Dynamic Virtual Address Range Adjustment for Intra-Level Privilege Separation on ARM

Yeongpil Cho, Donghyun Kown, Hayoon Yi, and Yunheung Paek
ECE and ISRC, Seoul National University
{ypcho, dhkwon, hyyi}@sor.snu.ac.kr, ypaek@snu.ac.kr

***Abstract*—Privilege separation has long been considered as a fundamental principle in software design to mitigate the potential damage of a security attack. Much effort has been given to develop various privilege separation schemes where a monolithic OS or hypervisor is divided into two privilege domains where one domain is logically more privileged than the other even if both run at an identical processor privilege level. We say that privilege separation is *intra-level* if it is implemented for software of a certain privilege level without any involvement or assistance of more privileged software. In general, realizing intra-level privilege separation mandates developers to rely on certain security features of the underlying hardware. So far, such development efforts however have been much less focused on ARM architectures than on the Intel x86 family mainly because the architectural provision of ARM security features was relatively insufficient. Unlike on x86, as a result, there exists no full intra-level scheme that can be universally applied to any privilege level on ARM. However, as malware and attacks increase against virtually every level of privileged software including an OS, a hypervisor and even the highest privileged software armored by TrustZone, we have been motivated to develop a technique, named as *Hilps*, to realize true intra-level privilege separation in all these levels of privileged software on ARM. Pivotal to the success of Hilps is the support from a new hardware feature of ARM's latest 64-bit architecture, called `TxSZ`, which we manipulate to elastically adjust the accessible virtual address range for a program. In our experiments, we have applied Hilps to retrofit the core software mechanisms for privilege separation into existing system software and evaluated the performance of the resulting system. According to the experimental results, the system incurs on average just less than 1 % overhead; hence, we conclude that Hilps is quite promising for practical use in real deployments.**

I. Introduction

A variety of system software such as an operating system (OS) and hypervisor has a monolithic design, which integrates its core services into one huge code base, thereby encompassing them all in a single address space and executing them in the same processor privilege level (i.e., ring 0 and VMX-root modes in Intel x86 or svc and hyp modes in ARM). Therefore, bugs, errors and vulnerabilities residing in a fraction of system software can be easily exploited to subvert other parts of or the entire system [12], [21], [49]. *Privilege separation* stemming from the work of Saltzer and Schroeder [33], [34] has been considered as a fundamental principle in software design that can mitigate such a security concern. To enforce this security principle in the design, one intuitive scheme has been isolating critical parts of system software inside another software with higher privilege. However, there are several problems with this scheme. First of all, it conflicts with the common tendency that, in many cases, system software already runs with the highest privilege level in the system. Also, the mandatory involvement of more privileged software in its design usually entails frequent switching between different privilege levels for the security enforcement, which will become a definite cause of the overall system performance degradation. Therefore, to cure these problems, there has been active research on *intra-level* schemes [15], [46], [4] whose aim is to enforce privilege separation without any reliance on or involvement of other privileged levels. In these schemes, system software is broken into typically two domains: the *inner* and *outer*. Since these domains are split from one monolithic body, they are running in the same processor privilege level, but logically graded in a way that the inner domain becomes more privileged than the outer one. In most cases, the inner domain occupies just a small fraction of the original system software, with (logically) higher privilege, and owns exclusive control authority over sensitive system resources, i.e., page tables and system control registers, that are critical to security. Thus, as a trusted computing base, the inner domain can defend the system software from being subverted entirely even if the outer domain is under control of attackers.

For protection of the inner domain, the first requirement is to guarantee that the memory region of one domain is isolated from the other in the same privilege level. To efficiently and completely fulfill this requirement for *intra-level isolation*, one must somehow rely on a special hardware facility that can apply different memory protection policies respectively to both the domains even if they share the same privilege to access the memory space. For memory protection, the conventional hardware support from the memory management unit (MMU) can no longer be expected because the MMU specializes only in enforcing access permissions according to processor privilege levels. In view of this observation, researchers, in their efforts to develop intra-level isolation mechanisms, had to find alternative hardware facilities for memory protection. As examples, there are several seminal studies [15], [46] that rely on the write protection feature of the x86 family architecture to implement their isolation mechanisms. The x86 processor has a special bit, called the *write-protection* (WP) bit, which is, in general, turned on or off in order to restrict or permit

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23024

write access to specific memory regions. In those studies, they make use of the WP bit in a way that they modify page tables to configure the memory regions for the inner domain to be write-protected by the bit. Now when the bit is on, the outer domain, even if it belongs to the same privilege level with the inner domain, cannot tamper with the inner one. Along with the isolation mechanism based on the WP bit, they have also devised an additional mechanism to properly manipulate the bit when the control switches between the two domains. This *domain switching* mechanism works follows. When the control is transferred to the inner domain, the bit is reset (WP = 0), which subsequently allows the inner domain to perform memory operations normally. After which, the bit will be set on when the control returns back to the outer domain in order to protect the inner domain again.

The greatest strength of the WP bit for privilege separation is that it can provide the same write protection for virtually all privilege levels and types of system software on the x86 processor. In modern software architectures, it is by no means unusual for more than one levels of system software to coexist in a system so as to provide versatile services for its users. The most familiar case found in a real system would be that an OS is installed together with a hypervisor. In these systems, as will be discussed in Section II, the attackers aiming to manipulate system resources and take over the control of a system have been able to compromise each and every level of system software. As a means to counteract such potential threats, the WP bit serves by capitalizing on its all-across-level memory protection capabilities. In fact, its versatile capabilities have already been actively used by researchers to diversify their intra-level privilege separation solutions for different levels of privileged software on x86 machines like those for an OS [15] and for a hypervisor [46].

Sadly on the other hand, such active research work has not been conducted for ARM-based machines whose security concerns are of utmost importance in today's mobile market. We ascribe this mainly to want of the ample architectural provision of ARM's security features for memory protection. More specifically, ARM traditionally had no security hardware functionally equivalent to the WP bit. Partially because of this, it was not until recently that some efforts to develop privilege separation solutions for ARM are reported in the literature [4]. But these solutions still have several limitations. Most of all, they are not truly intra-level schemes. For instance, when they design a privilege separation solution for the OS on ARM, they need to depend on more privileged software (i.e., a hypervisor). This is obviously for lack of due hardware to support intra-level isolation in ARM devices, as described above. This results in another limitation that their design to mandate the reliance on the hypervisor in the solution for the OS is no longer viable to solutions for the hypervisor itself or a TrustZone secure OS. Not surprisingly, their existing solutions are all targeting the normal OSes running on several variations of ARM architectures. As malware and attacks on ARM devices increase against virtually every level of privileged software including not a normal OS but also a hypervisor and even a secure OS armored by TrustZone, it is essential to develop a powerful solution that can be used to fully enforce privilege separation at the same time in any level or type of software on the devices.

In this paper, we introduce a novel technique, named as *Hilps*, to implement intra-level privilege separation in every level of privileged software on ARM. At the center of Hilps, there is a hardware field, called TxSZ, which has been first introduced in ARM's latest 64-bit architecture (a.k.a AArch64). Particularly noteworthy is the fact that we could tailor TxSZ to function for memory protection on ARM similarly to the WP bit on x86. For general purpose, we manipulate the values of TxSZ in order to expand or reduce the range of the valid (or accessible) virtual address space dynamically. To realize the two core mechanisms for privilege separation (i.e., intra-level isolation and domain switching), Hilps exploits this hardware function of TxSZ for *dynamic virtual address range adjustment*, as will be explained in Section IV. In short, for intra-level isolation, Hilps initially allocates the inner domain region separately from the memory region reachable from the outer domain. Later when the outer domain takes execution control, Hilps reduces the valid virtual address range so as to leave out the inner domain region, effectively rendering the inner domain unreadable from the outer domain. When the control is transferred to the inner domain, the valid virtual address range is expanded so as to cover the inner domain region again, which will allow the inner domain to operate normally with full access permissions on the entire memory region for both the domains. The ultimate objective of Hilps is to grant these domains asymmetric memory access permissions such that the inner domain in effect becomes more privileged than its counterpart. It is hereby worth empathizing that our TxSZ-based mechanisms can be universally applied to system software regardless of its privilege level. Therefore, we claim that Hilps achieves full intra-level privilege separation on AArch64 at all privilege levels of system software.

To test the feasibility of Hilps for intra-level privilege separation, we have retrofitted both the TxSZ-based isolation and domain switching mechanisms into existing system software running on AArch64, as exhibited in Section IV. Now if developers want to deploy their security applications in our security-enhanced system, they can deploy the applications simply in a protected region of the inner domain. As a result, residing in the secure execution environment, the applications would safely handle their secure transactions or monitor the potentially compromised outer domain. For our experiments, we have implemented intra-level privilege separation in existing system software running on the versatile express V2M-Juno r1 platform [6]. Our experiments (see Section VI) reveal that our TxSZ-based mechanisms for privilege separation are quite efficient in terms of performance, which we credit primarily to the efficacious memory protection support from the underlying TxSZ hardware. The bare system, which is augmented just with our privilege separation mechanisms and yet without any installed security applications, incurs on average less than 1 % overhead for the overall system. Of course, as more security applications are installed, the overhead should increase proportionally. However the experiments also show that it remains reasonably small with normal application loads.

The limitation of our TxSZ-based strategy for intra-level privilege separation is that it cannot be applied to the traditional ARM 32-bit architecture since the TxSZ field is newly equipped in ARM's latest 64-bit architecture. Nevertheless, we still believe that this limitation does not significantly devalue the technical contributions of our research because from the

fact that AArch64 is rapidly becoming the norm for the newest line of ARM processors, it is evident that our solution can be applied to not only today's but also tomorrow's ARM-based computing systems in the market. The technical contributions that we claim in this paper are listed below.

- An introduction of a novel technique, Hilps, that can implement intra-level privilege separation on AArch64 in a variety of system software with different processor privilege levels, such as a normal OS, a hypervisor and a secure OS.
- An intra-level isolation mechanism that protects the inner domain from the outer domain by elastically adjusting the virtual address range.
- A light-weight mechanism that ensures secure switchings between the inner and outer domains.
- A complete realization of Hilps that demonstrates the feasibility of our proposed isolation and switching mechanisms for AArch64 machines.

