This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3192524

Harnessing the x86 Intermediate Rings for Intra-Process Isolation

Hojoon Lee*, Chihyun Song†, Brent Byunghoon Kang†
*Department of Computer Science and Engineering, Sungkyunkwan University
†Graduate School of Information Security, KAIST
hojoon.lee@skku.edu, {chihyun.song,brentkang}@kaist.ac.kr

Abstract—Modern applications often involve the processing of sensitive information. However, the lack of privilege separation within the user space leaves sensitive application secrets such as cryptographic keys just as unprotected as a "hello world" string. Cutting-edge hardware-supported security features are being introduced. However, the features are often vendor-specific or lack compatibility with older generations of the processors. The situation leaves developers with no portable solution to incorporate protection for the sensitive application component. We propose LOTRx86, a fundamental and portable approach for user-space privilege separation. Our approach creates a more privileged user execution layer called *PrivUser* by harnessing the underused intermediate privilege levels on the x86 architecture. The PrivUser memory space, a set of pages within process address space that are inaccessible to user mode, is a safe place for application secrets and routines that access them. We implement the LOTRx86 ABI that exports the `privcall` interface to users to invoke secret handling routines in PrivUser. This way, sensitive application operations that involve the secrets are performed in a strictly controlled manner. The memory access control in our architecture is *privilege-based*, accessing the protected application secret only requires a change in the privilege, eliminating the need for costly remote procedure calls or change in address space. We evaluated our platform by developing a proof-of-concept LOTRx86-enabled web server that employs our architecture to securely access its private key during an SSL connection. We conducted a set of experiments, including a performance measurement on the PoC on *both* Intel and AMD PCs, and confirmed that LOTRx86 incurs only a limited performance overhead.

Index Terms—privilege separation; memory protection; operating system

1 Introduction

User applications today are prone to software attacks, and yet are often monolithically structured or lack privilege separation. As a result, adversaries who have successfully exploited a software vulnerability in an application can access sensitive in-process code or data that are irrelevant to the exploited module or part of the application. Today's applications often contain secrets that are too critical to reside in the memory along with the rest of the application contents, as we have witnessed in the incident of HeartBleed [1], [2].

The conventional software privilege model that coarsely divides the system privilege into only two levels (user-level and kernel-level) has failed to provide a fundamental solution that can support privilege separation in user applications. As a result, critical application secrets such as cryptographic information are essentially treated no differently than a "hello world" string in user memory space. When the control flow of a running user context is compromised, there is no access control left to prevent the hijacked context to access arbitrary memory addresses.

Many approaches have been introduced to mitigate the challenging issue within the boundaries of the existing application memory protection mechanisms provided by the operating system. A number of works proposed using the process abstraction as a unit of protection by separating a program into multiple processes [3], [4], [5]. The fundamental idea is to utilize the process separation mechanism provided by the OS; these works achieve privilege separation by splitting a single program into multiple processes. However, this process-level separation incurs a significant overhead due to the cost of the inter-process communication (IPC) between the processes or address space switching that incur TLB flushes. Also, the coarse unit of separation still leaves a large attack surface for attackers. The direction has advanced through a plethora of works on the topic. One prominent aspect of the advancements is the granularity of protection. Thread-level protection schemes [6], [7], [8] have reduced the protection granularity compared to the process-level separation schemes while still suffering from performance overhead due to their sole dependence on page table modifications for memory protection. Shreds presented fine-grained in-process memory protection using a memory partitioning feature that has long been present in ARM called *Memory Domains* [9]. However, the feature has been deprecated in the 64-bit execution mode of the ARM architecture (`AARCH64`).

In the more recent years, a number of processor architecture revisions and academic works have taken a more fundamental approach to provide in-process protection; Intel has introduced *Software Guard Extensions (SGX)* to its new x86 processors to protect sensitive application and code and data from the untrusted userspace as well as the possibly malicious kernel [10], [11].

Intel also offers hardware-assisted in-process memory safety and protection features [12], [13]. However, the support for the new processor features is fragmented; the features are not interoperable across processors from different vendors (Intel, AMD), and they are also only available on certain Intel processor models. Hypervisor-based application memory protection [14], [15] may serve to be a more portable solution, considering the widespread adaption of hypervisors nowadays. However, it is not reasonable for a developer to assume that her users are using a virtual machine.

The situation presents complications for developers who need to consider the *portability* as well as the security of the sensitive data their program processes. Therefore, we argue that there is a need for an approach that provides a basis for an *in-process* privilege separation based on only the portable features of the processor.

In this paper, we propose a novel x86 user-mode privilege separation architecture called *The Lord of the x86 Rings* (LOTRx86) architecture. Our architecture proposes a drastically different, yet portable approach for user privilege separation on x86. While the existing approaches sought to retrofit the memory protection mechanisms within the boundaries of the OS kernel's support, we propose the creation of a more privileged user layer called `PrivUser` that protects sensitive application code and data from the *normal* user mode. For this objective, LOTRx86 harnesses the underused x86 intermediate Rings (Ring1 and Ring2) with our unique design that satisfies security requirements that define a distinct privilege layer. The `PrivUser` memory space is a subset of a process memory space that is accessible to when the process context is in `PrivUser` mode but inaccessible when in user mode. In our architecture, user memory access control is *privileged-based*. Therefore, The architecture minimizes the costly run-time page table manipulations and address space switching.

We also implement the LOTRx86 ABI that exports the `privcall` interface that supports `PrivUser` layer invocation from user layer. To draw an analogy, the `syscall` interface is a controlled invocation of kernel services that involve kernel's exclusive rights on sensitive system operations. In our architecture, `PrivUser` holds an exclusive right to application secrets and sensitive routines with a program, and user layer must invoke `privcalls` to enter `PrivUser` mode and perform sensitive operations involving the secrets in a strictly controlled way. Our architecture allows developers to protect application secrets within the `PrivUser` memory space and also write `privcall` routines that can securely process the application secret. We developed a kernel module that adds the support for the `privcall` ABI to the Linux kernel (`lotr-kmod`). In addition, we provide a library (`liblotr`) that provides the `privcall` interface to the user programs and C macros that enable declaration of `privcall` routines, a modified C library for the building the `PrivUser` side (`lotr-libc`), and a tool for building LOTRx86-enabled program (`lotr-build`).

We implemented a prototype of our architecture that is compatible with *both* Intel and AMD's x86 processors. We developed a proof-of-concept LOTRx86-enabled webserver. In our PoC, the web server's private key is protected in the `PrivUser` memory space, and the use of the key (e.g., sign a message with the key) is only allowed through our `privcall` interface. In our PoC web server, the in-memory private key is inaccessible outside the `privcall` routines that are invoked securely. Hence arbitrary access to the key is automatically thwarted (i.e., HeartBleed). The evaluation of the PoC and other evaluations are conducted on both Intel and AMD PCs. We summarize the contributions of our LOTRx86 architecture as the following:

- We propose a portable privileged user mode architecture for sensitive application code and data protection that eliminates or minimizes address switching or run-time page table manipulation.
- We introduce the `privcall` ABI that allows user layer to invoke the `privcall` routines in a strictly controlled way. We also provide necessary software for building an LOTRx86-enabled software.
- We developed a PoC LOTRx86-enabled web server to demonstrate the protection of in-memory private key during SSL connection.

2 Background: The x86 Privilege Architecture

The LOTRx86 architecture design leverages the x86 privilege structures in a unique way. Hence, it is necessary that we explain the x86 privilege system before we go further into the LOTRx86 architecture design. In this section, we briefly describe the x86 privilege concepts focusing on the topics that are closely related to this paper.

2.1 The Ring Privileges

Modern operating systems on the x86 architecture adopt the two privilege level models in which user programs run in Ring3 and kernel in Ring0. The x86 architecture, in fact, supports four privilege layers – Ring0 through Ring3 where Ring0 is the highest privilege on the system. The x86 architecture's definition of privilege is closely tied to a feature called *segmentation*.

Segmentation divides virtual memory spaces into *segments* which are defined by a base address, a limit, and a *Descriptor Privilege Level (DPL)* that indicates the required privilege level for accessing the segment. A segment is defined by *segment descriptor* in either *Global Descriptor Table (GDT)* or *Local Descriptor Table (LDT)*. The privilege of an executing context is defined by a 16-bit data structure called *segment selector* loaded in the *code segment register* (`%cs`). The segment selector contains an index to the code segment in the descriptor table, a bit field to signify which descriptor table it is referring to (GDT/LDT), and a 2-bit field to represent the *Current Privilege Level (CPL)*. The CPL in `%cs` is synonymous to the context's current Ring privilege number.

The privilege level (the Ring number) dictates an executing context's permission to perform sensitive system operations and memory access. Notably, the execution of privileged instructions is only allowed to contexts running with Ring0 privilege. Also, the x86 paging only permits Ring0-2 to access supervisor pages.

2.2 Memory Protection

Operating systems use paging to manage memory access control, and the segmented memory model has long been an obsolete memory management technique. However, the paging-based *flat memory model*, which has become the standard memory management scheme, uses the Ring privilege levels for page access control. The x86 paging defines two-page access privileges: User and Supervisor. Ring 3 can only access User pages, while Ring 0-2 are allowed to access Supervisor pages. In general, the pages in the kernel memory space are mapped as Superuser such that they are protected from user applications. Table 1 outlines the privileges of each Ring level.

Algorithm 1 x86 callgate operation

```
1:  procedure CG:R_n → R_m(SEGSEL)
2:      DESC_TBL ← if SEGSEL.ti ? LDT : GDT
3:      CG ← DESC_TBL[SEGSEL.idx]
4:      if n > CG.RMPL or n ≤ m then
5:          return DENIED
6:      end if
7:      Save(%RIP,%CS,%RSP,%SS) ▷ Save caller context in temp space
8:      %SS ← TSS[m].SS ▷ Load new context to be used in Ring m
9:      %RSP ← TSS[m].RSP
10:     %CS ← CG.TargetCS ▷ Privilege Escalation: n → m
11:     %RIP ← CG.TargetEntrance
12:     Push SavedSS
13:     Push SavedRSP
14:     Push SavedCS
15:     Push SavedRIP
16:     RESUME
17: end procedure
```

Algorithm 2 x86 long return instruction (`%lret`)

```
1:  procedure LONG RETURN
2:      ▷ can only return to equal or lower privileges
3:      if DestPriv < CurrentPriv then
4:          return DENIED
5:      end if
6:      %RIP ← Pop() ▷ target addr
7:      %CS ← Pop() ▷ target ring privilege
8:      tempRSP ← Pop()
9:      tempSS ← Pop()
10:     %RSP ← tempRSP
11:     %SS ← tempRSP
12:     RESUME
13: end procedure
```

2.3 Moving Across Rings

The x86 architecture provides a number of mechanisms by which a running context can explicitly invoke privilege escalation for system services. While the privilege of the context is clearly specified in its `%cs` register, its contents cannot be directly altered (e.g., mov `%eax`, `%cs`) but indirectly with special instructions. The x86 ISA provides special instructions that allow switching of the code segment as well as the program counter, namely the *inter-segment control transfers* instructions. For instance, the execution of the *syscall* instruction elevates the `CPL` of the context to Ring0 by loading the `%cs` with the kernel code segment. It also loads the PC register (`%rip`) with the system call entrance point in the kernel. In modern operating system kernels, only the instructions that invoke system calls are frequently used. However, it is necessary that we explain the concepts and mechanisms of the inter-segment control transfer mechanisms that were introduced along with the four Ring system long before the instructions dedicated to invoking system calls

