Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86

Hojoon Lee[*]
CISPA Helmholtz Center i.G.
hojoon.lee@cispa.saarland

Chihyun Song
GSIS, School of Computing, KAIST
chihyun.song@kaist.ac.kr

Brent Byunghoon Kang
GSIS, School of Computing, KAIST
brentkang@kaist.ac.kr

ABSTRACT

Modern applications often involve processing of sensitive information. However, the lack of privilege separation within the user space leaves sensitive application secret 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.

CCS CONCEPTS

• **Security and privacy** → *Trusted computing*;

KEYWORDS

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 [20, 45].

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 [7, 29, 41]. The fundamental idea is to utilize the process separation mechanism provided by the OS; these work 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, 24, 44] have reduced the protection granularity compared to the process-level separation schemes while still suffering from performance overhead from page table modifications. Shreds presented fine-grained in-process memory protection using a memory partitioning feature that has long been present in ARM called *Memory Domains* [11]. 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 rest of the application as well as the possibly malicious kernel [4, 14]. Intel also offers hardware-assisted in-process memory safety and protection

[*] affiliation changed from KAIST to CISPA as of Sept 2018

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243748

features [13, 15] and AMD has announced the plans to embed a similar feature to its future generations of x86 processors [19]. However, the support for the new processor features are fragmented; not only that the features are not inter-operable across processors from different vendors (Intel, AMD), they are also only available on the newer processors. Hypervisor-based application memory protection [10, 35] 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 her 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. An *in-process* memory separation should not require a complete address space switching to access the protected memory or costly page table modifications.

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 does not require costly run-time page table manipulations nor 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 applications secret within the `PrivUser` memory space and also write `privcall` routines that 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. Based on our prototype, we developed a proof-of-concept LOTRx86-enabled web server. 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 does not require 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 adapt the two privilege level model 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 privilege: User and Supervisor. The Ring 3 can only access User pages while Ring 0-2 are allowed to access Supervisor pages[1]. 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

```
procedure CG:$R_n \rightarrow R_m(SEGSEL)$
    $DESC\_TBL \leftarrow$ if $SEGSEL.ti$ ? $LDT$ : $GDT$
    $CG \leftarrow DESC\_TBL[SEGSEL.idx]$
    if $n > CG.RMPL$ or $n \leq m$ then
        return $DENIED$
    end if
    Save(%RIP,%CS,%RSP,%SS)          ▷ Save caller context in temp space
    $\%SS \leftarrow TSS[m].SS$          ▷ Load new context to be used in Ring $m$
    $\%RSP \leftarrow TSS[m].RSP$
    $\%CS \leftarrow CG.TargetCS$          ▷ Privilege Escalation: $n \rightarrow m$
    $\%RIP \leftarrow CG.TargetEntrance$
    Push $SavedSS$
    Push $SavedRSP$
    Push $SavedCS$
    Push $SavedRIP$
    RESUME
end procedure
```