II. Threat Model and Related Work

In this section, we motivate the need for our approach. For this, we begin by defining the threat model, which shapes the subsequent discussion on the comparison with related work.

A. Threat Model

1) System Software and Security Threats: ARM processors have recently added two hardware extensions for the sake of strengthening support for virtualization and security. The virtualization extension enables to install and take advantage of a hypervisor, which, as a mediator located between an OS and the underlying hardware, facilitates the system to run multiple OSes by controlling interactions passing through it. The security extension, also known as TrustZone, partitions system resources into two worlds: the normal world and the secure world. In the normal world, which corresponds to a conventional execution environment, a normal OS and hypervisor are installed to execute ordinary applications and handle interactions with users. In the secure world, on the other hand, a secure OS is installed to build a trusted execution environment and to securely execute trusted applications that deal with sensitive data.

In summary, different levels of system software, such as a normal OS, hypervisor and secure OS, can coexist in ARM-based systems. This enriches the functionality of the system, but it also increases security risk as it introduces additional attack surface. For example, to subvert the system, attackers may try to compromise a normal OS, which has control authority over system resources. However, in a coexistent system, attackers could achieve the same objective by compromising a hypervisor or a secure OS. Which would be far more fatal because those types of system software run with a higher privilege level than a normal OS. Unfortunately, ensuring the security of system software is an arduous problem, considering that even carefully designed code inevitably contains bugs and vulnerabilities in proportion to its size [28], [10], [29]. In fact, normal OSes have been known to have a number of vulnerabilities [1] and other levels of system software are no exception. Hypervisors have been examined to have vulnerabilities [2], [23], and, recently, it was shown that even secure OSes can be compromised by exploitable vulnerabilities [38], [31], [43], [32].

```
struct tag_TC_NS_SMC_CMD {
    ...
    unsigned int operation_phys;
    ...
};
```

```
int get_sys_time() {
    ....
    // v2 and v3 are holding time values from get_time()
    *(int*)(cmd->operation_phys + 4) = v2;
    *(int*)(cmd->operation_phys + 4) = 1000 * v3;
}
```

Fig. 1. CVE-2015-4422. An example vulnerability of the secure OS.

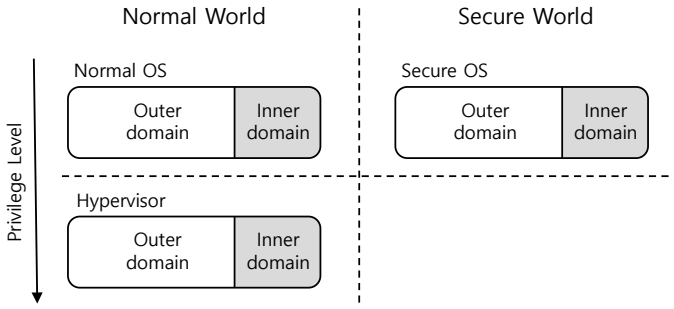


Fig. 2. Full intra-level privilege separation.

For example, Figure 1 describes the pseudo-code of the vulnerability, CVE-2015-4422 [38], that can be used to compromise the secure OS of the Huawei Hisilicon Kirin 925 processor. Note that, generally, a normal OS can communicate with a secure OS through a narrow interface based on the SMC (Secure Monitor Call) instruction. In the example system, the normal OS sends the secure OS a command with the `tag_TC_NS_SMC_CMD` structure, which is allocated in the normal OS. If the secure OS receives a command, then it handles the command and returns a result to the normal OS. In this case, the `get_sys_time()` function of the secure OS returns the current time of the secure world to the normal OS. However, the `get_sys_time()` function does not check the validity of a destination address (`tag_TC_NS_SMC_CMD::operation_phys`) where the time value will be stored. Therefore, attackers in the normal OS can manipulate arbitrary memory locations of the secure OS by sending commands with a crafted `tag_TC_NS_SMC_CMD::operation_phys`.

2) The Ability of Attackers: We assume that all software attacks that would compromise system software can be carried out. By exploiting vulnerabilities, attackers may attempt to do anything allowed within the privilege level of the victim system software. Specifically, they may exploit arbitrary memory read capabilities to disclose secrets in system software. They may also leverage arbitrary memory write capabilities to manipulate control data (return addresses or function pointers) and non-control data to subvert system software. In many cases, their attacks involve tampering with system resources, such as page tables and system control registers, in order to facilitate further attacks by incapacitating protection capabilities of the system.

However, we assume that attackers do not mount side-channel attacks to reveal secrecy of system software. They are also assumed not to launch any types of hardware attacks. Therefore, memory attacks, such as cold boot attacks and

TABLE I. COMPARISON OF HARDWARE-ASSISTED SECURITY SOLUTIONS FOR PRIVILEGED SYSTEM SOFTWARE.

Solution	Need for Higher Priv. Layer	Architecture	Key Hardware Feature	Normal OS	Hypervisor	Secure OS (TrustZone)
SecVisor [36]	Yes	x86 32/64-bit	Extended Paging	Support	Not Support	-
SIM [37]	Yes	x86 32/64-bit	Extended Paging, CR3-Target-List	Support	Not Support	-
SecPod [45]	Yes	x86 32/64-bit	Extended Paging, CR3-Target-List	Support	Not Support	-
HyperSentry [8]	Yes	x86 32/64-bit	System Management Mode	Support	Support	-
TZ-RKP [7]	Yes	ARM 32/64-bit	TrustZone	Support	Support	Not Support
SPROBES [19]	Yes	ARM 32/64-bit	TrustZone	Support	Support	Not Support
Nested Kernel [15]	No	x86 64-bit	Write-Protection	Support	Support	-
HyperSafe [46]	No	x86 64-bit	Write-Protection	Support	Support	-
SKEE [4]	No	ARM 32-bit	TTBCR	Support	Not Support	Support
SKEE [4]	Yes	ARM 64-bit	Extended Paging	Support	Not Support	Not Support
Hilps	No	ARM 64-bit	TxSZ	Support	Support	Support

bus monitoring attacks, are beyond the scope of this paper. Similarly, JTAG attacks are not considered either.

3) Security Guarantee: Hilps implements privilege separation to increase the security level of system software by isolating the inner, secure domain from the outer domain. To defend the outer domain, which potentially be under control of attackers, against corrupting the inner domain region, we ensure the integrity of its `TxSZ`-based mechanisms for intra-level isolation and domain switching. As a result, the code and data (i.e., sensitive system resources and security applications) residing in the inner domain are isolated and protected from the outer domain. Recall that multi levels of system software may run in a single machine. Hilps covers this typical situation by supporting full intra-level privilege separation. Therefore, even if different levels of system software simultaneously operate in the system, Hilps can be applied separately and offer each of them certain security advantage with privilege separation as described in Figure 2.

4) Trusted Computing Base: Hilps is designed to operate based on salient hardware features provided by AArch64. Therefore, we assume that all hardware components do not contain any bugs or vulnerabilities, such as RowHammer bug [22]. We assume that the integrity of the code base belonging to privilege separation mechanisms implemented by Hilps is formally verifiable. That is, there is no exploitable vulnerability in them, and this assumption is not violated during the boot sequence because we also assume that system software that applies our technique is loaded intact with a secure boot mechanism such as AEGIS [41] or UEFI [44].

B. Related Work

By abandoning the monolithic design and modularizing core services, the micro-kernel design [25], [24] can prevent damage of a fraction of the OS kernel from spreading across the whole kernel. Similar attempts have been made on different system software [47], [40]. Through in-depth study [24], [47], the performance of these schemes is almost comparable to current monolithic system software. Nevertheless, the micro-kernel design has been deemed unrealistic in that it usually necessitates a complete remodeling of the current system software architecture. Considering practicality, many research efforts have struggled to enhance the security of the monolithic system software without major modification.

As a more realistic design to strengthen system software security, researchers turn their focus onto an additional security layer in the system, such as a hypervisor [36], [37], [39], [45], SMM of Intel [20], [8], [51], [9], DRTM of Intel TXT [27] or TrustZone of ARM [19], [7], which has a higher privilege over the software they intend to protect. This more privileged layer may own exclusive capability of monitoring and controlling system resources such that it can safely guard sensitive system resources against potential attacks. A clear advantage of this design is that it does not entail substantial changes in the current software architecture. However, earlier studies have commonly agreed that this is inferior majorly in two aspects to the intra-level privilege separation solutions introduced in Section I. One is that it always tends to put its trust on more privileged software for its solution to work, hence being infeasible to solve security problems for the most privileged software. The other is that it suffers from a longer latency, possibly reaching up to thousands of CPU cycles [7], in switching between two different privilege layers.

In light of analysis on other security solutions, a growing number of studies have proposed various intra-level privilege separation schemes attempting to provide alternative solutions. Software Fault Isolation (SFI) [13], [26], [16], [14], one of such schemes, is realized by integrating a number of in-lined reference monitors [35] into system software for exhaustive access control enforcement. SFI boasts its excellent applicability thanks to its hardware independent design, but it has a serious drawback that the system performance degrades proportionally to the amount of instrumented code. To minimize performance degradation, other research has exerted effort to attain high efficiency by taking full advantage of underlying hardware supports in their schemes. To this end, some researchers strive to utilize hardware features for their hardware-assisted memory protection capabilities. As representative examples, Nested Kernel [15] and HyperSafe [46] have successfully evinced the effectiveness of their hardware-assisted privilege separation schemes in terms of performance as well as security. In particular, as explained in Section I, their underlying hardware feature, the `WP` bit, makes their solutions for privilege separation applicable to both an OS and a hypervisor.