TABLE 1: Privileges of Four Rings on x86

	Ring0	**Ring1**	**Ring2**	Ring3
Privileged instruction	✓	×	×	×
Supervisor page access	✓	✓	✓	×

Privilege escalation. Our design makes use of the *callgate* mechanism for privilege escalation, a feature present in all modern (since the introduction of the protected mode) x86 processors. A callgate descriptor can be defined at the descriptor tables to create an inter-privilege tunnel between the Rings. Specifically, it defines the target code segment, whose privilege will be referred to as the *Target Privilege Level (TPL)*, a *Target Addr*, and a *Required Minimum Privilege Level (RMPL)*. A context can pass through a call gate via a long call instruction[1] that takes a *segment selector* as its operand. The long call instruction first performs privilege checks when it confirms that the operand given is a reference to a callgate. A callgate demands its caller's CPL (the current Ring number) to be numerically equal to or lower than (higher privilege) the callgate's RMPL. Also, the caller's CPL cannot be numerically less than the TPL of the callgate. In other words, a control transfer through a callgate does not allow privilege de-escalation. If these privilege checks fail, the context receives a general protection fault and is forced to terminate. If the privilege check is successful, the privilege of the context is escalated, and the program counter (`%rip`), as well as the stack pointer (`%rsp`), are loaded with the target address. A long call instruction results in privilege escalation if and only if it references a valid callgate that defines a privilege escalation and minimum privilege required to enter the callgate. Therefore a callgate is a *controlled control transfer* that facilitates privilege escalation. We provide a pseudocode that describes the set of operations performed at the callgate in Algorithm 1. Note that we denote a control flow transfer where a context executing in Ring n enters Ring m through a callgate using the following notation:

$$CG : R_n \rightarrow R_m, \text{ where } n \leq CG.RMPL \text{ and } m \leq n$$

Privilege de-escalation. A context can return to its original privilege mode with a long return instruction[2] after privilege escalation. A long return instruction restores the caller's context that has been saved by the long call instruction as shown in Algorithm 2. It should be noted that a long return instruction only checks if the destination privilege level is numerically equal to or greater (lower privilege) by referencing the saved caller context. In fact, a long return instruction has no way of knowing if the saved context on the stack is indeed saved by the callgate. Hence, the long return instruction and similar return instructions such as `iret` can be thought of as *privilege de-escalating control transfer* instructions that pop the contents that are presumably saved registers. In this sense, a long return and its variants provide a *non-controlled control transfer* mechanism that is used to de-escalate privileges. We denote

1. "`lcall`" in AT&T syntax and "`call far`" in Intel syntax
2. "`lret`" in AT&T syntax and "`retf`" in Intel syntax

this specific type of control transfer where privilege is de-escalated (or stays the same) from m to n as the following:

$$R_m \rightarrow R_n, \text{ where } m \leq n$$

Inter-bitness control transfer. Inter-bitness control transfer is another type of an x86 control transfer that needs to be explained before we introduce our design. The x86-64 architecture provides *32-bit compatibility mode* within the x86-64 (AMD's amd64 or Intel's IA-32e architecture). As with the privilege level, the *bitness* is also defined by the currently active code segment descriptor. When a context is executing in a code segment whose descriptor has the `L` flag set, the processor operates in the 64-bit instruction architecture (e.g., registers are 64-bit, and 64-bit instructions are used). Otherwise, the context executes as if the processor is an x86-32 architecture processor. The bitness switching, although it changes the processor (current CPU core) execution mode, is no different than any other inter-segment control transfers with one exception: a callgate cannot target a 32-bit code segment. This is a perk that came with the introduction of the x86-64 implementation. In summary, we denote 32-bit code segments with a *x32* suffix as the following:

$$R_{n\_x32} \rightarrow R_m$$

3 Attack Model and Security Guarantees

3.1 Attack Model

We assume that the adversary is either an outside entity or a non-administrator user (i.e., no access to root account) who seeks to extract sensitive application code or data. The adversary may have an exploitable vulnerability in the victimized application that could lead to arbitrary code execution and direct access to the application secret. We assume such vulnerabilities are present when the app has been fully initialized and is servicing its user. However, we presume that the program is safe from the adversary during the initialization phase of the application. We also assume a non-compromised kernel that can support the LOTRx86 architecture. Our design requires the presence of a kernel module that depends on kernel capabilities, such as marking memory regions supervisor or installing custom segment descriptors. Also, our design includes Enter/Exit gates that facilitate the control transfer between the `PrivUser` and normal user mode. The gates amount to about 50 lines of assembly code, and we assume they are verifiable and absent of vulnerabilities. However, we do not consider microarchitectural attacks such as the variants of Meltdown [16] and Spectre [17]. It is an inherent limitation of all intra-process isolation schemes, as discussed in previous works [18], [19].

3.2 Security Guarantees

Our work focuses on providing developers with an underlying architecture, a new user privilege layer, which can be leveraged to protect application secrets and also program routines that access secrets securely. Using our architecture, we guarantee that a context in normal user mode cannot directly access a region protected (as a part of the `PrivUser` memory space) even in the presence of vulnerabilities. The adversary cannot jump into an arbitrary location in the `PrivUser` memory space to leak secrets since LOTRx86 leverages the x86 privilege structures to allow only controlled invocation of routines that handle sensitive information.

On the other hand, we do not focus on the security of the code that executes in our `PrivUser` mode. We also argue that the protection of application secrets in the presence of a vulnerability in the trusted code base (`PrivUser` code in our case) is an unrealistic security objective for any privilege separation scheme or even hardware-based Trusted Execution Environments [10], [20]. For instance, a recent work [21] proved that vulnerabilities inside SGX could be used to disclose protected application secrets. However, we *do guarantee* that the `PrivUser` layer is architecturally confined to its privilege that it cannot modify kernel memory nor infringe upon the kernel (Ring0) privileges even in the presence of a vulnerability in the `PrivUser` code. As we will explain in the coming section (section 4), this is a pivotal part of our architecture design. The privilege structures and gates that exactly achieve this security guarantee are one of the key contributions of this paper.

4 LOTRx86 Design

`LOTRx86` design harnesses the underused intermediate Ring levels of x86 to establish a new privilege layer called `PrivUser`, that is more privileged than the user (Ring3) and less privileged than the kernel(Ring0). `PrivUser` can safeguard sensitive program secrets and also host isolated code that interacts with the secret. In `LOTRx86`, access to the protected memory regions is granted based on the privilege and thus eliminating the necessity for page table switching or manipulation, and access to the protected memory regions is granted based on the privilege.

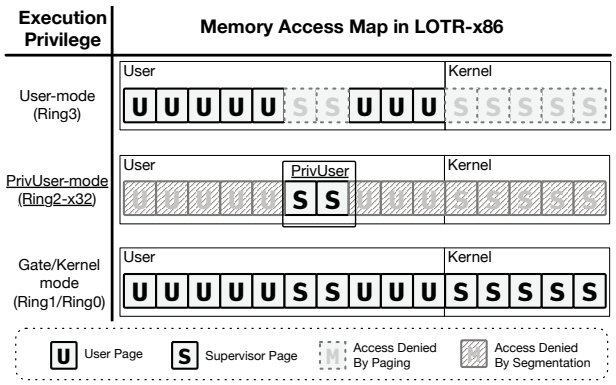
4.1 Privcall and Design Objectives

`LOTRx86` exposes the *Privileged User Call*, or `privcalls`, interface to developers. The developers can port their programs to use `LOTRx86` for sensitive code and data isolation. Below is the `privcall` interface that we provide to developers:

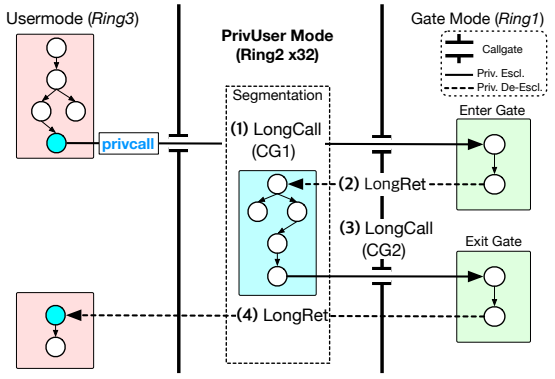
```
privcall(PRIVCALL_NR,...);
```

The `privcall` interface and its ABI are modeled after the Linux kernel's system call interface. The routine in `PrivUser` is identified with a number (e.g., `PRIV_USEPKEY=3`). For developers with experience in POSIX system programming, using the `privcalls` to perform the application's secret operations is intuitive.

Privileged-based memory access control. Our approach introduces a *privilege-based memory access control*, and it offers clear advantages over the existing process and thread-level approaches. The cost of the remote procedure calls for bridging two independent processes, or the cost of page table manipulation is eliminated. In our architecture, the memory access permissions do not change when the application secret needs to be accessed. Instead, the



(a) LOTRx86 process memory access map: PrivUser memory regions are mapped *Supervisor* protected by paging when in User-Mode (Ring3). In PrivUser-Mode (Ring2), in-place memory segmentation protects kernel and (optionally) normal user-mode memory.



(b) LOTRx86 gate design: implements inescapable segmentation enforcement through meticulously designed privilege and gate structures. `LOTRx86` uses Ring1 as Gate-mode in and out of the PrivUser-mode that executes in Ring2-x32.

Fig. 1: Memory access map and gate design of LOTRx86

privilege of the execution mode is elevated to obtain access to the protected memory.

Secure invocation. `privcall` is a single control transfer instruction (`lcall`), by which a context enters `PrivUser` mode through the `LOTRx86` Enter gate and returns upon finishing the `privcall` routine. Due to this design, the adversary cannot jump into an arbitrary location with the `PrivUser` privilege. Therefore, our architecture does not experience the security complications inherent to *enable and disable* models [9].

Portability. `LOTRx86` does not rely on new processor features for memory protection [10], [12], [13], [22]. Instead, we repurpose the underused privilege layers to implement `PrivUser`. Hence, our architecture is compatible across all generations of x86-64 processors. As we will present in §7, we evaluated our architecture and a PoC on both Intel and AMD's x86-64 processors.

4.2 Establishing PrivUser Memory Space

We face formidable challenges in the process of establishing the `PrivUser` layer. Our design creates a distinct execution mode (`PrivUser` execution mode) and its address space (`PrivUser` address space) for `PrivUser` layer. However, the resulting `PrivUser` layer must be intermediate, meaning that its address space should not be accessible by a user mode context, and at the same time, `PrivUser` execution mode must not be able to access the kernel address space. However, the x86 paging architecture provides only two memory privilege distinctions: `U`-pages and `S`-pages. The memory segmentation feature that existed in x86-32 is deprecated in x86-64, eliminating an additional memory access control mechanism to paging.

In summary, `PrivUser` layer must satisfy the two fundamental memory access security requirements (**M-SR1** and **M-SR2**) to function as an intermediate layer.