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

```
procedure Long Return
    ▷ can only return to equal or lower privileges
    if $DestPriv < CurrentPriv$ then
        return $DENIED$
    end if
    $\%RIP \leftarrow Pop()$          ▷ target addr
    $\%CS \leftarrow Pop()$          ▷ target ring privilege
    $tempRSP \leftarrow Pop()$
    $tempSS \leftarrow Pop()$
    $\%RSP \leftarrow tempRSP$
    $\%SS \leftarrow tempRSP$
    RESUME
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 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[2] 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 less 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:

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

Privilege de-escalation. A context can return to its original privilege mode with a long call instruction[3] 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

[1] Intel and AMD have introduced a CPU feature called *Supervisor Mode Execution Prevention (SMEP)* and *Supervisor Mode Access Prevention (SMAP)*. SMEP prevents contexts in Ring 0-2 from executing code in User pages, effectively preventing ret2usr style of attacks. SMAP prevents the kernel from accessing user pages as data [12, 27]

[2] "`lcall`" in AT&T syntax and "`call far`" in Intel syntax

[3] "`lret`" in AT&T syntax and "`retf`" in Intel syntax

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 *non-controlled control transfer* mechanism that is used to de-escalate privileges. We denote this specific type of control transfer where privilege is de-escalated (or stays the same) from m to n as the following:

$R_m \rightarrow R_n$, 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 application secret. We assume such vulnerabilities are present when the app has 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 that they are verifiable and absent of vulnerabilities.

3.2 Security guarantees

Our work focus 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 protection of application secret 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 [14, 34]. For instance, a recent work [32] 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 is one of the key contributions of this paper.

4 LOTRX86 DESIGN

The primary goal of the LOTRx86 design is to establish a new user memory protection and access mechanism through the introduction of new user mode privilege called `PrivUser`. Our design eliminates the necessity for page table switching or manipulation; the access to the protected memory regions is granted based on the privilege. Our architecture approaches the problem of application memory safety at a fundamental level. Instead of leveraging the existing OS-supported protection mechanisms, we create another privilege distinction within the user program execution model that resembles the user vs. kernel privilege.

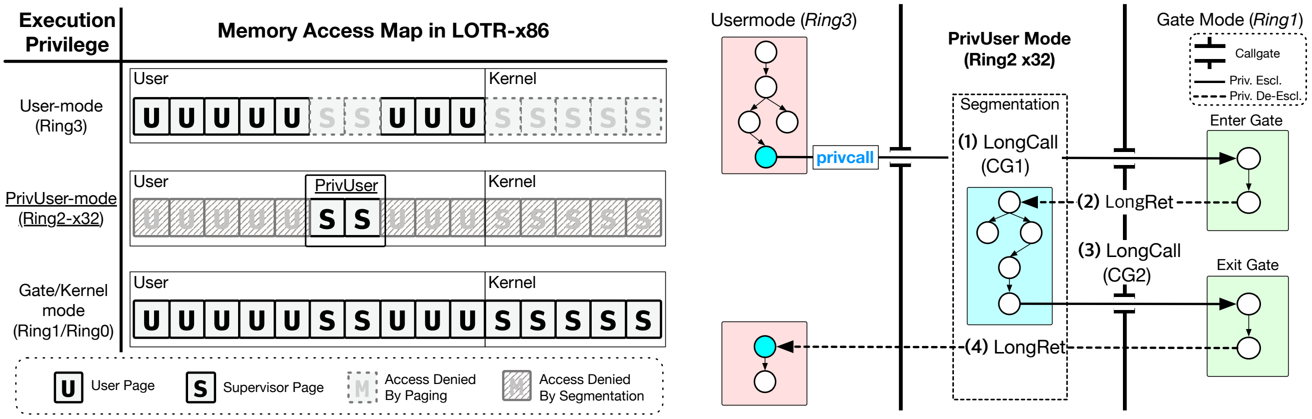
Developers can write sensitive memory handling routines into `PrivUser` layer, then simply place a `privcall` in place to invoke a routine that she defined. Below is the `privcall` interface that we provide to developers:

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

The `privcall` interface and its ABI is 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 who have experience in POSIX system programming, using the `privcalls` to perform operations that involve application secret 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 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 [11].



(a) LOTRx86 application 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.

Figure 1: LOTRx86 architecture overview

Portability. LOTRx86 does not rely on new processor features for memory protection [13–15, 19]. Instead, we re-purpose the underused privilege layers to implement `PrivUser`. Hence, our architecture is compatible across all generations of x86-64 processors. As we will present in section 6, we evaluated our architecture and a PoC on both Intel and AMD's x86-64 processors.

4.1 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 distinction: `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 protect 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 distinction recognized by the paging system, and we are unable to protect the kernel from `PrivUser` mode.

Satisfying M-SR2. LOTRx86 adapts a scheme that temporarily enables segmentation when a certain code segment is in use; we enforce `PrivUser` mode to be a *segmentation-enforced execution mode* by defining it 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 are capable of accessing `S`-pages. The resulting memory access map of the three execution mode is illustrated in Figure 1a. With our design, the `PrivUser` memory space serves as a *functionally* intermediate memory space for `PrivUser`.

Remaining challenge. However, we found that satisfying M-SR2 is a non-trivial issue. The segmentation enforcement alone is not sufficient for ensuring M-SR2. As mentioned in subsection 3.2, we must guarantee that the `PrivUser` layer has architecturally well-defined memory boundary against kernel; we must ensure that all memory access under any circumstances should not be able to affect kernel. The challenge is that we must carefully inspect all possible inter-segment control transfer from `PrivUser` mode, to verify that `PrivUser` mode cannot enter a state where it can access kernel memory.

4.2 Inescapable segmentation enforcement

LOTRx86 needs to guarantee that `PrivUser` mode is architecturally confined. Hence, we need to ensure it cannot escape the segmentation to access kernel memory. More specifically, we need to 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). We denote this control transfer security requirement for the enforcement of inescapable segmentation as CT-SR, and we explain how our privilege definitions (i.e., entries in LDT) and our gate structure (Figure 1b) satisfy the above requirement.

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

Hardware constraint and gate mode. Along with the CT-SR, 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 $CG : R_{n\_x32} \longrightarrow R_m$ is possible, $CG : R_m \longrightarrow R_{n\_x32}$ is an invalid callgate definition. Due to this constraint C, a privilege escalation and a switch to the 32-bit mode cannot be achieved in a single callgate transfer. Therefore, we that we need a separate 64-bit Gate mode segment to elevate privilege, then enter the 32-bit PrivUser mode. However, there exists a more important reason for the existence of the 64-bit Gate mode and that its privilege R_g must be higher than that of PrivUser mode (R_p).

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

Inspecting non-controlled control transfer routes. As we explained in section 2, an uncontrolled 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 are $e \geq 2$ (Ring privilege levels that are numerically equal or greater, meaning equal or lower privilege). First of all, we must make sure that a context in PrivUser mode cannot arbitrarily jump into an arbitrary place in Gate mode. In order to prevent a non-controlled control transfer $R_p \longrightarrow R_g$, 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 ⟶ 64bit) inter-segment control flow. We prevent such route by intentionally not defining a 64-bit code segment for the Ring level 2. A Ring privilege level in the x86 architecture come 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 the following:

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

Our privilege definitions and gate structures (Table 2 and Figure 1b) meet the constraint C. C is satisfied by the Enter gate in Gate mode. 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 are 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. Our design also satisfies CT-SR by maintaining the required properties P1 and P2. We chose Ring1 as the privilege level for Gate mode while enforcing the segmentation on all PrivUser mode execution by defining only 32-bit segmentation-enforced code segment for the Ring level 2. 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 PrivUser mode is architecturally bound to its memory space that it cannot access kernel memory under all circumstances.

5 PROTOTYPE 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 the use of 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* [1] 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 collection of compilation scripts and tools that help developers in compiling the PrivUser portion of their application and incorporating 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 need 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.

The initialization also set the Gate-mode stack to be loaded at the Enter callgate. As briefly explained in section 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 for each Ring levels. 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 request delivered via the ioctl requests from the host to thwart any possible attempt to compromise the PrivUser-space.

	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 ≤ 3
CG2	CG2 (R2→R1)	CPL ≤ 2

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