Due primarily to the relatively abundant hardware support for security, a majority of privilege separation schemes have been centered on Intel x86 architectures. To the best of our knowledge, SKEE [4] is the only and most notable work to realize privilege separation on ARM's commodity processors today. This is in fact, in our view, the closest work to ours in that our goal is also to find a doable solution for privilege separation on ARM. However, due to the lack of the availability of key hardware features, SKEE can only be applied to limited levels of system software in comparison with Hilps. First, targeting ARM's 32-bit architecture, SKEE capitalizes mainly on Translation Table Base Control Register (`TTBCR`) for dynamic page table activation and successfully implements the two essential mechanisms for isolation and

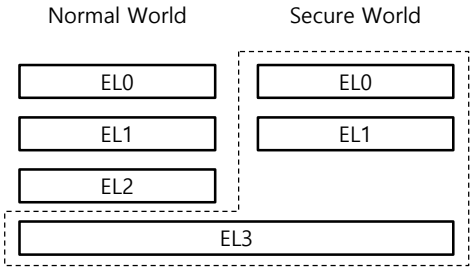


Fig. 3. Exception Levels of AArch64

switching described in Section I. To be more specific, SKEE creates separate page tables for the inner domain and activates them in a timely manner by modifying the N field of TTBCR only when the inner domain is in control. However, as this hardware feature is only defined in the kernel privilege level on AArch32, SKEE is not commonly applicable to different levels of system software, such as hypervisors.

Unfortunately, SKEE faces a similar limitation on ARM's 64-bit architecture. As ARM has abandoned the original TTBCR feature in this new line of processors, SKEE opts for the software-based page table swap technique introduced by Nooks [42]. Recall that this swap technique has an intrinsic security loophole [39], [37] in that attackers can exploit it to load a maliciously crafted page table. To resolve the problem, SKEE resorts to an additional method that hardens the technique in a way to operate deterministically. For this, it is compelled to rely on a complementary technique, called *extended paging*, which must be supported and controlled by a hypervisor. Despite the lack of due hardware support, the solution manages to achieve remarkable performance. It, however, also comes with a major limitation. The mandatory involvement of a hypervisor for extended paging manifests its limitation that the same solution is not viable for privilege separation on the hypervisor itself or secure OS. In contrast, owing to the availability of TxSZ at all levels of system software on AArch64, Hilps can be equally applied to any types of system software.

Table I summarizes the comparison of Hilps and other hardware-assisted security solutions for system software. To compare the coverage of each solution, we deemed that a solution supports a specific type of system software if its key hardware features are available on the privilege level where the system software runs. For example, the TrustZone-based approach of TZ-RKP [7] can be adopted to not only a normal OS but also a hypervisor. Therefore, even if authors of TZ-RKP did not handle a hypervisor in their paper, we considered that TZ-RKP can support a hypervisor as well. Such a comparison clearly shows two advantage of Hilps. First, Hilps does not require the help of higher privileged software such that it does not unnecessarily bloat the size of trusted computing base. Second, Hilps relies on TxSZ, which commonly exists in all processor privilege levels of AArch64, such that it facilitates the enforcement of intra-level privilege separation in all levels of privileged system software.

III. Background

In this section, we provide the background information relevant to our target 64-bit architecture, AArch64.

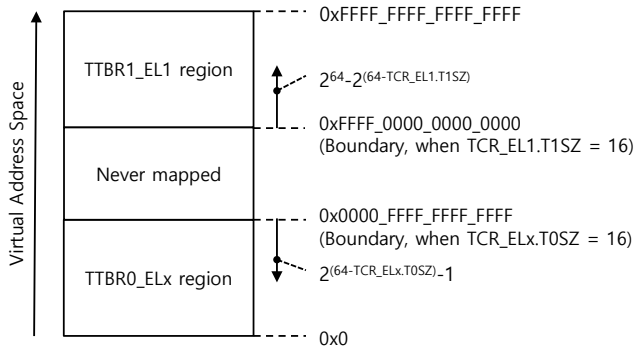


Fig. 4. The change of the virtual address range depending on TCR_ELx.TxSZ

A. Exception Level

In AArch64, the processor privilege levels of AArch32 are mapped onto the *exception levels* (ELx). As described in Figure 3, four exception levels are defined in AArch64 and an exception level with a larger number x corresponds to a higher privilege level. Generally, each exception level is used to execute different levels of software as follows:

- **EL0**: Applications
- **EL1**: Normal OSes and Secure OSes
- **EL2**: Hypervisors
- **EL3**: Secure monitors[1]

B. Virtual Address Range

At every exception level, AArch64 provides two types of core registers for the virtual address translation management. As the first type, Translation Table Base Registers (TTBRs) hold the base physical address of the current page table for mapping between virtual and physical addresses. At level 1, there are two registers, TTBR0_EL1 and TTBR1_EL1, provided for applications and the OS, respectively. At all the other levels, AArch64 supports only one register, that is, *TTBR0_EL2* at EL2 and *TTBR0_EL3* at EL3. As displayed in Figure 4, TTBR0_ELx is used to translate the virtual address space starting from the bottom (0x0), and TTBR1_EL1 is used to translate the virtual address space starting from the top (0xFFFF_FFFF_FFFF_FFFF). The registers of the second type are Translation Control Registers (TCR_ELx) that determine various features related to address translation at each exception level, such as translation granule size, cacheability and shareability. In particular, the two fields T0SZ and T1SZ within TCR_ELx are used to define the valid virtual address ranges that are allowed for virtual-to-physical address translation. Figure 4 depicts how virtual address ranges vary with the values of TCR_ELx.T0SZ and TCR_EL1.T1SZ. Since the current version of AArch64 supports the maximum virtual address range of 48-bit (256 TB) for each TTBR, the virtual address range reaches the boundary when TxSZ is 16, and it varies in inverse proportion to TxSZ. In the current AArch64 Linux, the default value of the TxSZ is 25, indicating that the 39-bit (512 GB) virtual address range is available in each kernel and user space. Once TxSZ is programmed, any memory access exceeding the virtual address range is forbidden, and the system generates a translation fault if violated.

[1]Generally, secure monitors act as a mediator that performs a context switch between the normal and secure worlds.

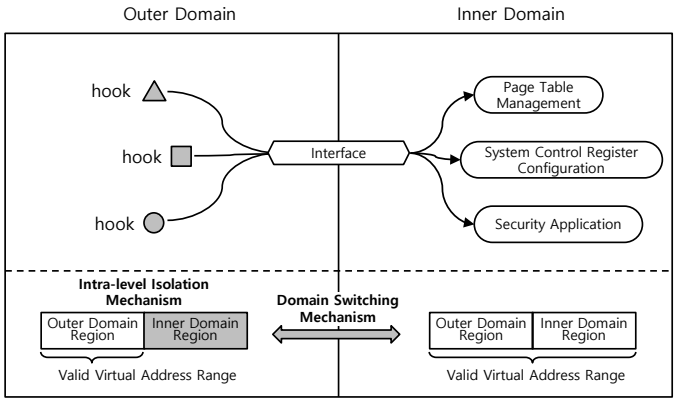


Fig. 5. The overview of our technique that implements intra-level privilege separation, Hilps divides system software into the inner domain (more privileged) and the outer domain (less privileged) by using the intra-level isolation and domain switching mechanisms.

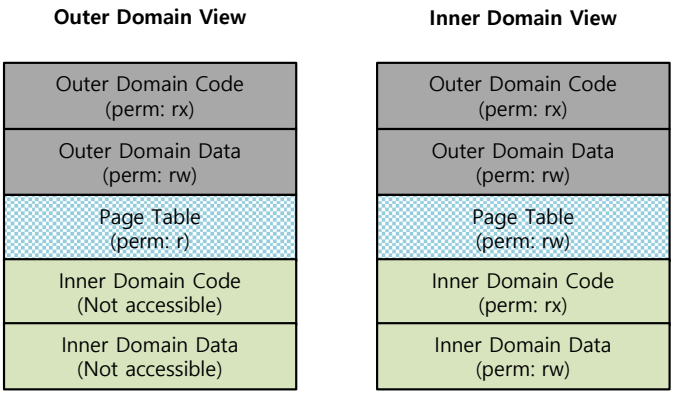


Fig. 6. Address space layout of the inner and outer domains with assigned permissions. Character r, w and x mean read, write and execution, respectively.

C. Translation Lookaside Buffer

A Translation Lookaside Buffer (TLB) is a hardware component that aims to reduce the overhead of address translation by caching recently used virtual-to-physical address mappings. In the situation that multiple tasks run in a system concurrently, cached TLB entries must be flushed at every context switch between tasks in order to prevent them from being exploited by others. However, such frequent flush would induce a substantial increase of TLB miss rates. To eliminate such redundant TLB flushes, therefore, AArch64 supports Address Space Identifier (ASID) at EL0 and EL1[2]. By adding the ASID field to TLB entries, it permits a task to use certain TLB entries holding the same ASID with the current. On AArch64, the current ASID is defined by TTBR. Recall hereby that two registers `TTBR0_EL1` and `TTBR1_EL1` are involved in address translation at EL1. Therefore at this level, AArch64 lets `TCR_EL1.A1` decide which ASID of these registers becomes the current ASID.

However, adopting ASID has its drawbacks as well. In the case of widely shared system resources, such as the OS kernel code and data, caching them in the TLB with multiple ASIDs might degrade performance because it increases TLB pressure. To mitigate this problem, AArch64 presents the non-Global (`nG`) flag in the page table descriptor. By clearing the flag, the corresponding pages come to be seen from every task regardless of their ASID values.

IV. Design

In this section, we present the design and implementation details of Hilps, focusing on the two core mechanisms of privilege separation, i.e., the intra-level isolation and the domain switching.