M-SR1. User mode must not be able to access `PrivUser` memory space

M-SR2. `PrivUser` mode must not be able to access kernel memory space

Satisfying M-SR1. We satisfy **M-SR1** by mapping all pages that belong to `PrivUser` as `S`-pages to prevent a user mode context from accessing `PrivUser` code and data. As a result, `PrivUser` memory space that is mapped as `S`-page is accessible to `PrivUser` mode, but not to user mode. Now, we see that we are already using both of the two privilege distinctions recognized by the paging system, and we are unable to protect the kernel from `PrivUser` mode.

Satisfying M-SR2. We enforce `PrivUser` mode to be a *segmentation-enforced execution mode* to prevent `PrivUser` from accessing kernel memory pages, by defining `PrivUser` execution mode as a 32-bit segmentation-enabled code segment as shown in Table 2. This way, entering `PrivUser` mode changes not only the currently active code segment but also the bitness of the execution mode. That is, when user mode enters `PrivUser` mode through `privcall` the execution mode is set to the 32bit compatibility mode. As a result, we can enforce segmentation to set boundaries for the powerful `PrivUser` mode (Ring2) that is capable of accessing `S`-pages. The resulting memory access map of the three execution modes is illustrated in Figure 1a. With our design, the `PrivUser` memory space serves as a *functionally* intermediate memory space for `PrivUser`.

However, adapting segmentation enforcement that satisfies **M-SR2** introduces new challenges that must be overcome by a meticulously designed `PrivUser` gate design. For one, the current x86-64 hardware does not allow a callgate to perform inter-privilege and inter-bitness control transfer. This instantly presents a challenge for `LOTRx86` that wishes to let `PrivUser` operate in the 32-bit Ring2 mode. Moreover, a naively designed control transfer structure may allow `PrivUser` to escape the enforced segmentation arbitrarily. We explain `LOTRx86`'s `PrivUser` gate design that overcomes the hardware constraint (§4.3) and achieves inescapable segmentation enforcement (§4.4). Then, we provide an overview of the final gate design (§4.5)

for completing **M-SR2** in the remainder of this section.

4.3 Hardware constraint and Gate mode

Hardware constraint. There is an x86-64 specific perk that has been proved to be a constraint in our design. The x86-64 mode (both 64-bit mode and the 32-bit compatibility mode) only supports a 64-bit mode callgate which is an extended version of its counterpart that existed in x86-32. Specifically, it does not allow the target code segment of a callgate to be a 32-bit segment. This implies that an inter-bitness control transfer through callgate is not supported both ways; while $CG : R_{n\_x32} \rightarrow R_m$ is possible, $CG : R_m \rightarrow R_{n\_x32}$ is an invalid callgate definition. We define this constraint **C** in a more formal manner as the following:

C. $CG : R_n \nrightarrow R_{m\_x32}$: callgate cannot target a 32-bit code segment

Overcoming hardware constraint with gate mode. We overcome the hardware constraint **C** through the use of a `Gate` mode segment that facilitates elevation of privilege and switching the execution mode to the 32-bit compatibility mode. Due to **C**, a privilege escalation and bitness switching cannot be simultaneously achieved in a single callgate transfer. However, we observe that the two individual operations are still valid in x86-64. Our design, therefore, adapts a 64-bit Ring1 gate mode to enable user-to-`PrivUser` control transfer. The gate mode, then performs a subsequent `lret` control transfer into the 32-bit Ring2 `PrivUser` mode. With the necessity for a `Gate` mode explained, we proceed to explain the properties of `LOTRx86` control transfer design (**P1** and **P2**) that provide the control transfer security requirement for `PrivUser` (**CT-SR**).

4.4 Inescapable Segmentation Enforcement

`LOTRx86`'s control transfer structure must ensure that no *non-controlled* (i.e., not through callgates) inter-segment control transfer paths *out* of the `PrivUser` mode arrives in a segment that is *1.* has a Ring privilege numerically less than 3 (can access `S`-pages), *2.* and is a 64-bit segment (no segmentation is enforced). This requirement must be strictly maintained for the memory boundary between `PrivUser` and kernel (**M-SR2**) to hold. We denote this control transfer security requirement for the enforcement of inescapable segmentation as **CT-SR**:

CT-SR. $R_{2\_x32} \nrightarrow R_{e\_x64}$, where $e < 3$: there must be no possible non-controlled control transfer from `PrivUser` mode (R_{2_x32}) to a 64-bit Ring privilege e (escape) that is capable of accessing `S`-page access privilege

Preventing non-controlled control transfer routes. An adversary may break **CT-SR** through non-controlled control transfer if the gate structure is not carefully formulated. As we explained in §2, a non-controlled inter-segment control transfer can be made to jump to a less privileged code segment without any security checks. Therefore we must rigorously verify all possible non-controlled transfers from R_{p_x32} to all Ring levels e that is $e \geq 2$ (Ring privilege levels that are numerically equal or greater, meaning equal or lower privilege).

First, we must make sure that a context in `PrivUser` mode cannot arbitrarily jump into an arbitrary place in the `Gate` mode. Due to constraint **C**, we must make use of two intermediate Ring levels; one for the `Gate` mode and another for the `PrivUser` mode. In order to prevent a non-controlled control transfer $R_p \rightarrow R_g$, we realize that the gate mode privilege must be higher (numerically lower in terms of Ring number). Hence the following property must hold in our design:

P1. $g < p$ (R_g is higher in privilege than R_p) : privilege of `Gate` mode must be higher than that of `PrivUser` mode

The second possible escape route is to perform a same-privilege inter-bitness (32bit $\rightarrow$ 64bit) inter-segment control flow. We prevent such a route by intentionally not defining a 64-bit code segment for the Ring level 2. A Ring privilege level in the x86 architecture comes into existence when it is defined in the descriptor table, and a context loads the segment selector that points to the code segment through inter-segment control flow instruction. Hence, a Ring level that is not defined in the descriptor tables, *does not* exist within the system. Hence, by only defining a 32-bit code segment for Ring2, Ring2 becomes a 32-bit only, segmentation enforced Ring level in our system definition. We denote this property of our privilege structure design as shown below:

P2. $\nexists\ R_{p\_x64}$: 64-bit counterpart of `PrivUser` mode segment must not exist

4.5 Final gate design overview

In summary, our privilege definitions and gate structures (Table 2 and Figure 1b) is the only configuration that satisfies constraint **C** and also the properties **P1** and **P2**. Our design satisfies **CT-SR** by maintaining the required properties **P1** and **P2**. We chose Ring1 as the privilege level for `Gate` mode ($g < p$), while enforcing the segmentation on all `PrivUser` mode execution by defining only 32-bit segmentation-enforced code segment for the Ring level 2 ($\nexists\ R_{p\_x64}$). By meeting **CT-SR**, we complete our solution for the establishment of the `PrivUser` memory space that satisfies both **M-SR1** and **M-SR2**; the `PrivUser` memory space is protected from context running in the user mode, while the `PrivUser` mode is architecturally bound to its memory space that it cannot access kernel memory under all circumstances.

The resulting control transfer between the user and `PrivUser` are as the following: a `privcall` first enters `Gate` mode through the CG1 into the Enter Gate (Table 2). At the gate mode, we load the stack with the following arguments: {`PrivUser` entry point, `PrivUser` code segment selector, `PrivUser` stack address, `PrivUser` stack segment selector}, and then perform a far return `lret` to enter `PrivUser` mode. While this control transfer is made through a non-controlled control transfer instruction, the Enter gate consisting of about 30 lines of assembly instructions is guaranteed to be executed *from the beginning* by the CG1. In other words,

we chain a non-controlled control transfer with a controlled control transfer (CG1) to guarantee its correct execution.

5 IMPLEMENTATION

In this section, we explain the prototype of our LOTRx86 architecture in detail. Our prototype implementation consists of the following components:

lotr-kmod. We built a Linux kernel module that communicates with the host process (LOTRx86 enabled process). The module creates a virtual device interface at /dev/lotr, and an LOTRx86 enabled program communicates with our kernel module with the ioctl interface. The kernel module builds the PrivUser-space for the program when requested.

liblotr. The user library liblotr allows developers to use our architecture in the host program, isolate the application secrets, and implement privcalls that securely access the secrets. A developer can initialize the PrivUser-space and utilize the privcall interface through our user library. The library also includes tools and scripts for building the executable that runs in the PrivUser-space.

lotr-libc. We provide a modified version of the *musl* [23] libc for building the PrivUser executable. We modified the heap memory manager such that only S-pages are allocated to the heap managers used in the PrivUser mode. In this way, we prevent the leakage of the application secret and the by-products of its processing to the user space.

lotr-build. lotr-build is a dedicated toolchain that allows developers to compile the PrivUser portion of their application and incorporate it into the host application. We further explain this procedure later.

5.1 PrivUser mode Initialization

The lotr-kmod kernel module initializes the LOTRx86 infrastructure such as the Gate-mode, PrivUser mode, and control transfer structures for the host process. The host application is required to call init_lotr(&req) function from liblotr with an argument of the struct init_request type during its initialization. The request structure contains the addresses and sizes of PrivUser components that lotr-kmod needs in its initialization routine. Such information includes the range of PrivUser code segment, data segment, the entry point for the PrivUser-space, pages to be used as a stack in PrivUser, and so forth. The addresses of the segments are available through the symbols generated by our build tools during the compile-time, while the stack is allocated through mmap in liblotr. Additionally, lotr-kmod contains the Enter gate and Exit gate that are loaded into the kernel memory upon module load.

The lotr-kmod kernel module creates an LDT for the host process and writes the segment and callgate descriptors that are used for the Gate-mode and PrivUser mode. Unlike the GDT, an LDT is referenced on a *per-process* basis; an LDT can be created for each process, and the register that points to the currently active LDT called *ldtr* is updated in each context switch. For this reason, the LOTRx86 descriptors can only be referenced by the host process that explicitly requested the initialization of the LOTRx86 infrastructure. lotr-kmod creates the descriptor segments listed in Table 2. A set of Ring1 code and data segments are used for the Gate-mode, and Ring2-32bit segment descriptors are loaded as a context enters the PrivUser mode.

	Type	Priv.
Gate-mode CS	Code Segment	Ring1
Gate-mode DS	Data Segment	Ring1
PrivUser mode CS	Code Segment	Ring2-x32
PrivUser mode DS	Data Segment	Ring2-x32
CG1	CG1 (R3→R1)	CPL $\leq$ 3
CG2	CG2 (R2→R1)	CPL $\leq$ 2

TABLE 2: LOTRx86 LDT descriptors: by defining segment and callgate descriptors in LDT, LOTRx86 creates Gate-mode and PrivUser mode for a process

The initialization also sets the Gate-mode stack to be loaded at the Enter callgate. As briefly explained in §2, the x86 callgate mechanism finds the address of the new stack for the control transfer at the TSS structure. The TSS structure holds the addresses of each Ring level. In our case, we use two callgates, CG($R_3 \rightarrow R_1$) and CG($R_{2\_x32} \rightarrow R_1$), that both require a stack for Ring1. Hence, we allocate stack space and record the top of the stack in the Ring1 stack field of the TSS (TSS.SP1).

Another important task carried out during the initialization (in lotr-kmod) is marking the pages that belong to the PrivUser-space Supervisor pages. The kernel module walks the page tables and marks PrivUser pages Supervisor by clearing the User bit in the page table entry. All pages that are marked Supervisors are maintained in a linked list so they can be reverted or freed when necessary as the host process terminates.

