfohjo/nitro.
[46] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 563–577. https://doi.org/10.1109/SP40000.2020.00041
[47] Jerome H. Saltzer. 1974. Protection and the control of information sharing in multics. *Commun. ACM* 17, 7 (July 1974), 388–402. https://doi.org/10.1145/361011.361067
[48] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. 2024. HECKLER: Breaking Confidential VMs with Malicious Interrupts. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/schl%C3%BCter
[49] 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 *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS 2023, Melbourne, VIC, Australia, July 10-14, 2023*, Joseph K. Liu, Yang Xiang, Surya Nepal, and Gene Tsudik (Eds.). ACM, 469–480. https://doi.org/10.1145/3579856.3590341
[50] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 335–350. https://doi.org/10.1145/1294261.1294294
[51] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 335–350. https://doi.org/10.1145/1323293.1294294
[52] Abhinav Srivastava and Jonathon T. Giffin. 2011. Efficient Monitoring of Untrusted Kernel-Mode Execution. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society. https://www.ndss-symposium.org/ndss2011/efficient-monitoring-untrusted-kernel-mode-execution
[53] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. 2015. SecPod: a Framework for Virtualization-based Security Systems. In *Proceedings of the 2015 USENIX Annual Technical Conference, USENIX ATC 2015, July 8-10, Santa Clara, CA, USA*, Shan Lu and Erik Riedel (Eds.). USENIX Association, 347–360. https://www.usenix.org/conference/atc15/technical-session/presentation/wang-xiaoguang
[54] Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *2010 IEEE Symposium on Security and Privacy*. 380–395. https://doi.org/10.1109/SP.2010.30
[55] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. 2020. SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 592–607. https://doi.org/10.1109/SP40000.2020.00087
[56] Huamao Wu, Yuan Chen, Yajin Zhou, Yifei Wang, and Lubo Zhang. 2023. DriverJar: Lightweight Device Driver Isolation for ARM. In *60th ACM/IEEE Design Automation Conference, DAC 2023, San Francisco, CA, USA, July 9-13, 2023*. IEEE, 1–6. https://doi.org/10.1109/DAC56929.2023.10247974
[57] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1187–1204. https://www.usenix.org/conference/usenixsecurity19/presentation/wu-wei
[58] Jiali Xu, Mengyao Xie, Chenggang Wu, Yinqian Zhang, Qijing Li, Xuan Huang, Yuanming Lai, Yan Kang, Wei Wang, Qiang Wei, and Zhe Wang. 2023. PANIC: PAN-assisted Intra-process Memory Isolation on ARM. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 919–933. https://doi.org/10.1145/3576915.3623206
[59] Zachary Yedidia. 2024. Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 649–665. https://doi.org/10.1145/3620665.3640408
[60] Ming Yuan, Bodong Zhao, Penghui Li, Jiashuo Liang, Xinhui Han, Xiapu Luo, and Chao Zhang. 2023. DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 2849–2866. https://www.usenix.org/conference/usenixsecurity23/presentation/yuan-ming
[61] Ziqi Yuan, Siyu Hong, Ruorong Guo, Rui Chang, Mingyu Gao, Wenbo Shen, and Yajin Zhou. 2024. LightZone: Lightweight Hardware-Assisted In-Process Isolation for ARM64. In *Proceedings of the 25th International Middleware Conference* (Hong Kong, Hong Kong) *(Middleware '24)*. Association for Computing Machinery, New York, NY, USA, 467–480. https://doi.org/10.1145/3652892.3700786
[62] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. 2023. RetSpill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Copenhagen Denmark, 3093–3107. https://doi.org/10.1145/3576915.3623220
[63] Bingnan Zhong and Qingkai Zeng. 2021. SecPT: Providing Efficient Page Table Protection based on SMAP Feature in an Untrusted Commodity Kernel. In *20th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2021, Shenyang, China, October 20-22, 2021*. IEEE, 215–223. https://doi.org/10.1109/TRUSTCOM53373.2021.00045
[64] Qihang Zhou, Wenzhuo Cao, Xiaoqi Jia, Peng Liu, Shengzhi Zhang, Jiayun Chen, Shaowen Xu, and Zhenyu Song. 2025. RContainer: A Secure Container Architecture through Extending ARM CCA Hardware Primitives. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society.

A Secure Memory Areas Ownership Transfer

Once the IPA pages in the IEE and WP memory are insufficient, the *MM agent* will be invoked to apply for a batch of free and continuous (immediate) physical pages from the kernel's buddy system as the memory area to be transferred and notify the IEE. IEE will invoke the minivisor to complete the transfer of the memory area and then register it. Minivisor only accepts invocations from the IEE and performs legitimacy checks on the area to prevent potential confused deputy attacks. The security check policy is that a memory area can only be transferred from the regular memory, which can be done by traversing the stage-2 page table to make sure its corresponding access permissions are *rwx* and *PXN*. Once finished, the mappings of the transferred pages in the regular memory will be canceled. IEE releases memory in the opposite process, with an extra memory zeroing handling. Noted that TIDE transfers 1/32 of the pages from the regular memory in advance to avoid frequent minivisor invoking; when the physical memory is insufficient, TIDE will apply for memory on demand.

B Protection against DMA Attacks

Since attackers may abuse the DMA functionality to bypass isolation [36], it is necessary to restrict DMAs. If a device supports SMMU, the access will be translated based on SMMU page tables. SMMU supports two translation stages (i.e., IOVA→IPA→PA) by controlling the input and output address sizes, similar to MMU [6]. The difference is the base address of page tables, and the translation control is configured to specific data structures in memory: the stream table entry (STE) controls the stage-2 translation and contains a pointer to context descriptors (CDs), which control the stage-1 translation. To prevent the IEE memory from being accessed by DMA, TIDE also uses the configurable output address size feature: the minivisor is responsible for managing the STEs and uses the same stage-2 page table as the MMU; the CDs are placed/isolated into the IEE, and the output address size of the stage-1 translation is configured only to access the regular memory. The WP memory cannot be corrupted naturally during the DMA because it uses the same stage-2 page table. For DMA without using SMMU, TIDE inspects the update of the memory-mapped registers of the DMA controller via isolating them into the IEE to prevent access to the IEE and WP memory.