A. Overview of Hilps

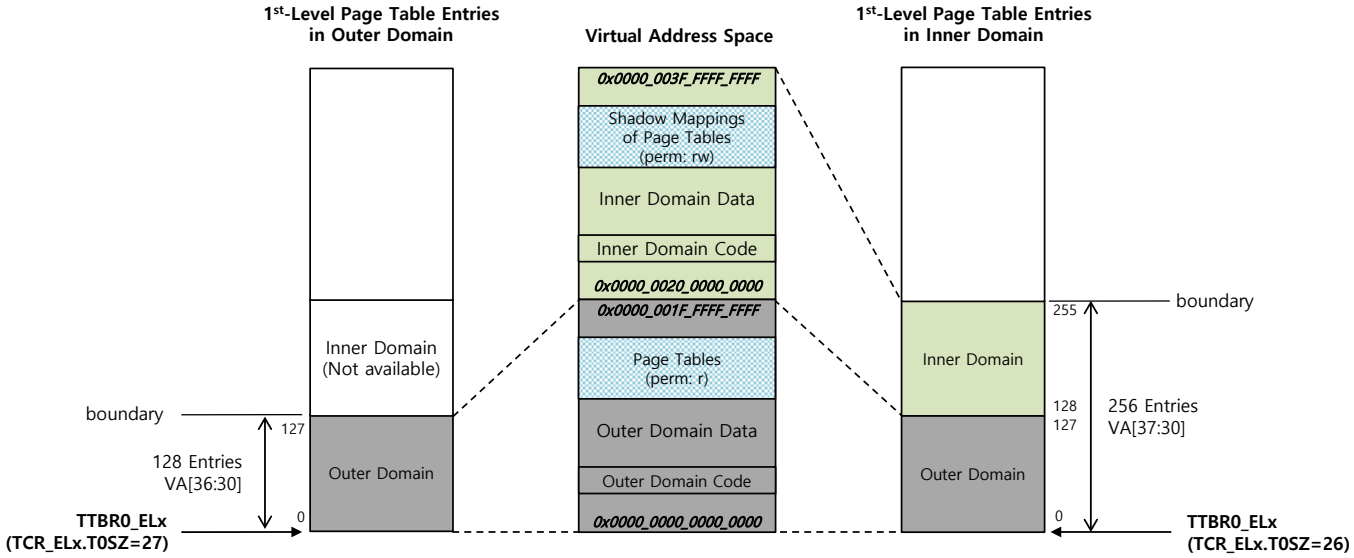
Figure 5 describes our technique that we use to implement the two core mechanisms of privilege separation. We divide system software into the inner and outer domains, adhering to the principle of privilege separation. We implement the intra-level isolation mechanism to protect the inner domain. By dynamically adjusting the range of virtual address space, this mechanism enables the isolation and concealment of memory for the inner domain. This operates in close interaction with the domain switching mechanism whose primary role is to securely transfer control between the two domains. At every moment of control transfer, the former is controlled by the latter in a way to reveal or hide the inner domain accordingly in a timely manner. Basically, the key purpose of these mechanisms is to make the two domains have asymmetric views on the memory address space. For this, as displayed in Figure 6, each domain is assigned different access permissions for memory blocks such that the inner domain obtains unrestricted permissions to access the whole memory region whereas the outer one has more restricted accesses, particularly to the region for the inner domain.

A pivotal condition for the success of this strategy is that the outer domain must not be allowed to manipulate any security-sensitive system resources, such as page tables and system control registers, that may be used to invalidate our core mechanisms. To meet this condition, the outer domain is deprived of control authority over security-sensitive system resources. Instead, the outer domain is only allowed to send requests through a specified interface to the inner domain for controlling these resources. Upon receiving such a request, the inner domain determines whether to accept or reject it. In the sense that a secure entity possesses exclusive control authority over sensitive resources, our intra-level privilege separation technique is as powerful in terms of security strength as a conventional virtualization-based security solution relying on trap-and-emulation. Consequently, if there are security applications demanding a secure environment for execution, the system software redesigned by our technique can offer them protection and monitoring capabilities comparable to the Virtual Machine Introspection (VMI) research [18].

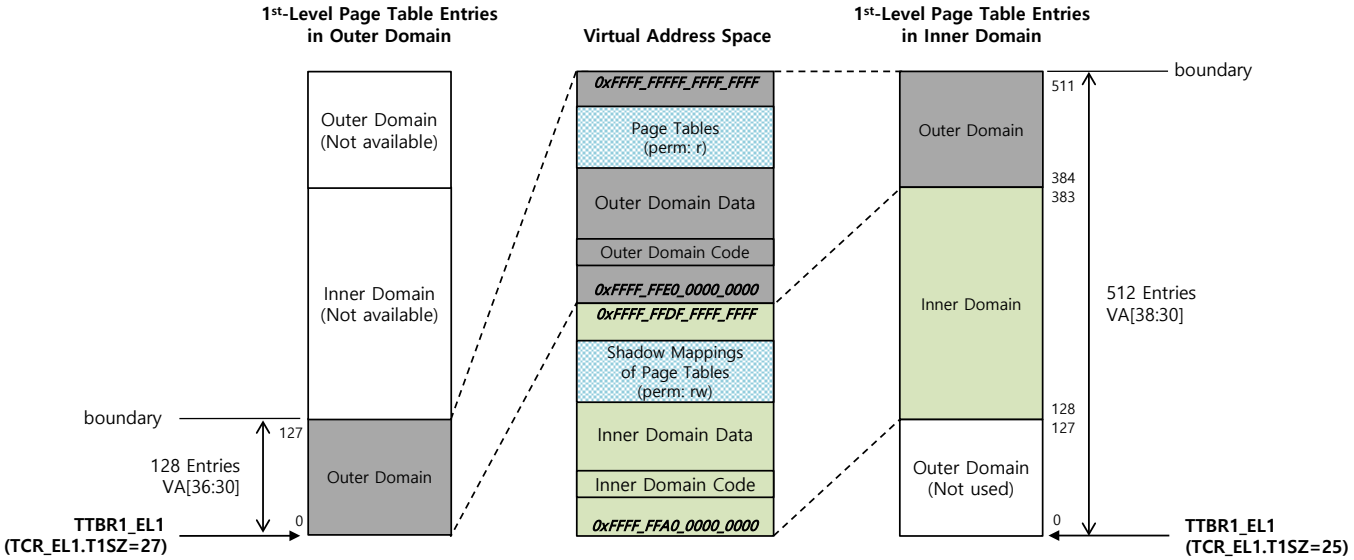
B. Intra-Level Isolation Mechanism

Figure 7 illustrates how we realize the intra-level isolation mechanism based on the dynamic virtual address range adjustment. By either reducing or expanding the range of the valid address space at runtime, Hilps blocks or allows access to the memory region of the inner domain depending on which of the

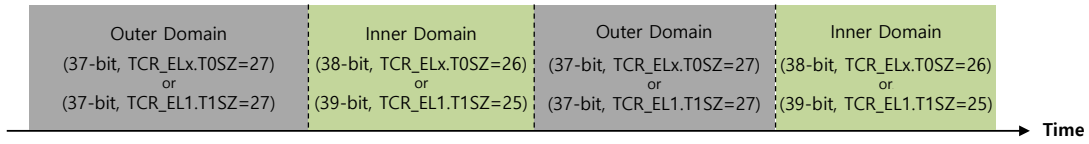
[2]EL2 and EL3 do not support ASID.



(a) The address mapping strategy when system software uses TTBR0



(b) The address mapping strategy when system software uses TTBR1



(c) Timeline of virtual address range adjustment

Fig. 7. The virtual address mapping strategies for isolating the inner domain from the outer domain. A hypervisor and a secure monitor running at EL2 and EL3, respectively, use the mapping strategy **(a)**, and a normal OS and a secure OS running at EL1 use the mapping strategy **(b)**.

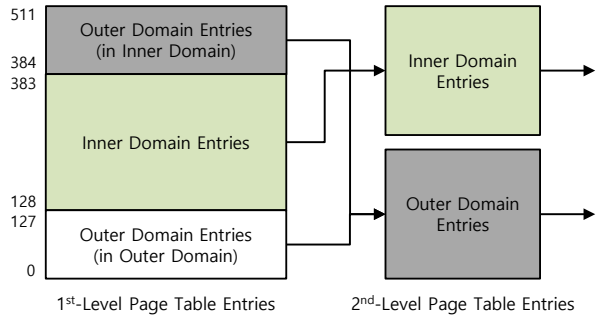


Fig. 8. The linkage between the first-level and second-level page table entries at EL1.

two domains is currently in control. When the outer domain seizes execution control, Hilps leaves the inner domain region out of the valid virtual address range, thereby preventing the inner domain from being exposed and possibly exploited. On the other hand, while the inner domain holds execution control, the valid virtual address range is expanded to cover both the inner and outer domain regions, implying that the inner domain owns a full access coverage reaching the entire memory space.

1) Intra-Level Isolation with TTBR0: Figure 7.(a) describes our general address mapping strategy used to carry out intra-level isolation that can be applied for any system software running with `TTBR0_ELx` at all exception levels ELx, such as a hypervisor (EL2) and secure monitor (EL3). Note that as the two domains share the same page table, the value of `TTBR0_ELx` remains constant whether either the inner or outer domain is in control. While the outer domain has control, its valid virtual address range is restricted to 37-bits[3] (128 GB) by setting `TCR_ELx.T0SZ` to 27. By doing this, all 128 first-level entries of the current page table, indicated by the upper seven bits of the virtual address (VA[36:30]), are used to map the outer domain. In this case, the outer domain cannot access the inner domain because there are no valid entries associated with the memory region of the inner one. On the other hand, when the inner domain has control, the valid virtual address range is expanded to 38-bits (256 GB) by changing `TCR_ELx.T0SZ` to 26. As a result, the number of valid first-level entries of the current page table increases to 256 from 128, and the upper eight bits of the virtual address (VA[37:30]) indicate associated entries. In this case, the original 128 entries still correspond to the memory region of the outer domain, and the expanded 128 entries are used for the inner domain, located outside of the outer domain. Therefore, the inner domain can access the entire memory regions of both domains without restrictions.