When all necessary initialization procedures are finished, the kernel module creates a lock for the host process based on its PID. From this point on, lotr-kmod ignores additional initialization requests delivered via the ioctl requests from the host to thwart any possible attempt to compromise the PrivUser-space.

5.2 LOTRx86 ABI

The privcall interface of the LOTRx86 is almost identical to the syscall interface; privcall follows the x86-64 System V AMD64 ABI system call convention [24]. That is, we use %rax, %rdi, %rsi, %rdx, %r10, %r8, %r9 registers for passing arguments to the PrivUser mode, and the return value is stored in %rax. Underneath the surface, however, our unique design enables establishment and secure use of the PrivUser-space. From here on, we explain each stage of the control flow transfers in the ABI – starting from a privcall and its return to its caller.

privcall interface. A privcall (NR_PRIVCALL, ...) consists of layers of macros that handle a variable number of arguments and place them in the argument registers in order. After the arguments are placed according to the x86-64 syscall ABI, a long call (lcall) instruction is executed with a segment selector that points to the Enter callgate as an argument. Upon the executing of lcall, the execution continues at the Enter gate with a privilege of Ring1.

```
# Entered from privcall in user mode
LOTREnterGate:
# (a) Allow only Ring 3 to enter this gate
movq    8(%rsp), %r11
cmp     $3, %r11
jnz     EXIT
# (b) Save User mode(R3) Context
pushq   24(%rsp);
pushq   16(%rsp);
pushq   8(%rsp);
pushq   0(%rsp);
SAVE_REGS();
# (c) Transfer Arguments into PrivUser Stack
movq    $PrivUserStack, %r11;
subq    $60, %r11;
movl    $DummyEIP, 0(%r11d);
movq    %rax, 4(%r11d);
movq    %rdi, 12(%r11d);
movq    %rsi, 20(%r11d);
movq    %rdx, 28(%r11d);
movq    %r10, 36(%r11d);
movq    %r8, 44(%r11d);
movq    %r9, 52(%r11d);
# (d) Push PrivUser(RIP,CS,RSP,SS) onto stack,
# then perform control flow transfer
movq    $PrivUserEnter, %r9;
pushq   $PrivUserSS;
pushq   %r11;
pushq   $PrivUserCS;
pushq   %r9;
lret;

# Entered from privret in PrivUser mode
LOTRExitGate:
sub     $GateContextSize, %rsp
RESTORE_REGS();
# in case security check (a) fails
EXIT:
lret;
```

Fig. 2: Simplified pseudo assembly code of LOTRx86 Enter gates

Enter gate. The LOTRx86 Enter gate plays a pivotal role in safeguarding the user mode context that invoked a privcall into the PrivUser mode. Figure 2 is a simplified pseudo assembly code of the implementation. The Enter gate is written in assembly code and is about 30 instructions that carry out three main operations.

First, the Enter gate checks the saved %cs in the gate stack. At this point, the ring privilege has been escalated to that of the Gate mode (Ring1), stack pointer now points to Gate mode stack, and the caller context is saved in the new stack. (for detailed x86 callgate operation, revisit Algorithm 1 in §2). The least significant 2 bits of the saved %cs (%cs[1:0]) indicate the caller's Ring privilege. By ensuring the value to be 3, we prevent PrivUser mode from entering the Enter gate for possibly malicious intent.

Then the gate saves the user mode caller context in the Gate mode stack. Note that the x86 long call instruction has context-saving feature built in. However, since we use the Ring1 for both Enter gate and Exit gate, the saved context is overwritten when the context returns from PrivUser mode back to the Exit gate. Therefore, we found that it is necessary to perform a manual context saving of the four registers (%RIP, %CS, %RSP, %SS) in the beginning of our Enter gate as shown in the code block (b) in Figure 2.

The second operation (code block (c) in Figure 2) illustrates the transforming of the privcall arguments that follow the x86-64 calling convention into that of the PrivUser mode ABI; the in-register arguments must be transferred to the PrivUser mode stack as preparation before entering the PrivUser mode. Unlike the conventional x86-32 ABI, we use the 64-bit arguments in the PrivUser mode by default. The fact that the PrivUser mode runs in 32-bit compatibility mode but uses 64-bit length arguments is a peculiar characteristic of our design, and the LOTRx86 Enter gate resolves the calling convention discrepancy.

The last operation (code block (d)) performed in the Enter gate is to transfer the control flow into the entry point of PrivUser mode. We push the entry point address, the address of the PrivUser mode stack that contains the arguments passed on by the privcall in the user mode at this point, and their segments (%cs and %ss) on to the current (Gate mode stack). Then, we execute the lret instruction to enter the PrivUser mode.

PrivUser entry point. The PrivUser mode entry point first performs a bound check on the %eax that contains the privcall number (i.e., $1 \leq$ nr_Privcall $\leq MAX\_PRIVCALL$). The pointers to the predefined privcall routines are arranged in the *Privuser Call Table (PCT)* whose role is identical to the *system call table* in the Linux kernel. This mechanism prevents a maliciously crafted privcall from calling an arbitrary memory address. If the check is valid, then the entry point calls the *wrapper function* for the privcall routine that corresponds to the number is invoked.

PrivUser routine. The developers can define a privcall routine through the PRIVCALL_DEFINE(func_name,...) macro. The macro creates and exports a wrapper function that calls the main function. This particular implementation is borrowed from the Linux kernel [25]. The wrapper casts the 64-bit arguments into function-specific argument sizes (e.g., 64-bit to int (32bit)) for the defined privcall routine. After the privcall routine is finished, the execution returns to the PrivUser entry point to be concluded by lcall that transfers the control flow *back* into the Exit gate with the privilege of the Gate mode (Ring1).

Exit gate. The exit gate scrubs the scratch registers (the six general-purpose registers as stated in the System V i386 calling convention) to prevent information leakage from the PrivUser mode. Recall that we manually saved the user mode context in the stack from the Enter gate. We subtract the stack pointer ($48(8 \times 6)$ bytes in our implementation) to move it to the saved context. We execute popq instruction to restore %rbp then the lret instruction to restore %RIP, %CS, %RSP, %SS to return to the original caller of the privcall with a privilege of user mode (Ring3).

5.3 Developing LOTRx86-enabled program

We developed tools and libraries that allow developers to write LOTRx86-enabled programs. Writing a privcall routine is similar to writing a regular user-level code. However, there are key differences both in the developer's perspective and underneath the surface. Here, we outline the important aspects of LOTRx86-enabled program development.

The privcall interface and the development of privcall routines are intentionally modeled after the Linux

kernel's system call interface. For this reason, the procedures for developing the PrivUser side of the program and invoking them as necessary are nearly identical to those of developing new system calls to the kernel.

privcall declaration. liblotr provides two important macros through <lotr/privuser.h>. First is the declaration macro #PRIVCALL_DEFINE. The macro takes the name of the function as the first argument and up to six arguments. The type and the name of the arguments must be entered as if they are separate arguments (e.g., (int, mynumber)). This is because PRIVCALL_DEFINE generates a wrapper function that casts the ABI-defined arguments into the argument's type. We restrain from further explaining the details of the macro since it is almost identical to the kernel's SYSCALL_DEFINE macro.

Compiling with lotr-libc. We provide gcc-lotr which is a wrapper to the gcc compiler. gcc-lotr links the user's PrivUser code with lotr-libc instead of the default glibc (32bit). lotr-libc is a modified version of musl-libc. We modified the malloc function in the musl-libc so that it manages a memory block from the PrivUser memory space S-pages. This is to prevent the by-products or the application itself from being placed in a memory region accessible to the normal user mode. Additionally, we implemented a function that initializes process *Thread Local Storage (TLS)* that can be called from liblotr's init_lotr() function. The initialization of a process TLS is performed by the libc library before the program's main() is executed. Therefore, it is necessary to implement a separate function to initialize the TLS for LOTRx86.

Building final exectuable. Compiling the PrivUser code with our build tools yields two files: a header file in which privcall numbers are defined, and a LOTRx86 object in .lotr extension. The header file lists the assigned number for each declared privcall routines, and the .lotr file is an object file ready to be linked to the main program. Our build tool compiles the PrivUser code in x86-32 code. However, we copy the sections of the 32-bit object into a new 64-bit ELF object format so that it can be linked to the main program. The PrivUser build tools also strip all the symbols to prevent symbol collision between the 32-bit libc (lotr-libc) and the 64-bit libc used in the main program, then it generates a symbol table that includes addresses of the PrivUser object sections and most importantly, the PrivUser entry point. The main program is built with our linker script (LOTR.linkerscript) that loads the symbol table generated during PrivUser build. When the main program launches, init_lotr() fetches the symbols and transfers them to lotr-kmod, and the kernel module marks the memory pages that belong to the PrivUser memory space S-pages.

5.4 Kernel changes

The LOTRx86 prototype is implemented as a kernel module. However, we also made minor but necessary modifications to the Linux kernel. First of all, we made sure that system calls (e.g., mprotect) that alter the memory permissions of the user memory space ignore the request when the affected region includes PrivUser-memory. This is achieved by simply placing a "*if-then-return -ERR*" statement for the case where the address belongs to the user-space but the page is an S-page. We made a similar change to the munlock system call so that PrivUser's P-pages are excluded from possible memory swap-outs.

6 Supporting 64-bit Execution Isolation with PrivUser64

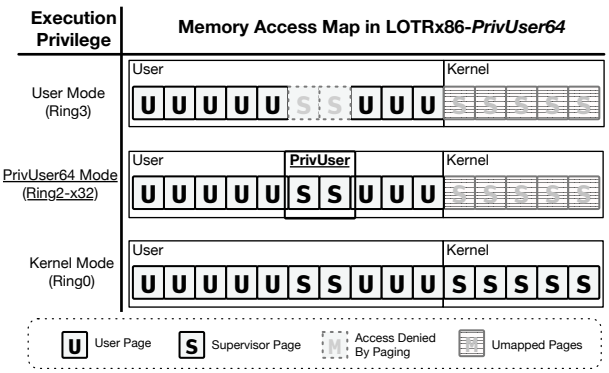
In this section, we describe an extension to the LOTRx86 architecture that allows 64-bit PrivUser execution. The alternative privilege separation design allows the PrivUser mode to operate in Ring2 64-bit execution mode, namely the PrivUser64 mode. We employ kernel page unmapping such that the kernel pages are unmapped during user and PrivUser mode execution. This eliminates the necessity for the segmentation feature enabled through making the PrivUser code segment to 32-bit. We add PrivUser64 support through extending the components of LOTRx86 to provide options to choose either PrivUser or PrivUser64. Both execution modes cannot be enabled simultaneously. However, liblotr allows selecting either PrivUser or PrivUser64 during initialization through an argument and also provides a new privcalls interface. The introduction of PrivUser64 is achieved through modifications mostly in the kernel module and the LOTRx86 toolchain; the user space changes are minimal, and PrivUser64 is designed to be compatible with the existing LOTRx86 ABI. This means that the previous applications written with the LOTRx86 API can be recompiled to use PrivUser64 with minor changes.