```
# 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;
```

Figure 2: Simplified pseudo assembly code of LOTRx86 Enter gates

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 [16]. 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.

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 section 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 [40]. 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 × 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 program. Writing a privcall routine is similar to writing a regular user-level code. However, there are a few key differences both in developer's perspective and underneath the surface. Here, we outline the important aspects in LOTRx86-enabled program development. Figure 3 illustrates the overall build process of a LOTRx86-enabled executable.

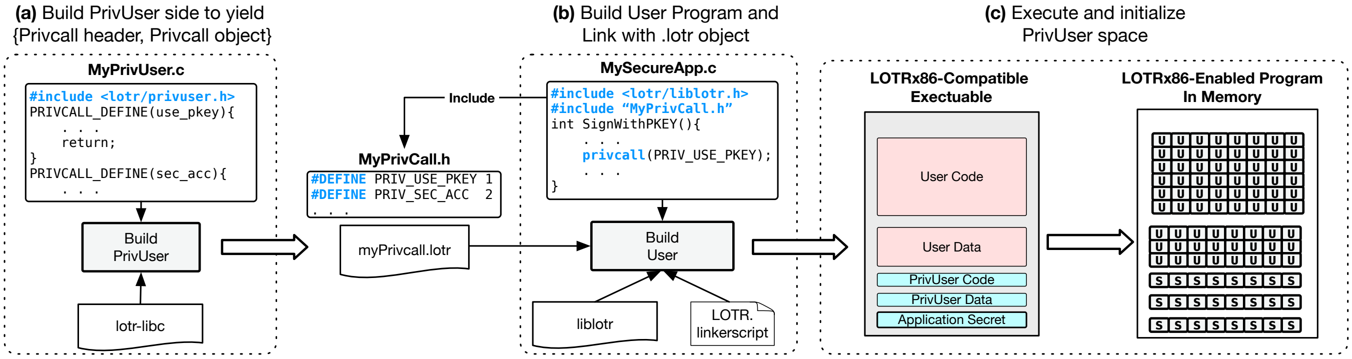
The privcall interface and the development of privcall routines are intentionally modeled after the Linux kernel's system call interface. For this reason, the procedures for developing the PrivUser side of the program and invoking them as necessary are nearly identical to those of developing new system 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.

Argument passing. liblotr provides an argument page that is always allocated within the 32-bit address space. Developers can first copy the argument into the argument page then pass the reference to PrivUser mode. Alternatively, developers can use LOTR_SECRET keyword to global variables to force them to be placed in a data section called .lotr_secret which is loaded into the PrivUser memory space.

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) Build PrivUser side to yield {Privcall header, Privcall object}
MyPrivUser.c
Build PrivUser
lotr-libc
Include
MyPrivCall.h
myPrivcall.lotr
(b) Build User Program and Link with .lotr object
MySecureApp.c
Build User
liblotr
LOTR. linkerscript
(c) Execute and initialize PrivUser space
LOTRx86-Compatible Exectuable
User Code
User Data
PrivUser Code
PrivUser Data
Application Secret
LOTRx86-Enabled Program In Memory
U
S