2) Intra-Level Isolation with TTBR1: A normal OS and secure OS, which typically run at EL1, adopt `TTBR1_EL1` for address translation on AArch64. Unfortunately, for OSes, we cannot use the general isolation mechanism designed for other system software adopting `TTBR0_ELx`, described in Section IV-B1. That is because the translation style with `TTBR1_EL1` somewhat differs from that with `TTBR0_ELx`. To explain this in more detail, recall our remarks in Section III-B that the virtual address space translated by `TTBR1_EL1` expands in the opposite direction to that by `TTBR0_ELx`. This means that when Hilps increases or decreases the value of `TCR_EL1.T1SZ`, the valid virtual address range and the valid first-level entries of the current page table change in the opposite direction. For example, when `TCR_EL1.T1SZ` is 27, the address 0xFFFF_FFE0_0000_0000 is linked to the 0th first-level entry (see VA[36:30] is 0). Whereas, when `TCR_EL1.T1SZ` is changed to 26, the same virtual address is linked to the 256th entry (see VA[37:30] is 256), and instead, the address 0xFFFF_FFC0_0000_0000 is linked to the 0th entry (see VA[37:30] is 0). Such a discrepancy in the linkage between the virtual addresses and the first-level page entries would lead the same virtual addresses of the inner and outer domains to be mapped to different physical addresses.

To resolve the problem, Hilps introduces an alternative address mapping strategy illustrated in Figure 7.(b). In this case, the valid virtual address range of the inner domain (39-bit, 512 GB) is four times larger than that of the outer domain through a change of `TCR_EL1.T1SZ` to 25 from 27; accordingly, the number of the first-level entries of the current page table increases to 512 from 128. Hilps uses 256 entries in the middle (from the 128th to the 383rd) to map the inner domain region. Note that, even if the bottom 128 entries (from the 0th to the 127th) are originally used to map the outer domain region, Hilps, in the inner domain, does not use these entries to map the outer domain region due to the aforementioned discrepancy problem. Instead, Hilps uses the top 128 entries (from the 384th to the 511st) for this purpose, as, in the inner domain, these entries correspond to the virtual address space of the outer domain region. To do this, Hilps copies the contents of the bottom 128 entries to the top 128 entries, thereby configuring the top first-level entries to point to the same second-level entries that are pointed to by the bottom first-level entries as described in Figure 8. As a result, Hilps can let the inner domain access the outer domain region without the discrepancy problem because the virtual address space of the outer domain region remains the same between the inner and outer domains. Lastly, as the inner domain must be able to maintain a synchronized address view of the outer domain, if the bottom 128 entries of the current page table are populated or modified, Hilps repeats the same operations onto the top 128 entries. This synchronization method does not incur any noticeable overhead as the first-level entries are rarely modified after initial set-up.

3) Shadow Mappings of Page Tables: To prevent the outer domain from modifying the contents of page tables, the memory regions of page tables are configured as read-only. However, as the inner and outer domains share the same page tables, the inner domain also can be hindered by such a restriction. To address this problem, Hilps creates the shadow mappings of the page tables that are configured as not only readable but also writable and locates them by adding a fixed offset in the virtual address space of the inner domain as described in Figure 7. Therefore, the inner domain can update the contents of the page tables through the shadow mappings.

4) Page Table Integrity: To maintain the validity of the intra-level isolation mechanism, Hilps must ensure that the page tables satisfy the following constraints: (1) no part of the inner domain can be mapped to the memory region of the outer

[3]37-bits refers to a quarter of the default virtual address range setting of Linux, but we believe it would be enough for mobile devices.

domain, (2) the outer domain code must be configured as read-only, and (3) privileged instructions that can configure system control registers must not be executable in writable memory regions or in less privileged software's memory regions[4]. To achieve this, Hilps adopts the paging delegation technique used in previous work [7], [45], [15], [4]. Hilps allows only the inner domain to conduct page table modifications after verifying that each modification adheres to these constraints. In order to accomplish this, Hilps initially configures page tables as read-only to prevent the outer domain from modifying them. In addition, it instruments the outer domain code to route all page table modification operations to the inner domain. The inner domain then checks the constraints and performs those operations for the outer domain. Although the page tables are configured as read-only, the inner domain can modify the contents of the page tables through the shadow mapping described in Section IV-B3.

5) Control Authority for System Control Registers: Even if the integrity of the page tables is preserved, our isolation mechanism can still be incapacitated through exploiting system control registers. For example, the outer domain could modify `SCTLR` to remove memory protection by disabling the MMU or `TCR` to enlarge its virtual address range and access the inner domain region. Therefore, Hilps must deprive the outer domain of control authority over system control registers. For this, similar to past research [7], [15], [4], Hilps replaces privileged instructions that control such sensitive registers in the outer domain with hooks so as to verify and emulate them in the inner domain. We can ensure the validity of this method because, first, as instruction opcodes have a fixed length and are aligned on AArch64, Hilps can exhaustively identify privileged instructions. Second, due to the constraints enforced on the page tables mentioned in Section IV-B4, attackers cannot execute any privileged instruction in the outer domain.

6) Support for Multi-core Environment: Our intra-level isolation mechanism relies on the dynamic virtual address range adjustment based on `TxSZ`. Fortunately, as `TCR` containing the `TxSZ` field exists per processor core, Hilps can enforce the intra-level isolation to each core separately by controlling each `TxSZ` of cores. That is, the outer domain is banned from accessing the inner domain region, even if the inner and outer domains simultaneously run on different cores.

7) Support for Loadable Module: Loadable modules, sometimes, can be added to the outer domain to extend functionality (particularly, in a normal OS). However, it can provide attackers with room to compromise the inner domain by inserting privileged instructions relevant to system control registers into the outer domain. To address this problem, if Hilps detects any populations of new code pages or any modifications of code pages from the outer domain, it scans the corresponding pages to confirm whether or not they include privileged instructions and to enforce the protection policy described in Section IV-B5.

8) Restriction on DMA: To improve the performance, peripherals can access DRAM through Direct Memory Access (DMA) without the mediation of the CPU. Unfortunately, it is well known that attackers exploit DMA to avoid the monitoring of a security entity residing in CPU [11], [30]. In the same way, attackers would be able to evade monitors residing in the inner domain.

As a means of thwarting DMA attacks, leveraging IOMMU [3] has been popularly used on x86 systems. On ARM, Hilps can use the System MMU [5] as the counterpart of IOMMU. To accomplish this, Hilps prevents the outer domain from modifying the page tables of the System MMU. Then when a request comes from the outer domain, the inner domain modifies the tables and allows DMA only after ensuring that there is no page table entry pointing to an inner domain region.

However, not all peripherals can take advantage of the System MMU. Even in this case, Hilps needs to restrict the outer domain from controlling DMA directly. On ARM, peripherals can perform DMA with their own custom DMA controller or with the general-purpose DMA controller of the SoC. In either case, the outer domain can only control DMA through memory-mapped control registers. Therefore, similar to DMA protection with the System MMU, Hilps can restrict DMA by only allowing the inner domain to write to regions corresponding to DMA control registers.

C. Domain Switching Mechanism

To transfer control between execution environments with different privilege levels, special instructions, i.e., SVC, HVC or SMC, have been utilized. However, Hilps cannot use this traditional method, as the inner and outer domains run at an identical privilege level. Therefore, Hilps needs to design and implement its own mechanism that performs domain switching by executing a series of ordinary instructions. To achieve this, we create an interface function, `IDC`, which stands for Inner Domain Call. The `IDC` performs the control switching operation between the inner and outer domains, acting as a wrapper function for a handler which processes incoming requests in the inner domain. It provides the outer domain with a unique way to enter the inner domain. In addition, `IDCs` are implanted across the outer domain and are invoked with specific parameters in order to handle sensitive resources by sending relevant requests to the inner domain (refer to Section IV-B4 and IV-B5).

Figure 9 describes the details of the `IDC`. Although this particular implementation is intended to operate at EL1, it is generally applicable to other ELs with slight modifications described in Section IV-C5. The `IDC` is divided into the entry and exit gates. If the `IDC` is invoked in the outer domain, the entry gate disables interrupts, expands the virtual address range, switches to the inner domain stack, and then jumps to the inner domain handler. After the inner domain finishes its work and returns from the handler, the exit gate executes the sequenced tasks of the entry gate in the opposite direction.

1) Virtual Address Range Adjustment: The key role of the `IDC` is to control the valid virtual address range to reveal or conceal the inner domain region depending on the direction of the domain switch. Hence, the entry and exit gates expand and reduce the valid virtual address range by modifying `TCR_ELx.TxSZ`.

The outer domain has no means to modify the value of `TCR` by the restriction described in Section IV-B5. To expose the