6.1 Establishing PrivUser64 Memory Space

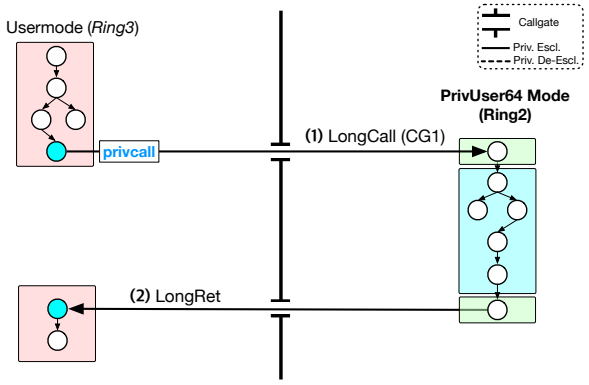
LOTRx86 with PrivUser64 achieves 64-bit execution mode isolation through replacing the segmentation with kernel memory space unmapping. Segmentation in LOTRx86 default design serves to create a boundary between PrivUser and kernel. We observe that we can let PrivUser operate in 64-bit intermediate privilege Ring by adapting a hybrid approach that utilizes page unmapping. To achieve this, we leverage *Kernel Page Table Isolation (KPTI)* [26], [27].

KPTI, introduced as mitigation for Meltdown [16] has brought permanent changes in the memory isolation model in many x86 systems; KPTI is obligatory for a large number of machines with Intel-based x86 processors. KPTI-enabled systems no longer map user and kernel memory pages simultaneously mapped. That is, when a context is in user mode, the kernel pages except a few that are essential for facilitating kernel entrance are unmapped. This, quite ironically, satisfies the design requirement **M-SR2** of the LOTRx86 architecture explained in §4.2. Consequently, LOTRx86 in x86 systems that are affected by Meltdown no longer require the 32-bit mode segmentation.

By leveraging KPTI, we opportunistically explore the 64-bit mode enabling design for LOTRx86. The memory space separation between user and PrivUser32 is still achieved through marking PrivUser32 pages as S-pages, and utilizing the inter-privilege callgates, while PrivUser32 and kernel memory space isolation is achieved through unmapping kernel space while in user and PrivUser. For systems with KPTI enabled (e.g., most Intel-based x86 systems), the 64-bit mode LOTRx86 can be readily deployed.



(a) LOTRx86 PrivUser64 process memory access map: PrivUser memory regions are mapped as *Supervisor* protected by paging when in User-Mode (Ring3). Kernel pages are unmapped during User and PrivUser64 mode execution



(b) LOTRx86 PrivUser64 gate design: PrivUser64 uses Ring2-64 as its execution mode, and gate structure is simplified compared to that of LOTRx86.

Fig. 3: Memory access map and gate design of LOTRx86 PrivUser64

For systems that do not require KPTI, incorporating a *per-process* KPTI-like kernel memory isolation for LOTRx86-enabled processes can be considered. While we regard such effort future work, we evaluate the PrivUser64 design and show that it outperforms the 32-bit PrivUser32 through microbenchmarks and LOTRx86-enabled webserver in §7. In all, we outline satisfying the memory security requirements (**M-SR** for PrivUser64) as the following:

Satisfying M-SR1. LOTRx86's PrivUser64 extension prevents user-bound PrivUser memory space access using the same method as in LOTRx86. Since PrivUser pages are marked as s-pages, a context running in user mode cannot access PrivUser memory.

Satisfying M-SR2. PrivUser64 extension adapts the paging-based memory isolation technique to isolate the kernel memory space from PrivUser. More specifically, the kernel pages are unmapped while the context is executing in user or PrivUser mode. Such feature has already been incorporated in the Linux kernel since the rise of meltdown [16] We leverage the KPTI in the Linux kernel. We reuse the KPTI to evaluate the PrivUser64 isolation model. Figure 3a shows the memory access map for the user, PrivUser64 and the kernel. The PrivUser memory space is protected from user mode through the use of s-pages as in LOTRx86. Unlike LOTRx86, however, the PrivUser-kernel boundary is defined by unmapping the kernel pages while executing in user(Ring3) and PrivUser (Ring2) execution.

6.2 Improved Binary Compatibility

Besides the general performance and memory capacity advantages of 64-bit mode execution over 32-bit mode execution, the 64-bit support also attenuates the binary compatibility issues that may arise when porting programs to use LOTRx86. A 32-bit PrivUser execution mode creates a binary compatibility problem between the user mode and PrivUser mode. The necessity of data structure serialization during argument passing is one such example. A developer must define *binding* code and data structures that are dedicated to argument passing or serialize the 64-bit data structure into its 32-bit counterpart. These approaches are feasible when the isolated compartment in PrivUser is small and exposes only a handful number of functions via privcalls (e.g., our Nginx PoC presented in §5). However, the procedure may be cumbersome when isolating a large component into PrivUser, although it is not advisable from a security perspective. A 64-bit PrivUser will resolve the binary compatibility issue and simplify the porting process for LOTRx86.

6.3 Supporting Systems Calls inside PrivUser64

The adaptation of a 64-bit Ring2 PrivUser execution mode introduces a complication; we found that the x86-64 syscall and sysret instructions, that lets the context enter and exit kernel mode for system calls, assume that all incoming system calls are from Ring3. This is a behavior that was not present in PrivUser32 that used the int 0x80 instruction for kernel entrance. More specifically, the syscall instruction overwrites the caller cs segment register content with that of the kernel without saving that of the caller. As such, the sysret instruction simply overwrites the cs segment register with that of the Ring3 user mode without respecting the caller's cs segment. This poses a challenge to the design of LOTRx86, since a PrivUser context will be in Ring3 execution privilege when returning from a system call.

Authenticating PrivUser contexts in system call entry. In order to mitigate the issue, we make a slight modification to the system call entry and exit points. That is, we must alter the system call handling behavior such that it respects the privilege level of the user context. While the original content of the cs register is lost upon entering kernel, we

	Type	Priv.
PrivUser64 mode CS	Code Segment	Ring2-*x64*
PrivUser64 mode DS	Data Segment	Ring2-*x64*
CG	CG (R3→R2)	CPL $\leq$ 3

TABLE 3: LDT descriptors of LOTRx86 with PrivUser64.

observe that the ds register is intact. This allows us to differentiate the PrivUser contexts from user mode contexts at the system call entry point. We use the ds segment register as a token for the possession of the Ring2 execution mode privilege. Loading the Ring2 data segment into the ds segment register requires the context to be executing in a privilege higher than Ring2. Hence, we can verify that the context was executing in Ring2 when syscall instruction was invoked if ds contains a selector to a data segment whose privilege is Ring2. Based on this observation, we let contexts load the the Ring2 ds segments as they enter PrivUser mode in the entry point. Then, we check if the context's ds segment register points to a Ring2 ds segment at the system call entry point to authenticate the context's possession of the Ring2 privilege. That is, the DPL of the %ds segment register is used as a token for authenticating that the context has invoked the system call during the PrivUser (Ring 2) execution.

Returning contexts to PrivUser64. To enable returning from a completed system call back to PrivUser mode, we use the iret instruction that respects the caller's %cs register. We slightly modify the system call exit routine so that the contexts identified as PrivUser contexts in the system call entry point returns to the PrivUser mode with their original Ring2 privilege.

6.4 Summary of Implementation Changes

We explain the summary of changes to extend LOTRx86 implementation to support PrivUser64. The PrivUser64 mode support is added in a non-intrusive manner such that the LOTRx86 ABI is respected and thus allowing the existing LOTRx86-enabled applications can be easily recompiled to use PrivUser64 (e.g, our LOTRx86-enabled webserver to be presented in §5). liblotr now provides two header files and shared libraries for PrivUser32 and PrivUser64 (e.g., #include "lotr/privcall.h" vs. #include "lotr/privcall64.h"). We add PrivUser64 compilation mode to lotr-build in which the process of building a 32-bit executable is omitted, and the PrivUser64 sections are directly embedded into the final executable to be loaded into PrivUser64 memory space during runtime. Regarding PrivUser entry point, the gate mode (Ring1) is no longer necessary and hence a user mode context enter PrivUser64 directly via a lcall instruction that targets a Ring3 to Ring2 callgate ($CG : R_3 \rightarrow R_2$). The entry point to PrivUser64 is positioned at the target of the callgate is shown below:

```
1  SAVE_ALL
2  movq    $call_table, %rbx
3  shl     $3, %rax
4  addq    %rax, %rbx
5
6  # Load Ring2 DS for auth at syscall entry
7  mov     $0xe, %r15
8  mov     %r15, %ds
9
10 # Invoke corresponding privcall
11 call    *0(%rbx)
12
13 # Clear DS before return to user
14 xor     %r15, %r15
15 mov     %r15, %ds
16 RESTORE_ALL
17 rex64 lret
```

In addition to simplifying the complex privilege structure, the PrivUser64 entry point is concise as there is no need to bridge the gap between the 32-bit and 64-bit ABI differences. For instance, there is no need to transfer in-register privcall arguments from user mode onto the 32-bit PrivUser's stack. The mov instruction that manipulates the %ds instructions allow the system call entry point to authenticate system calls invoked from PrivUser64 as we explained.

We evaluate LOTRx86 through a set of experiments in the next section. For all experiments, we compile the isolated program component to both 32-bit PrivUser and PrivUser64 to show the performance characteristics of the two PrivUser execution modes.

7 Evaluation

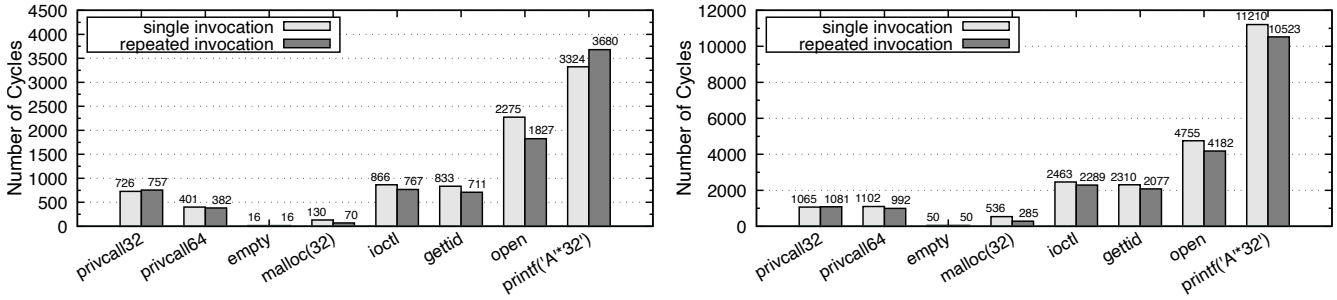
To show the feasibility and efficiency of the LOTRx86 architecture approach, we performed a set of microbenchmarks (§7.1), a comparison against traditional memory protection methods using an example program, and a real-world application evaluation. For the real-world application evaluation, we developed a proof-of-concept (PoC) by incorporating our architecture into the *Nginx* webserver [28] as well as the *LibreSSL* [29] that is used by the web browser to support SSL. We modified the parts of the webserver to protect the in-memory private key in the PrivUser memory space and only allow access to the key through our privcall interface.

We evaluate LOTRx86 implementation, including the 64-bit extension. To avoid confusion, we use the term PrivUser in a more general sense, while we use PrivUser32 and PrivUser64 to refer to each PrivUser execution modes that LOTRx86 support, respectively. Also, we call privcalls to the two PrivUser execution modes as privcall32 and privcall64. For all programs used in the evaluations, we compile two versions such that one runs on PrivUser32 while the other runs in PrivUser64. For LOTRx86 with PrivUser64 mode, we enabled the KPTI feature for both Intel and AMD machines. Hence, it should be taken into account that in the evaluations in which PrivUser64's performance is measured, KPTI induced a system-wide overhead.

We present the results from a set of microbenchmarks that we performed to measure the latency induced by the privcalls. Then, we compare the performance of the PoC webserver whose private key is protected with its original version. Our experiments are conducted on both Intel and AMD to show that our approach is portable. The evaluations were conducted with an Intel machine with Core i7-10700 (8 cores, 2.89Hz) and 32GB of RAM and an AMD machine with Ryzen Threadripper 3990x (64 cores, 2.9GHz) and 256GB of RAM. Both machines run Ubuntu 20.04.2 with Linux kernel 4.14.72. KPTI is enabled for both PrivUser32 and PrivUser64 isolation on Intel and AMD.

7.1 Microbenchmarks

The privcall allows developers to invoke routines that access application secrets in the PrivUser layer. A certain



(a) privcalls vs. common C library calls (Intel) (b) privcalls vs. common C library calls (AMD)

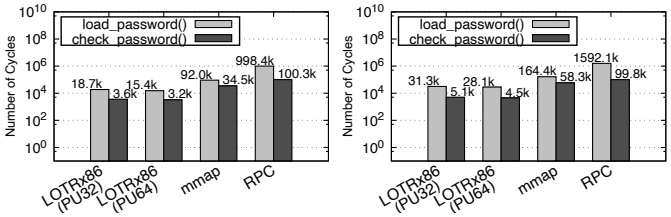
Fig. 4: Microbenchmarks of `privcall` and `privcall64` on Intel - (a) and AMD (b)

amount of added latency is inevitable since we perform a chain of control transfers to securely invoke the `privcall` routines. We also conducted the microbenchmark in two varying setups. In the first setup, we built executables that make a single invocation of each call (`privcalls`, library calls), and we produced the results by executing the executables 1000 times. In the second setup, we measure the latency of 1000 consecutive invocations of each call in a loop. These two setups represent the two situations where `privcall` is infrequently called and frequently called. The latencies of the `privcalls` are measured in the number of cycles consumed, alongside the latencies of common C library calls. The microbenchmarks provide a general context into the latency of `LOTRx86`'s privilege switches, and we further evaluate its overhead in private key protection in webservers in §7.3

Single invocation. Figure 4 shows the microbenchmark results for `privcall32` and `privcall64` on Intel and AMD machines. For both types of `privcalls`, the latency of the `privcalls` are on par or faster with simple libc functions such as `ioctl()`.

Repeated invocation. It is noticeable that the latency of both types of `privcalls` does not improve drastically. On the Intel machine, both `privcall32` and `privcall64` do not improve when repeatedly called whereas some of the other calls, such as `gettid()` dropped from 549 to 387. As to this result, we surmise that the control flow transfer chain used in our architecture affects the caching behavior of the processor negatively. Also, the libc and kernel's system call invocation have been extremely well optimized for a long period of time. Hence, we plan to investigate possible optimizations that can be applied to `LOTRx86` in the future.

`privcall32` vs. `privcall64` The particular microbenchmark does not involve any system calls, allowing us to compare the two types of `privcalls` directly. `privcall64` shows noticeable latency improvement over `privcall32` on the Intel machine (757 vs. 382 in repeated invocation), while the two show similar cycles on the AMD machine (1081 vs. 992). However, the invocation latency alone cannot serve as an effective measure for their performance in general applications. We further discuss the real-world performance of `PrivUser` and `PrivUser64` with our `LOTRx86`-enabled webserver example in §7.3.



(a) `LOTRx86` on Intel (b) `LOTRx86` on AMD

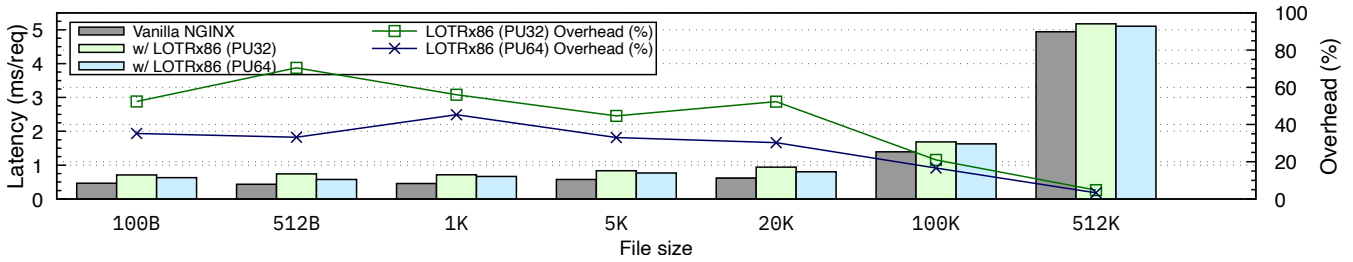
Fig. 5: Execution time comparison of test program using `LOTRx86` `PrivUser32` and `PrivUser64` vs. traditional memory protection mechanisms. Y-axis is in log scale.

7.2 Comparison with Traditional Memory Protection Techniques

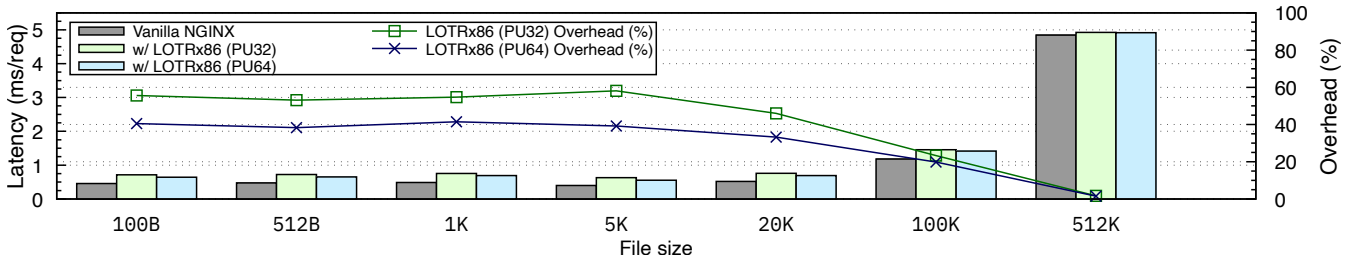
We implemented a simple demonstration in-process memory protection using `LOTRx86`, page table manipulation technique implemented with `mmap` and `mprotect`, and a socket-based remote procedure call mechniasm (from `<rpc/rpc.h>`).

Test program. Our simple program first loads a password from a file into the protected memory region, then receives an input from the user via stdin to compare it against the protected password. In more detail, we implemented two functions `load_password` and `check_password` using the three protection mechanisms to evaluate their performance overhead. For page-table-based method, we use `mprotect` to set the page that contains the `load_password` and `check_password` and the page dedicated for storing the loaded password to `PROT_NONE`. In the case of the RPC mechanism, we place the two measured functions and the password-storing buffer in a different process and make RPCs execute the functions remotely.

Performance overhead comparison. The measurements for the execution time of the two functions, implemented with three different mechanisms, are illustrated in Figure 5. We averaged the results from 1000 trials (the y-axis is in log scale). The results show that `LOTRx86` with `PrivUser32` proves to be much faster than the two traditional methods by a large margin. On Intel PC (Figure 5a, `LOTRx86` greatly reduces the number of cycles consumed of `load_password` by 79.3% (18,728



(a) Nginx latency measurements on Intel



(b) Nginx latency measurements on AMD

Fig. 6: SSL KeepAlive response latency measured with with varying file sizes on Nginx with `LOTRx86` (`PrivUser` and `PrivUser64`)

vs. 91952.9 cycles) and by ~99.99% (18,728 vs. 998,352.3 cycles), compared to `mmap` and RPC-based implementation, respectively. `LOTRx86` with `PrivUser64` mode enabled shows similar levels of improvements over the traditional techniques; `LOTRx86` with `PrivUser64` marked 15382.3 in `load_password` and 3240.2 cycles in `check_password`. Expectably, `LOTRx86` shows a clear performance advantage over the traditional mechanisms in the same experiment on the AMD machine (Figure 5b). Regarding the performance difference between the two execution modes of `LOTRx86`, we observe that `PrivUser64` performs better than `PrivUser` in both functions; `PrivUser64` performs 17.9% faster in `load_password` and 10.9% in `check_password`.

7.3 LOTRx86-Enabled Web Server

To develop a proof-of-concept `LOTRx86`-enabled webserver, we made changes to LibreSSL and the Nginx web browser. Specifically, we replaced parts of the software that accesses private keys with a `privcall` routine that performs the equivalent task. In the resulting webserver's process address space, the private key always resides in the `PrivUser` memory space. Therefore, any arbitrary memory access (e.g., buffer over-read in HeartBleed) is thwarted. Only through the pre-defined `privcall` routines, the webserver can perform operations that involve the private key.

Implementation. During its initialization, Nginx loads the private key through a function called `SSL_CTX_use_PrivateKey_file`. This function performs a series of operations to read the private key and then parse the contents into an ASN1 structure. The function eventually produces an RSA structure that LibreSSL uses during SSL connections. We re-implemented the function using `privcalls`. In our version of the function, the opening of the file and loading its contents into memory are performed in `PrivUser` mode, and the structures that contain the private key or its processed forms are stored in the `PrivUser` memory space. For passing arguments, we created a custom C structure that contains the necessary information that needs to be passed via `privcalls`. Once the private key is converted into an RSA data structure, it is stored safely in the `PrivUser` memory space until it needs to be accessed during the handshake stage in an SSL connection. During the handshake, the server digitally signs a message using the private key to authenticate itself to its client. We modified the `RSA_sign()` such that it makes `privcalls` to request operations involving the RSA structure. In more detail, we copy the message to be signed in the argument page shared between user mode and `PrivUser` mode that is designated by `liblotr`.

The `LOTRx86`-enabled libreSSL and Nginx were compiled to 32-bit (`PrivUser32`) and 64-bit (`PrivUser64`). Since the two supported `PrivUser` execution modes share the same `privcall` interface, we were able to compile the webserver implementation to both `PrivUser` modes with only minor changes. One factor that may induce the performance difference between the two modes is the argument passing. This allows us to run the same benchmark on the two `PrivUser` modes and evaluate their performances.

Performance measurements. We used the *ab* apache benchmark tool to perform a benchmark similar to the one performed in [14], a work that leverages hypervisor to achieve a similar objective to `LOTRx86`. Using the tool, we make 10,000 KeepAlive requests to the server, then the server responds by sending a data block back to the client. In the benchmark, we measured the average execution time from the socket connection to the last response from the server. The client that runs `ab` to perform the benchmarks, was measured to have an end-to-end latency of around 5ms ~8ms to the webserver. On the other hand, the local client is placed in the same machine as the webserver to better capture the latency induced by `LOTRx86`. We also

varied the size of the requested file size to represent different configurations. (we used {100B, 512B, 1k, ... 100k, 500k} and [14] uses {5k, 20k, 50k}). The results are shown in Figure 6.