[4]This can be achieved by setting the `XN` (eXecute-Never) and `PXN` (Privileged eXecute-Never) bit on the corresponding page table descriptors.

```
1  /**
2   * Inner Domain Call (IDC)
3   *
4   * @param  cmd            a command
5   * @param  [arg0-arg3]    four parameters
6   * /
7  .global  IDC
8  IDC:
9
10 /* The entry gate */
11 mrs    x5, DAIF                    Read interrupt status
12 stp    x30, x5, [sp, #-16]!        Save interrupt status
13 msr    DAIFset, 0x3                Disable interrupts
14 1:
15 mrs    x5, tcr_el1                 Read the current TCR
16 and    x5, x5, #0xffffffffffdfffff ; Set TCR.T1SZ to 25
17 orr    x5, x5, #0x400000           ; Set TCR.A1
18 msr    tcr_el1, x5                 Configure TCR
19 isb                                Instruction synchronization barrier
20
21 mov    x6, #0xc03f                 Check the value of TCR
22 mov    x7, #0x1b                   ; TCR.T1SZ = 25 (39-bit address space)
23 movk   x6, #0xc07f, lsl #16        ; TCR.T0SZ = 27 (37-bit address space)
24 movk   x7, #0x8059, lsl #16        ; TCR.TG1 = 0b10 (4KB page size)
25 and    x5, x5, x6                  ; TCR.TG0 = 0b00 (4KB page size)
26 cmp    x5, x7                      ; TCR.A1 = 1 (Use TTBR1.ASID)
27 b.ne   1b                          If not correct, configure TCR again
28
29 mrs    x6, mpidr_el1               Get number of the current core [0-n]
30 ubfx   x5, x6, #8, #4
31 and    x6, x6, #0xf
32 orr    x6, x6, r5, lsl #2
33 add    x6, x6, #1                  Add 1 to the core number
34 adrp   x5, InnerDomain_stack       Get the base address of the inner domain stack
35 add    x5, x5, x6, lsl #12         Get the inner domain stack of the current core
36 mov    x6, sp                      Get the outer domain stack
37 mov    sp, x5                      Switch to the inner domain stack
38 str    x6, [sp, #-8]!              Save the outer domain stack
39
40 adrp   x5, InnerDomain_handler     Get the address of the inner domain handler
41 blr    x5                          Jump to the inner domain handler
42
43
44
45 /* The exit gate */
46 ldp    x6, [sp], #8                Restore the outer domain stack
47 mov    sp, x6                      Switch to the outer domain stack
48 2:
49 mrs    x5, tcr_el1                 Read the current TCR
50 and    x5, x5, #0xffffffffffbfffff ; Set TCR.T1SZ to 27
51 orr    x5, x5, #0x20000            ; Clear TCR.A1
52 msr    tcr_el1, x5                 Configure TCR
53
54 mov    x6, #0xc03f                 Check the value of TCR
55 mov    x7, #0x1b                   ; TCR.T1SZ = 27 (37-bit address space)
56 movk   x6, #0xc07f, lsl #16        ; TCR.T0SZ = 27 (37-bit address space)
57 movk   x7, #0x801b, lsl #16        ; TCR.TG1 = 0b10 (4KB page size)
58 and    x5, x5, x6                  ; TCR.TG0 = 0b00 (4KB page size)
59 cmp    x5, x7                      ; TCR.A1 = 0 (Use TTBR0.ASID)
60 b.ne   2b                          If not correct, configure TCR again
61
62 ldp    x30, x5, [sp], #16          Restore interrupt status
63 msr    DAIF, x5                    Enable interrupts
64 isb                                Instruction synchronization barrier
65
66 ret                                Return to the outer domain
```

Fig. 9. The detailed implementation of an IDC (Inner Domain Call) at EL1. The IDC consists of the entry gate that expands the valid virtual address range to reveal and enter the inner domain and the exit gate that reduces the valid virtual address range to hide the inner domain and return to the outer domain.

inner domain, therefore, attackers residing in the outer domain may attempt to manipulate TCR by jumping to the TCR control instructions (Line 18 and 52) in the gates. This could be more fatal because TCR consists of several fields having a direct bearing on the security of the inner domain, i.e., TxSZ, Ax and TGx that control the address translation system. Fortunately, all fields of TCR just hold constant values after they are set up at system boot-up. Therefore, even if attackers succeed to manipulate TCR, we can prevent them from accessing the inner domain by not allowing the outer domain to run with a modified TCR. To do this, we insert simple code snippets confirming the correctness of the value of TCR behind TCR control instructions, as seen in Line 21-27 and 54-60 of the IDC code.

2) Interrupt Disabling: The IDC disables interrupts at the entry gate to ensure the atomicity of the gates. It may simply harden our TxSZ-based privilege separation mechanisms by preventing control from being intercepted by the outer domain when the control is in the gates or in the inner domain. However, this can be bypassed if attackers bend the control-flow to skip the interrupt-disabling instructions (Line 11-13) in the entry gate. Attackers then can maliciously generate interrupts to get control (1) immediately after modifying TCR in the entry gate or (2) while the execution of the inner domain. Figure 10 describes how we thwart this attack. We add another code snippet, similar to Line 54-60 of the IDC code, before the interrupt handler. The code snippet checks the value of TCR, and if TCR does not have the value corresponding to the outer domain (that is, interrupts are occurred in the inner domain or in the middle of the IDC), then it halts the system. Additionally, through the method explained in Section IV-B5, we prohibit attackers from modifying VBAR to relocate the interrupt handler, thereby preventing them from evading the explained verification process for TCR.

The overall performance impact of disabling interrupts is limited under the assumption that the inner domain does not run time-consuming security applications. Our evaluation results show that this assumption does not undermine the value of Hilps. Nonetheless, supporting more complex security applications by allowing interrupts remains as one of our goals for future work.

3) ASID Assignment: After control returns back from the inner domain to the outer domain, attackers in the outer domain may be able to eavesdrop on the inner domain region through cached TLB entries storing address mappings of the inner domain. To resolve this problem, the IDC needs to invalidate all TLB entries associated with the inner domain in the exit gate before returning to the outer domain, but this solution is likely to degrade performance as it increases the TLB miss rate. Fortunately, AArch64 features ASID at EL1. Therefore, we can eliminate such expensive TLB invalidations when system software runs at EL1. To achieve this, we configure the address space of the inner domain as non-global and assign a unique ASID to the inner domain. Then the outer domain having a different ASID is restricted from accessing the inner domain through cached TLB entries due to the mis-match of the ASID.

To implement this, Hilps must change the current ASID while switching domains. Considering that, at EL1, ASID is defined by TTBRx_EL1, Hilps needs to change the current ASID by updating the value of TTBRx_EL1. However, performing this in the IDC may weaken the security level of Hilps, as it may expose a sensitive TTBR update instruction to attackers. Therefore, Hilps uses TCR_EL1.A1 to change the current ASID. According to the default setting of AArch64 Linux, TCR_EL1.A1 is 0; i.e., TTBR0_EL1 determines the current ASID. Hilps leaves the outer domain to follow the setting of Linux, but in the inner domain, it lets TTBR1_EL1 determine the current ASID by toggling TCR_EL1.A1 during

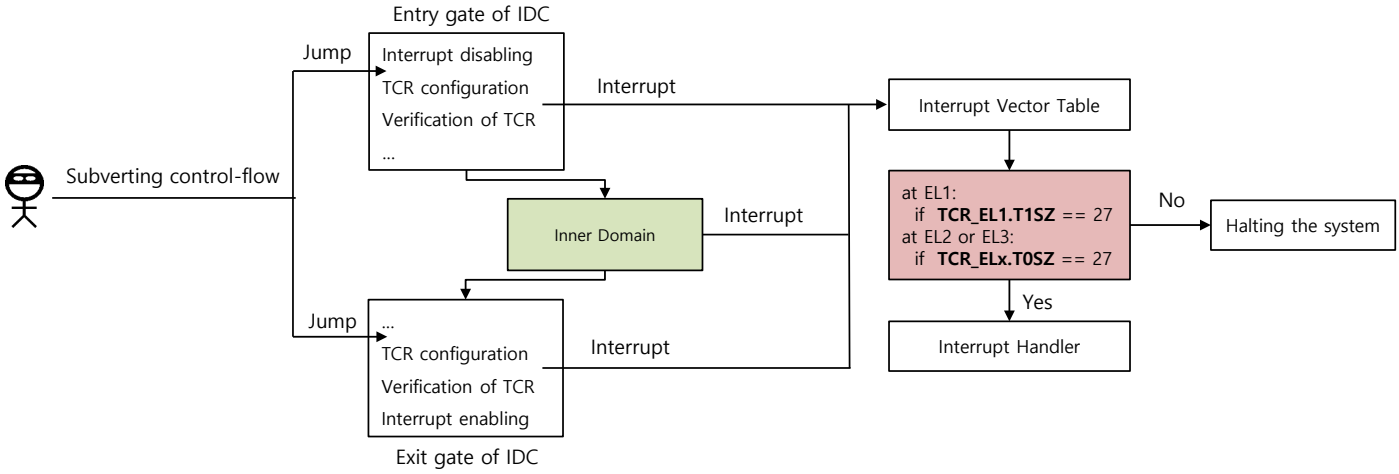


Fig. 10. A defense mechanism for protecting the atomicity of the switching mechanism against control-flow hijacking attacks, initiated from the outer domain.

the domain switching. Recall that TTBRs are managed by the inner domain; thus, Hilps can assign a unique ASID to the inner domain by (1) writing a unique ASID to `TTBR1_EL1` and (2) avoiding the assignment of the same ASID to `TTBR0_EL1`.

4) Inner Domain Stack: The stack is frequently used as a means of attacks, such as code reuse attacks; moreover, many types of critical data are temporally stored in the stack. Thus, the inner domain should use its own stack, separate from the outer domain. Therefore, the entry gate of the `IDC` switches the value of the stack pointer to a preallocated stack in the inner domain (Line 29-38), and the exit gate restores the stack pointer to that of the original one (Line 46-47). Note that, as Hilps supports multi-core environments, the inner domain of each core has its own stack.

5) Port to Different Exception Levels: When we incorporate Hilps into a normal OS or a secure OS running at EL1, we can use the `IDC` as described in Figure 9. However, if Hilps is applied to other levels of system software running at EL2 or EL3, such as a hypervisor and a secure monitor, the `IDC` needs to be modified slightly. Note that the ASID feature is not supported at EL2 and EL3; thus, the `IDC` must perform the TLB invalidation to prevent the inner domain from being revealed to the outer domain. More specifically, in the code of `IDC`, `TCR` control instructions (Line 18 and 52) do not have to change the value of `TCR.A1` and a TLB invalidation instruction (`TLBI*`) must be inserted in Line 61 of the exit gate. Subsequently, according to Section IV-B1, the `IDC` needs to set `TxSZ` to 26 instead of 25 when entering the inner domain (However, this is not mandatory if the inner domain requires a much larger address space).

D. Monitoring Capability

Our intra-level privilege separation technique can make the inner domain provide sufficient monitoring capabilities to security applications residing in the inner domain region. The inner domain initially provides unrestricted memory access for security applications. In addition, it can enable security applications to monitor system behaviors by mimicking the trap-and-emulation technique, widely used in virtualization environments. To achieve this, our technique may employ code instrumentation. By instrumenting the outer domain with `IDCs`, the inner domain can accumulate the behavior of the outer domain, such as system call invocations or system resource accesses, and pass them to security applications for monitoring. Second, as the inner domain has exclusive control authority over page tables, it can enforce access-policies over specific regions of the outer domain. The inner domain can also detect access-policy violations by inserting `IDCs` into the exception handler of the outer domain and can then let security applications inspect these violations.