The additional performance overhead due to LOTRx86 mainly comes from the execution mode transition (user mode to PrivUser32 or PrivUser64 mode). A total of three privcall invocations are made in opening and loading the contents of the private key file into a buffer in PrivUser memory space, and a single privcall to sign the message using the private key.

Both LOTRx86 with PrivUser32 and PrivUser64 show similar performance characteristics over the varied file sizes. PrivUser32 shows the worst case at 512B with 70.5% on Intel. However, the performance overhead attenuates as the file size increases. For instance, 21.0% overhead is observed for 100K, and the overhead becomes nearly negligible for 512K with 4.8%. PrivUser32 performs marginally better in AMD machines for most file size cases; notably, it showed 53.1% in 512B. The PrivUser64 mode of LOTRx86 mostly surpasses the PrivUser32 mode in all cases and on both machines. On the Intel machine, the worst case was measured to be 45.3% when the file size was 1K. However, it marked around 30% to 35% overhead in most (100B, 512B, 5K, 20K) cases and showed low overhead for larger file sizes (16.6% at 100K and 3.3% at 512K). While slightly higher, the performance overhead measured on the AMD machine was similar to that of the Intel machine.

PrivUser32 vs. PrivUser64. in webserver performance. Our experiment shows PrivUser64 mode brings a notable performance increase in latency-sensitive workloads. The performance increase in the webserver experiment is not directly proportionate to the pure latency differences shown in PrivUser32 vs. PrivUser64 microbenchmark (Figure 4). We surmise that the largest factor responsible for the performance gain in PrivUser64 is the performance difference between the 64-bit mode and the 32-bit mode. Also, there might be microarchitectural negative performance side effects from switching between 64-bit and 32-bit modes. That is, it is highly unlikely that the microarchitectural optimizations have taken the case of bitness change during program exchange. In all, we conclude that the LOTRx86 PrivUser64 lowers the effort required to port an application to use LOTRx86 and brings performance gains in general applications, as shown through our experiment.

8 Related Work

The LOTRx86 architecture creates a new protected domain in user space using only the existing features through its unique design. LOTRx86 can be a practical alternative to the recently introduced hardware security features when the software must be deployed to general users. In this section, we discuss previous work on user-space memory protection and system privilege restructuring methods for system fortification.

Alternative Privilege Models. Nested kernel [30] introduced a concept of inner-kernel that takes control of the hardware MMU by deprivileging the original kernel by disabling a subset of its Ring 0 power. By removing all privileged instructions that may disable memory protection, Nested Kernel protects itself and the kernel memory mappings. Nested Kernel exports a virtual MMU interface that allows the deprivileged kernel to request sensitive memory management operations explicitly.

Dune leveraged the Intel VT-x virtualization technology to provide user-level programs with privileged system functionalities that were only allowed to kernels [31]. Dune migrates a user-level process in Ring 0 of the VT-x non-root mode, allowing the process to enjoy kernel privileges securely. This process in *Dune Mode* is dependent on the host kernel, and makes *hypercalls* to invoke the *root-mode* kernel system calls.

The x86-32 hypervisor implementation before the introduction of Intel's hardware-assisted virtualization features [32], [33], made use of the intermediate Rings and segmentation to achieve virtualization. Hypervisor implementations [34], [35], [36] deprivileged the operating system kernel by making them run in the intermediate Rings, then enforced segmentation to protect the in-memory hypervisor.

Use of hardware features. Some works employed the x86-32 segmentation feature for application memory protection [37], [38](and as aforementioned in early hypervisor implementations). Both Native Client(NaCl) and Vx32 provide a safe is a user-level sandbox that enables safe execution of guest plug-ins to the host program. Also, The Nacl sandbox has adopted SFI to compensate for the lack of segmentation in x86-64 [39].

Processor architectures have been extended to support user-space memory protection. Intel has recently introduced Software Guard eXtensions (SGX) that creates an enclave in which a predefined set of code and data can be protected [10]. It also provides new instructions to invoke the code residing in the enclave. Furthermore, Intel has been planning for the release of SGX version 2, which will support dynamic memory management [40].

Many works have leveraged the memory partitioning processor feature, MPK, to implement intra-process isolation [18], [19], [41]. LOTRx86 regards portability as one of the main design objectives while MPK requires hardware support and thus is not supported on older and legacy systems. Regarding performance, MPK's memory protection domain switching is known to consume 11-260 cycles [18], while privcall64 was shown to consume about 400 cycles on average. One challenge in adapting MPK is the necessity of removing unintended occurrences of MPK configuring instruction called wrpkru, a non-privileged instruction that can be abused to nullify the memory isolation. LOTRx86, on the other hand, has kernel-maintained privilege descriptor (e.g., the LDT) and uses hardware-defined callgates to switch privileges.

The fragmented hardware support for application-level memory protection served as a central motivation for our approach. Our design does not rely on any specific hardware feature and preserves portability. However, there are differences in the attack model and security guarantees. SGX distrusts kernel, and the protected user memory within its enclave stays intact even under kernel compromise. On the other hand, LOTR-x86 is incapable of operating in a trustworthy way when the kernel is compromised. Nevertheless, we argue that our work presents a unique

approach that achieves in-process memory protection while preserving portability.

Hypervisor-based Approaches: A number of works have leveraged hypervisors to protect applications in virtualized systems. memory [14], [15], [42], [43], [44], [45], [46]. xMP [47] leveraged the new virtualization hardware feature available on x86 processors to implement efficient intra-process isolation. The new feature allows a user process inside a VM to initiate a `EPTP` switching with a `VMFUNC` instruction without causing a VM exit. SEIMI [48] achieves intra-process isolation through the combination of the *Supervisor Mode Access Prevention (SMAP)* and virtualization features. Hypervisor-based approaches leverage hypervisor-controlled page tables and other hypervisor control over the virtualized system to ensure the trustworthiness of applications and system services. Similar to SGX, hypervisor-based approaches are designed on the premise that the kernel is vulnerable or possibly malicious. Additionally, these works assume the presence of a hypervisor on the system. On the contrary, we propose a portable solution that does not require special hardware features or virtualization technologies.

Process/thread level partitioning. Isolating program components into separate processes have been the traditional privilege separation method [3], [4], [5], [37], [49]. In essence, placing program components into process-level partitions aims to achieve complete address space separation. However, the approach presents many disadvantages. First, the approach inevitably involves a *Inter-Process Communication (IPC)* mechanism to establish a communication channel between the partitions so that they can remotely invoke functions (i.e., *Remote Procedure Calls (RPC)*) in other partitions and pass arguments as necessary.

The endeavor for in-application privilege separation continued, and more recent work used threads as a unit of separation compartment that prevents leakage of sensitive memory [6], [7], [8], [50]. Chen et al. [9] pointed out that even the thread compartments are still too coarse-grained, introduce a high-performance overhead due to page table switches, and require developers to make structural changes to their program. Their work, Shreds takes advantage of the memory domain feature on the ARM architecture to create a secure code block within the program. However, the Domain-based memory partitioning is only available and has been deprecated on `AARCH64`.

Our approach is fundamentally different from the previous work; the process and thread-level protection retrofit the protection mechanisms supported by the operating system kernel. However, our approach creates a new privilege layer in between the user and the kernel for the protection of sensitive application code and data. Another significant difference is that our approach does not require a full address space switch or runtime page table modifications.

Address-based isolation. Software-based fault isolation techniques [37], [38], [39], [51], [52], [53], [54] employ software techniques such as compilers or instrumentation to create logical fault domains within the same address space, often to contain code execution or memory access. SFI is often used to partition an untrusted module into a sandbox to protect the host program [37], [38], [39], [54]. More recently, WebAssembly [55] has been gaining traction as a byte-code virtual machine that incorporates mature SFI to create a sandbox for executing untrusted code in web browsers. Intel's MPX [12] technology had been introduced to provide hardware support for address-based isolation techniques[3].

Address-based isolation techniques can protect application secrets by applying bound checking to the program's load and store instructions that can potentially access the sensitive memory addresses [56]. On the other hand, ConfLLVM [57] proposes a process memory layout that reduces the overhead of runtime checks, whose effectiveness is shown through its MPX or x86 segment register enforcement mechanism.

`LOTRx86` creates a protected domain called `PrivUser` that consists of an isolated memory space and execution mode. Direct performance comparison between address-based and domain-based techniques can be difficult. For instance, address-based isolation techniques often require strong CFI defenses to be in place [56], [58] while `LOTRx86` (and similar domain) includes a controlled control transfer between the two domains.

Encryption and in-register data protection. Several works have explored the in-memory encryption and register-only computation of in-process sensitive data. DynPTA [59] keeps the sensitive data encrypted inside memory and selectively allows decrypting the data into the registers. Ginseng [60] protects application secrets from the untrusted OS kernel according to its attack model through the use of in-register data protection and memory encryption.

Automatic isolation and partitioning Automated identification of in-process secrets and their isolation have been studied through a number of existing works [49], [57], [59], [61], [61], [62]. DynPTA [59]'s selective encrypted data access scheme is backed by automatic identification of secret-accessing instructions achieved through its static and dynamic data-flow tracking. Similarly, Ginseng [60] and ConfLLVM [57] also automatically isolates sensitive data with respect to developer annotations. Glamdring [61] achieves source-level, function-granular automatic program partitioning for SGX, using static data-flow analysis. CryptoMPK [62] focuses on automatic identification and MPK-based protection of sensitive cryptographic data.

`LOTRx86`'s isolation is similar to that of so-called enclaves, as seen in the SGX model. Hence, using `LOTRx86` inherently requires a manual porting process.

GPU for secure computation Besides, PixelVault [63] proposed leveraging commodity GPUs such that they serve as secure key storage as well as an isolated cryptographic computation environment.

9 Limitations and Future Work

`LOTRx86` proposes a novel approach to application memory protection. However, the architecture is still in its infancy. We describe the limitations of the current prototype and discuss issues that need to be addressed.

3. Intel MPX has been deprecated since Linux kernel 5.6 and GCC 9

SMEP/SMAP. Intel's SMEP and SMAP [32] prevent supervisor mode (Ring0-2) from accessing or executing U-pages. SMEP does not affect LOTRx86 since PrivUser does not execute any code in u-pages. However, SMAP prevents PrivUser mode from accessing the argument page shared with user mode. One possible solution is to implement a system call or an ioctl call that toggles the SMAP enforcement such that PrivUser mode can fetch data from the shared page. Note that kernel's copy_from_user API also temporarily disables SMAP to copy from the user-supplied pointer to the argument.

Per-process kernel memory unmapping for PrivUser64. We plan to investigate the feasibility of isolating kernel memory space on a per-process basis. The PrivUser64 support relies on the presence of KPTI that isolates the kernel memory space from the user and PrivUser64 mode. While a large portion of Intel-based x86 processors requires KPTI, the latest and future (e.g., 12th gen and later) iterations supposedly do not require KPTI. Not to mention that AMD-based x86 processors have not been affected by Meltdown, and thus KPTI is disabled by default on AMD machines. KPTI accompanies a system-wide overhead since it has a noticeable impact on kernel entry and exit [27]. Our evaluation of PrivUser64 allowed us to confirm the approach's viability in terms of general application performance. Hence, we regard adapting the well-optimized kernel memory space isolation in the KPTI implementation such that it is only applied to processes with the LOTRx86's PrivUser64 mode enabled.