V. Implementation

In this section, we explain how we implemented a prototype of Hilps to demonstrate the feasibility. The prototype is incorporated into AArch64 Linux Kernel 3.10 of Android 5.1.1

For a prototype implementation, the kernel corresponding to the outer domain should be modified to be deprived of control capabilities for sensitive system resources. For this, we substituted `IDCs` for privileged instructions of the kernel that modify the contents of page tables and sensitive system control registers, such as `TCR`, `TTBR0_EL1`, `TTBR1_EL1`, `VBAR` and `SCTLR`, which can affect the safety of the inner domain. We also modified the kernel to configure all page tables as read-only by setting access permission bits of specific page table entries mapping the memory regions of the page tables. This is necessary to prevent the page tables from being compromised by attackers, but entails a problem in implementation. Actually, as there are mixed-pages containing both page tables and kernel data objects, an access permission modification to the mixed-pages would disturb benign memory operations for the kernel data objects and cause the kernel to crash. To address this problem, we eliminated such mixed-pages by reserving read-only memory regions and allocating page tables from these regions.

TABLE II. Round-Trip Cycles (RTC)

	Big core		Little core	
	w/ ASID	w/ TI	w/ ASID	w/ TI
RTC	424	832	210	249

TABLE III. LMBENCH RESULTS

Test	Big core					Little core				
	Native	Hilps		Overhead		Native	Hilps		Overhead	
		w/ ASID	w/ TI	w/ ASID	w/ TI		w/ ASID	w/ TI	w/ ASID	w/ TI
null syscall	0.44	0.44	0.44	0.00 %	0.00 %	0.43	0.44	0.44	2.33 %	2.33 %
open/close	6.37	6.35	6.44	-0.31 %	1.10 %	12.65	12.67	12.74	0.16 %	0.71 %
stat	2.65	2.64	2.66	-0.38 %	0.38 %	5.06	5.11	5.14	0.99 %	1.58 %
sig. handler inst	0.68	0.68	0.69	0.00 %	1.47 %	0.91	0.91	0.91	0.00 %	0.00 %
sig. handler ovh	3.26	3.27	3.32	0.31 %	1.84 %	5.98	5.94	5.97	-0.67 %	-0.17 %
pipe latency	12.81	14.27	18.38	11.40 %	43.48 %	26.70	28.54	31.80	6.89 %	19.10 %
page fault	1.88	2.40	3.80	27.66 %	102.13 %	2.81	3.69	5.52	31.32 %	96.44 %
fork+exit	148.36	176.84	240.18	19.20 %	61.89 %	255.05	292.21	369.69	14.57 %	44.95 %
fork+execv	163.58	195.35	254.10	19.42 %	55.34 %	279.70	314.50	396.36	12.44 %	41.71 %
mmap	2323.00	2796.00	3992.00	20.36 %	71.85 %	4654.00	5187.00	6718.00	11.45 %	44.35 %

TABLE IV. SYNTHETIC BENCHMARK RESULTS

Test		Native	Hilps		Overhead	
			w/ ASID	w/TI	w/ ASID (σ)	w/ TI (σ)
CF-Bench		42243.5	41111.8	36770.4	2.68 % (11.00)	12.96 % (13.58)
GeekBench	single core	842.6	844.4	840.0	-0.21 % (0.93)	0.31 % (0.97)
	multi core	1891.6	1880.5	1886.0	0.59 % (0.98)	0.30 % (1.64)
Quadrant		8137.9	8092.7	8139.2	0.56 % (1.48)	-0.02 % (1.47)
Smartbench	productivity	4382.3	4291.8	4494.5	2.07 % (10.80)	-2.56 % (4.26)
	gaming	2597.7	2552.4	2563.4	1.74 % (13.45)	1.32 % (9.29)
Vellamo	browser	2895.1	2893.2	2862.6	0.07 % (2.03)	1.12 % (2.63)
	metal	1350.9	1352.6	1348.9	-0.13 % (0.47)	0.15 % (0.57)
Antutu		41033.9	40964.3	40298.7	0.17 % (2.22)	1.79 % (1.83)

To retrofit our technique into the kernel, additional modifications are needed. We defined memory regions for the page tables and the inner domain's code and data by modifying the linker script of the kernel. In the early boot-up sequence (before enabling the MMU), kernel page tables are initialized to map these memory regions following our mapping strategy described in Figure 7. Page table regions are mapped in the outer domain with read-only permission. The inner domain regions are mapped out of the valid address space of the outer domain. Subsequently, the outer domain transfers control to the inner domain by invoking an `IDC`. The inner domain then initializes its data structures and creates the shadow mappings of the page tables to be able to perform page table managements instead of the outer domain. Next, it enables the MMU by configuring `SCTLR` and returns to the outer domain.

Note that the protection scheme for DMA attacks is not considered. However, we do not believe that this will damage the accuracy of the performance evaluation in the next section, as `IDC` invocations for DMA protection would only account for a small portion in the whole execution time.

VI. EVALUATION

In this section, we evaluate our intra-level isolation technique by performing a case study for measuring the performance overhead of the prototype of Hilps, described in Section V. Experiments have been conducted on the versatile express V2M-Juno r1 platform [6], which ships with Cortex-A57 1.15 GHz dual-core processor and Cortex-A53 650 MHz quad-core processor in a big.LITTLE design and 2 GB of DRAM.

Experimental Group. In the case study, as the prototype runs at EL1 by default, our privilege separation technique can benefit from the ASID feature. On the other hand, in the case that the prototype does not run at EL1, the technique needs to rely on TLB invalidation to protect the inner domain, as it can no longer use the ASID feature. In our experiments, therefore, we evaluate both variations of the prototype that operate with ASID and TLB invalidation to help reasonable performance prediction when our privilege separation technique is incorporated into different levels of system software.

A. Switching Overhead

The major portion of overhead of our privilege separation mechanisms is from the transitions between the inner and outer domains by `IDCs`. To investigate the overhead imposed by each `IDC`, we invoked a null `IDC`, which does not perform any operation, and measured the elapsed time using the performance monitor supported by AArch64. In addition, as the big.LITTLE is a prevalent feature in recent ARM-based mobile devices, we performed experiments separately in big and little cores. The experiment was repeated 100 times and the average results are reported in Table II.

The result shows the lightweightness of the `IDC`. Even though big cores operate with about two times faster clock speed, we can see that the `IDC` using ASID consumes near constant time regardless of types of cores. However, the same tendency is not found when the `IDC` invalidates TLB entries. This result is attributed to the different TLB structure of big and little cores.

B. Micro Benchmarks

As the prototype is targetting the normal OS, it could impose a performance penalty on system calls. To measure such overhead, we performed experiments using the LMBench test suite. Similar to the case of measuring RTC of the `IDC`, experiments are done in consideration of the big.LITTLE feature. Table III reports the results for two versions of the prototype together. It shows that the prototype does not slow down null, open/close, stat and signal handling system calls because they do not touch sensitive resources that are managed in the inner domain. Contrarily, the prototype degrades the performance of other system calls that are related to memory management. For example, to handle a page fault (by copy-on-write or demand paging), the outer domain has to modify

TABLE V. LMBENCH RESULTS WITH A SECURITY APPLICATION

Test	Big core					Little core				
	Native	Hilps		Overhead		Native	Hilps		Overhead	
		w/ ASID	w/ TI	w/ ASID	w/ TI		w/ ASID	w/ TI	w/ ASID	w/ TI
null syscall	0.44	0.81	1.28	84.09 %	190.91 %	0.43	1.01	1.66	134.88 %	286.05 %
open/close	6.37	7.28	8.73	14.29 %	37.05 %	12.65	13.82	16.09	9.25 %	27.19 %
stat	2.65	3.09	3.78	16.60 %	42.64 %	5.06	5.79	6.84	14.43 %	35.18 %
sig. handler inst	0.68	1.09	1.64	60.29 %	141.18 %	0.91	1.49	2.19	63.74 %	140.66 %
sig. handler ovh	3.26	3.68	4.44	12.88 %	36.20 %	5.98	6.55	7.45	9.53 %	24.58 %
pipe latency	12.81	19.86	27.84	55.04 %	117.33 %	26.70	40.44	50.04	51.46 %	87.42 %
page fault	1.88	2.38	3.74	26.60 %	98.94 %	2.81	3.73	5.48	32.74 %	95.02 %
fork+exit	148.36	182.54	237.13	23.04 %	59.83 %	255.05	292.61	374.27	14.73 %	46.74 %
fork+execv	163.58	195.19	257.85	19.32 %	57.63 %	279.70	322.22	404.57	15.20 %	44.64 %
mmap	2323.00	2786.00	3878.00	19.93 %	66.94 %	4654.00	5148.00	6641.00	10.61 %	42.69 %

TABLE VI. SYNTHETIC BENCHMARK RESULTS WITH A SECURITY APPLICATION

Test		Native	Hilps		Overhead	
			w/ ASID	w/TI	w/ ASID (σ)	w/ TI (σ)
CF-Bench		42243.5	36218.1	33107.9	14.26 % (5.00)	21.63 % (4.37)
GeekBench	single core	842.6	842.0	839.2	0.07 % (0.54)	0.40 % (1.04)
	multi core	1891.6	1890.6	1882.3	0.05 % (1.11)	0.49 % (1.59)
Quadrant		8137.9	8032.8	8056.8	1.29 % (1.88)	1.00 % (2.28)
Smartbench	productivity	4863.8	4738.4	4253.8	2.58 % (4.60)	12.54 % (6.54)
	gaming	2649.9	2434.2	2613.4	8.14 % (9.58)	1.38 % (12.18)
Vellamo	browser	2895.1	2892.2	2807.2	0.10 % (2.10)	3.04 % (2.32)
	metal	1350.9	1341.7	1341.8	0.68 % (0.65)	0.67 % (0.67)
Antutu		41033.9	40861.1	40307.9	0.42 % (1.92)	1.77 % (2.58)

certain bits of the corresponding page table entries by invoking `IDCs`. The test results also show that the performance impact of TLB invalidations to be more significant, even considering its relatively long RTC. This is attributed to TLB invalidations increasing the TLB miss rate. In summary, the prototype using ASID and TLB invalidation introduce about 8.9 % and 29.5 % performance overhead on average, respectively.

C. Macro Benchmarks

To evaluate the performance impact of the prototype on the overall system, we experimented with six different synthetic benchmark applications that can be publicly downloaded from the Google Play Store: CF-Bench 1.3, GeekBench 3.4.1, Quadrant 2.1.1, Smartbench 1.0.0, Vellamo 3.2 and Antutu 6.0.1. We repeated each benchmark 10 times, and the results are reported in Table IV with a standard deviation. In conclusion, the final benchmark scores reflecting real-world scenarios exhibit the feasibility of the prototype with 0.97 % (when using ASID) and 2.42 % (when using TLB invalidation) performance overhead on average.

D. Security Application Benchmark

If system software adopts our intra-level privilege separation technique, developers can deploy various security applications to monitor the system. It is difficult to deterministically measure or estimate the amount of influence security applications may have on performance. Therefore, instead of struggling to provide general information, we build, as an example, a security application performing system call examination and present its performance impact.

The example security application was created based on the idea of Forrest [17]. It intercepts system calls and extracts high-level information from them. This is relatively simple to implement in our technique, but it provides a useful means for monitoring the system behavior. For example, Aurasium [48] shows that examining the system call data enables a more fine-grained policy enforcement than that of the default permission system of Android. Therefore, our security application mimics Aurasium to monitor the behavior of applications. Specifically, we first inserted `IDCs` in system call handlers to pass system call numbers and arguments to the security application in the inner domain. The transferred data are stored in a ring buffer that are allocated in the inner domain for each core. Then, the security application parses the data to understand the corresponding behavior of applications. For example, by monitoring a system call, `sys_connect`, and its argument, the security application can identify the IP address and port number of a network connection being established, thereby denying applications access to banned websites. Moreover, by monitoring another system call, `sys_ioctl`, and arguments, we can track the binder, which provides an inter-process communication capability to applications. In particular, as applications use the binder to communicate with other applications and service processes, by inspecting established bind connections, the security application can monitor whether applications comply with given access policies for services and resources.

To measure the performance degradation when the security application is installed, we experimented with the same micro and macro benchmarks. The results of Table V show that this security application incurs certain overhead in system calls due to the number of intercepts and parsing operations, but the overhead could be considered negligible in the case of time-consuming system calls such as `mmap`. Table VI shows that the amount of performance overhead imposed by the example security application is acceptable. In conclusion, the performance overhead increases to 3.07 % (when using ASID) and 4.77 % (when using TLB invalidation) on average.

VII. DISCUSSION

In this section, we discuss remaining issues and possible future extensions for Hilps.

Porting Effort. In order to logically deprivilege the outer domain, Hilps entrusts the inner domain with exclusive control

authorities for privileged registers and page tables. We achieve this by adopting a code instrumentation technique that incurs porting cost. For example, we modified about 1800 SLOC of the AArch64 Linux kernel to apply Hilps. According to previous works using this technique [15], [4], [7], [19], such a porting effort is commonly considered reasonable and acceptable.

Vulnerable Security Applications. Attackers may tamper with both the inner and outer domains by exploiting vulnerabilities of security applications. In Hilps, however, it would be extremely difficult to manipulate security applications in such a fashion. One reason is that the outer domain communicates with a security application in the inner domain through a very narrow interface. Another is that only authorized security applications are included in the binary of system software, and they are loaded intact into the system alongside Hilps via a pre-verified secure boot sequence.

On-demand Installation of Security Applications. To cope with attacks which cannot be handled by the security applications existing in the inner domain, installing a new security application would be preferred. In Hilps, updating a firmware image is the only available means for this purpose. However, as it adversely affects the flexibility, we consider allowing security applications to be installed on demand. To enable this, we plan to extend Hilps with two kinds of interfaces respectively supporting the development and installation of security applications. In this case, however, Hilps itself would be threatened if malicious security applications are installed. To relieve this problem, we may need to strongly isolate each security applications to inhibit their influence by using sandbox solutions like NaCl [50].

VIII. Conclusion

Privilege separation has been a popular security principle in the software design that can enhance the security level of monolithic system software. This paper introduces our technique, Hilps, that has been developed to enforce this security principle in system software running on ARM-based machines. The major novelty of Hilps lies in its unique implementation scheme for two underpinning mechanisms, domain switching and intra-level isolation, based on the `TxSZ` hardware field for dynamically adjusting virtual address ranges of running software. Thanks to ARM's new salient hardware support, Hilps has been used to securely incorporate various security solutions for the first time into all levels of privileged software on AArch64, including a normal OS, a hypervisor and even an ARM TrustZone secure OS. In addition, the paper argues for practical use of our technique in real deployments by presenting our experimental evidence that the extra runtime overhead incurred by Hilps is acceptably small. Considering that AArch64 is the standard architecture for ARM's new generation 64-bit processors, we suggest that our technique would be a viable tool to efficiently enforce privilege separation on commodity mobile devices in the future as well as the present.

Acknowledgment

We thank anonymous reviewers for the support and insightful remarks that improved the paper. This work was partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No. R0190-16-2010, Development on the SW/HW modules of Processor Monitor for System Intrusion Detection) and (No. R-20160222-002755, Cloud based Security Intelligence Technology Development for the Customized Security Service Provisioning), the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIP) (No. 2014R1A2A1A10051792), and the Brain Korea 21 Plus Project in 2017.

References

[1] "Linux kernel vulnerabilities," http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[2] "Xen: Vulnerability statistics," http://www.cvedetails.com/vendor/6276/XEN.html.

[3] D. Abramson, "Intel virtualization technology for directed i/o," *Intel technology journal*, 2006.

[4] R. B. J. M. W. S. R. W. Ahmed M. Azab, 1 Kirk Swidowski and P. Ning, "Skee: A lightweight secure kernel-level execution environment for arm," in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[5] ARM, "System memory management unit (smmu)," http://www.arm.com/products/system-ip/controllers/system-mmu.php.

[6] ——, "Versatile express juno r1 development platform," in *ARM 100122_0100_00_en*, 2015.

[7] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[8] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "Hypersentry: enabling stealthy in-context measurement of hypervisor integrity," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[9] A. M. Azab, P. Ning, and X. Zhang, "Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[10] V. R. Basili and B. T. Perricone, "Software errors and complexity: an empirical investigation," *Communications of the ACM*, 1984.

[11] M. Becher, M. Dornseif, and C. N. Klein, "Firewire: all your memory are belong to us," *Proceedings of CanSecWest*, 2005.

[12] J. Bickford, R. O'Hare, A. Baliga, V. Ganapathy, and L. Iftode, "Rootkits on smart phones: attacks, implications and opportunities," in *Proceedings of the 11th workshop on mobile computing systems & applications*, 2010.

[13] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proceedings of the 22nd ACM SIGOPS symposium on Operating systems principles*, 2009.

[14] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[15] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[16] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[17] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings of the 17th IEEE Symposium on Security and Privacy*, 1996.

[18] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of the Network and Distributed System Security Symposium*, 2003.

[19] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," 2014.

[20] Intel, "Trusted execution technology: Software development guide," 2008.

[21] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[22] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[23] S. T. King and P. M. Chen, "Subvirt: Implementing malware with virtual machines," in *Proceedings of the 27th IEEE Symposium on Security and Privacy*, 2006.

[24] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the 22nd ACM SIGOPS symposium on Operating systems principles*, 2009.

[25] J. Liedtke, "On micro-kernel construction," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[26] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Software fault isolation with api integrity and multi-principal modules," in *Proceedings of the 23rd ACM SIGOPS Symposium on Operating Systems Principles*, 2011.

[27] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proceedings of the ACM European Conference in Computer Systems*, 2008.

[28] S. C. Misra and V. C. Bhavsar, "Relationships between selected software measures and latent bug-density: Guidelines for improving quality," in *Computational Science and Its ApplicationsICCSA*, 2003.

[29] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *ACM SIGSOFT Software Engineering Notes*, 2002.

[30] D. R. Piegdon and L. Pimenidis, "hacking in physically addressable memory," in *Seminar of Advanced Exploitation Techniques, WS 2006/2007*, 2007.

[31] D. Rosenberg, "Qsee trustzone kernel integer overflow," in *Black Hat USA*, 2014.

[32] T. Roth, "Next generation mobile rootkits," in *Hack In Paris*, 2013.

[33] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, 1974.

[34] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, 1975.

[35] F. B. Schneider, G. Morrisett, and R. Harper, "A language-based approach to security," in *Informatics*, 2001.

[36] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[37] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[38] D. Shen, "Attacking your trusted core: Exploiting trustzone on android," in *Black Hat USA*, 2015.

[39] A. Srivastava and J. T. Giffin, "Efficient monitoring of untrusted kernel-mode execution," in *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[40] U. Steinberg and B. Kauer, "Nova: a microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European conference on Computer systems*, 2010.

[41] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "Aegis: architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th annual international conference on Supercomputing*, 2003.

[42] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[43] J. Thomas and N. Keltner, "Here be dragons," in *RECON Canada*, 2014.

[44] E. Unified, "Inc. unified extensible firmware interface specification," 2014.

[45] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou, "Secpod: a framework for virtualization-based security systems," in *USENIX Annual Technical Conference*, 2015.

[46] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[47] C. Wu, Z. Wang, and X. Jiang, "Taming hosted hypervisors with (mostly) deprivileged execution." in *Proceedings of the Network and Distributed System Security Symposium*, 2013.

[48] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Proceedings of the 21st USENIX Security Symposium*, 2012.

[49] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[50] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[51] F. Zhang, J. Wang, K. Sun, and A. Stavrou, "Hypercheck: A hardware-assisted integrity monitor," *Dependable and Secure Computing, IEEE Transactions on*, 2014.