Further optimization. We believe there is room for further optimizations for the LOTRx86 architecture. However, finding resourceful optimization guides for using the intermediate Rings was absent due to their rare usage in modern operating systems. However, we plan to investigate further to improve the performance of our architecture.

Porting effort. Porting a program to use LOTRx86 to protect its secrets requires a manual porting process. This includes possible incompatibilities that may arise with the required compilation to 32-bit (in case of PrivUser32) and using the intermediate Rings. Further investigation on compatibility issues with various programs and partial or full automation of the porting process remains a future direction.

10 Conclusion

We presented LOTRx86, a novel approach that establishes a new user privilege layer called PrivUser that protects and safeguards secure access to application secrets. The new PrivUser memory space is protected from user mode access. We introduced the privcall interface that provides user mode a controlled invocation mechanism of the PrivUser routines to securely perform operations involving application secrets. Our design introduced unique privilege and control transfer structures that establish a new user mode privilege. We also explained how our design satisfies the security requirements for the PrivUser layer to have a distinct execution mode and memory space. In our evaluation, we showed that the latency added by a privcall is on par with frequently used C function calls such as ioctl and malloc. We also implemented and evaluated the LOTRx86-enabled Nginx web server that securely accesses its private key through the privcall interface. Using the Apache *ab* server benchmark tool, we measured the average keep-alive response time of the server to find the average overhead incurred by LOTRx86 in various response file sizes. The average overhead is limited to 30.40% on the Intel processor and 20.19% on the AMD processor.

11 Acknowledgments

This work was supported by National Research Foundation of Korea (NRF) Grant by the Korean Government through (NRF-2022R1C1C1010494), Institute of Information Communications Technology Planning Evaluation (IITP) grant funded by the Korea government (MSIT) (2022-0-01199), Research on Security of 5G-data-intensive Computing Platforms Based on Disaggregated Architecture (IITP-2020-0-00666), the High-Potential Individuals Global Training Program)(2021-0-01587), the National Research Foundation of Korea(NRF) grant (No. NRF-2020R1A2C2101134), and the Office of Naval Research (ONR) through Award N00014-18-1-2661.

References

[1] D. A. Wheeler, "Preventing heartbleed," *Computer*, vol. 47, no. 8, pp. 80–83, Aug 2014.

[2] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14. New York, NY, USA: ACM, 2014, pp. 475–488. [Online]. Available: http://doi.acm.org/10.1145/2663716.2663755

[3] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 16–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251369

[4] D. Kilpatrick, "Privman: A library for partitioning applications," in *2003 USENIX Annual Technical Conference (USENIX ATC 03)*. San Antonio, TX: USENIX Association, Jun. 2003. [Online]. Available: https://www.usenix.org/conference/2003-usenix-annual-technical-conference/privman-library-partitioning-applications

[5] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251375.1251380

[6] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting applications into reduced-privilege compartments," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 309–322. [Online]. Available: http://dl.acm.org/citation.cfm?id=1387589.1387611

[7] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer, "Enforcing least privilege memory views for multithreaded applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 393–405. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978327

[8] J. Wang, X. Xiong, and P. Liu, "Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '15. Berkeley, CA, USA: USENIX Association, 2015, pp. 361–373. [Online]. Available: http://dl.acm.org/citation.cfm?id=2813767.2813794

[9] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-grained execution units with private memory," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 56–71.

[10] I. Corperation, "Intel® software guard extensions (intel sgx)," https://software.intel.com/en-us/sgx, 2018, last accessed Feb 27 , 2018,.
[11] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 8:1–8:26, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2799647
[12] I. Corperation, "Introduction to intel® memory protection extensions," https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, 2018, last accessed Feb 22 , 2018,.
[13] J. Corbet, "Memory protection keys," https://lwn.net/Articles/643797/, 2015.
[14] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1607–1619. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813690
[15] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," *SIGPLAN Not.*, vol. 43, no. 3, pp. 2–13, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1353536.1346284
[16] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp
[17] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.
[18] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner
[19] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 489–504. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/hedayati-hodor
[20] A. Limited, "Building a secure system using trustzone® technolog," http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.
[21] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 523–539. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk
[22] T. W. David Kaplan, Jeremy Powell, *White Paper: AMD Memory Encryption*, http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, AMD, April 2016.
[23] "musl libc," https://www.musl-libc.org, 2018, last accessed Jan 23, 2018.
[24] I. Corperation, "System v application binary interface," https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf, 2018, last accessed Feb 21 , 2018,.
[25] L. K. Organization, "The linux kernel archives," https://www.kernel.org, 2018, last accessed April 2 , 2018,.
[26] J. Corbet, "Kernel page-table isolation merged," https://lwn.net/Articles/742404/, 2017.
[27] ——, "Kaiser: hiding the kernel from user space," https://lwn.net/Articles/738975/, 2017.
[28] N. Inc, "Nginx," https://www.nginx.com, 2018, last accessed Feb 27 , 2018,.
[29] OpenBSD, "Libressl," http://www.libressl.org, 2017, last accessed Feb 27 , 2018,.
[30] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 191–206, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2786763.2694386
[31] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged cpu features," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 335–348. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387913
[32] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, December 2016, no. 325462-061US.
[33] *AMD64 Architecture Programmer's Manual*, http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf, AMD, May 2013.
[34] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177. [Online]. Available: http://doi.acm.org/10.1145/945445.945462
[35] O. Corporation, "Virtualbox technical documentation," https://www.virtualbox.org/wiki/Technical_documentation, 2017, last accessed Aug 23, 2017.
[36] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang, "Bringing virtualization to the x86 architecture with the original vmware workstation," *ACM Trans. Comput. Syst.*, vol. 30, no. 4, pp. 12:1–12:51, Nov. 2012. [Online]. Available: http://doi.acm.org/10.1145/2382553.2382554
[37] 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 2009 30th IEEE Symposium on Security and Privacy*, ser. SP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 79–93. [Online]. Available: http://dx.doi.org/10.1109/SP.2009.25
[38] B. Ford and R. Cox, "Vx32: Lightweight user-level sandboxing on the x86," in *USENIX 2008 Annual Technical Conference*, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 293–306. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404014.1404039
[39] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures," in *Proceedings of the 19th USENIX Conference on Security*, ser. USENIX Security'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929820.1929822
[40] *Intel Software Guard Extensions Programming Reference*, https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, INTEL, Oct 2014.
[41] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel MPK)," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds. USENIX Association, 2019, pp. 241–254. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/park-soyeon
[42] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," *SIGPLAN Not.*, vol. 48, no. 4, pp. 265–278, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499368.2451146
[43] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "Minibox: A two-way sandbox for x86 native code," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 409–420. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_yanlin
[44] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 143–158.
[45] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. New

York, NY, USA: ACM, 2008, pp. 71–80. [Online]. Available: http://doi.acm.org/10.1145/1346256.1346267

[46] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel, "Sego: Pervasive trusted metadata for efficiently verified untrusted system services," *SIGOPS Oper. Syst. Rev.*, vol. 50, no. 2, pp. 277–290, Mar. 2016. [Online]. Available: http://doi.acm.org/10.1145/2954680.2872372

[47] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xmp: Selective memory protection for kernel and user space," in *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp. 563–577. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00041

[48] Z. Wang, C. Wu, M. Xie, Y. Zhang, K. Lu, X. Zhang, Y. Lai, Y. Kang, and M. Yang, "SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 592–607.

[49] S. Liu, G. Tan, and T. Jaeger, "Ptrsplit: Supporting general pointers in automatic program partitioning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2359–2371.

[50] S. Kamara, P. Mohassel, and B. Riva, "Salus: A system for server-aided secure function evaluation," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 797–808. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382280

[51] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '93. New York, NY, USA: ACM, 1993, pp. 203–216. [Online]. Available: http://doi.acm.org/10.1145/168619.168635

[52] 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*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 75–88. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298463

[53] S. McCamant and G. Morrisett, "Evaluating sfi for a cisc architecture," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS'06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267336.1267351

[54] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "RockSalt: Better, Faster, Stronger SFI for the x86," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 395–404. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254111

[55] WebAssembly, "Webassembly," https://webassembly.org, 2021.

[56] S. A. Carr and M. Payer, "Datashield: Configurable data confidentiality and integrity," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: ACM, 2017, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/3052973.3052983

[57] A. Brahmakshatriya, P. Kedia, D. P. McKee, D. Garg, A. Lal, A. Rastogi, H. Nemati, A. Panda, and P. Bhatu, "Conflllvm: A compiler for enforcing data confidentiality in low-level code," in *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, G. Candea, R. van Renesse, and C. Fetzer, Eds. ACM, 2019, pp. 4:1–4:15. [Online]. Available: https://doi.org/10.1145/3302424.3303952

[58] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 437–452. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064217

[59] T. Palit, J. F. Moon, F. Monrose, and M. Polychronakis, "DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1919–1937. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00082

[60] M. H. Yun and L. Zhong, "Ginseng: Keeping secrets in registers when you distrust the operating system," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/ginseng-keeping-secrets-in-registers-when-you-distrust-the-operating-system/

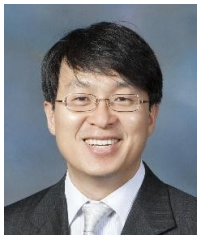
[61] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. M. Eyers, R. Kapitza, C. Fetzer, and P. R. Pietzuch, "Glamdring: Automatic application partitioning for intel SGX," in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, D. D. Silva and B. Ford, Eds. USENIX Association, 2017, pp. 285–298. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind

[62] X. Jin, X. Xiao, S. Jia, W. Gao, H. Zhang, D. Gu, S. Ma, Z. Qian, and J. Li, "Annotating, tracking, and protecting cryptographic secrets with cryptompk," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 473–488.

[63] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Pixelvault: Using gpus for securing cryptographic operations," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1131–1142. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660316

Hojoon Lee is currently an assistant professor at the Dept. of Computer Science and Engineering at Sungkyunkwan University since September 2019. Prior to his current position, he spent one year as a postdoctoral researcher at CISPA under the supervision of Prof. Michael Backes. He received Ph.D. from KAIST in 2018, advised by Prof. Brent Byunghoon Kang and his B.S. from The University of Texas at Austin. His main research interests lie in retrofitting security in computing systems against today's advanced threats. His research interests include but are not limited to Operating System Security, Trusted Execution Environments, Program Analysis, Software Security, and Secure Machine Learning Computation in Cloud.

Chihyun Song received his B.S. degree in Computer Science from Yonsei University (2017). He also received his M.S. in the Graduate School of Information Security at Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2019. He is currently in his Ph.D. course at the Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST). His research interests are trusted execution environments and intrakernel privilege separation.



Brent Byunghoon Kang is currently a Professor in the Graduate School of Information Security, School of Computing at KAIST (Korea Advanced Institute of Science and Technology). He received Ph.D. in Computer Science from the University of California at Berkeley, an M.S. from the University of Maryland, College Park, and a B.S. from Seoul National University. He has also been with George Mason University as an Associate Professor. His research interests include designing trusted computing/execution environments, OS kernel integrity monitors/memory defenses, hardware assisted systems security, botnet defenses and dialect computing. He is currently a member of the IEEE, the USENIX and the ACM.