```
#include <lotr/privuser.h>
PRIVCALL_DEFINE(use_pkey){
    . . .
    return;
}
PRIVCALL_DEFINE(sec_acc){
    . . .
```

```
#DEFINE PRIV_USE_PKEY 1
#DEFINE PRIV_SEC_ACC  2
. . .
```

```
#include <lotr/liblotr.h>
#include "MyPrivCall.h"
int SignWithPKEY(){
    . . .
    privcall(PRIV_USE_PKEY);
    . . .
}
```

Figure 3: Building LOTRx86 compatible executable

a new 64-bit ELF object format so that it can be linked into 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 PROOF-OF-CONCEPT AND PERFORMANCE EVALUATION

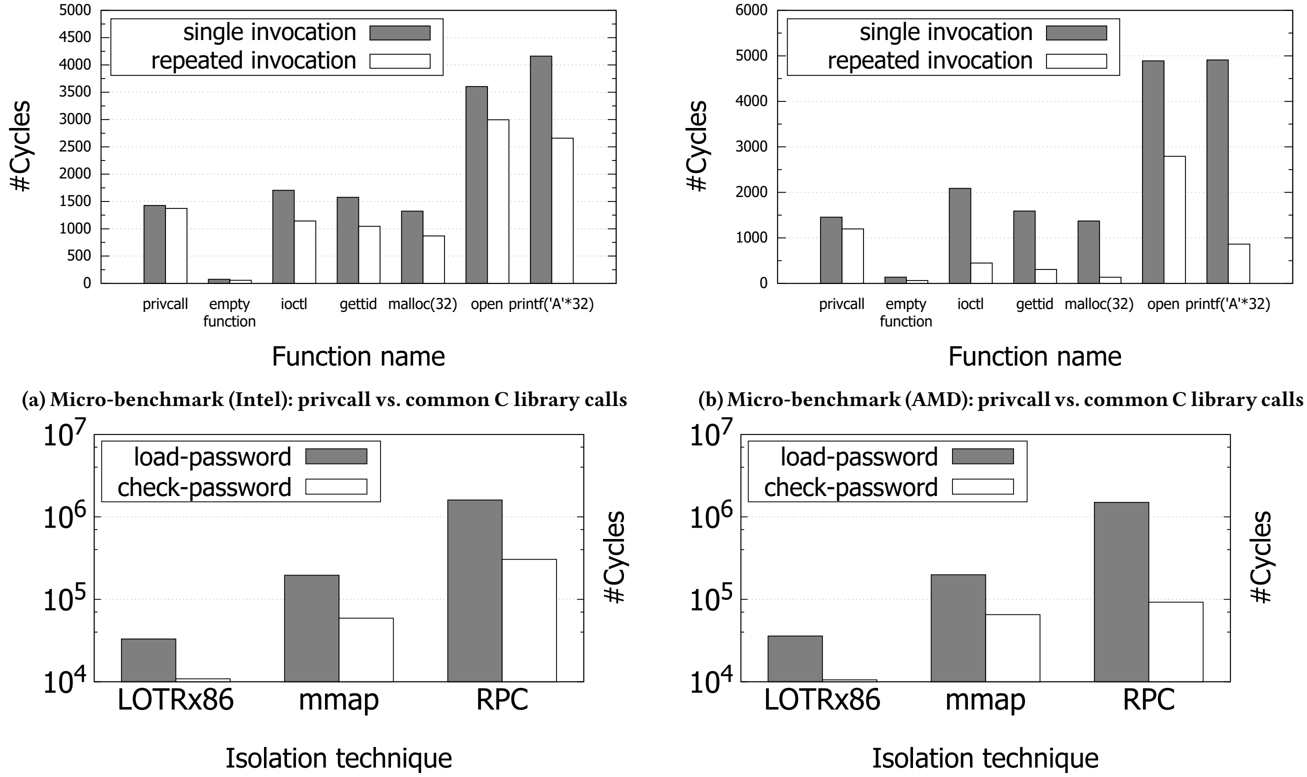
To show the feasibility and efficiency of the LOTRx86 architecture approach, we develop a proof-of-concept (PoC) by incorporating our architecture into the *Nginx* web server [25] as well as the *LibreSSL* [39] that is used by the web browser to support SSL. We modified the parts of the web server to protect the in-memory private key in the PrivUser memory space and only allow accesses to the key through our privcall interface. In this section, we present the results from a set of microbenchmarks that we performed to measure the latency induced by a privcall. Then we compare the performance of the PoC web server whose private key is protected with its original version. Our experiments are conducted on both Intel and AMD to show that our approach to show that our approach is portable. It should be noted that the PoC and all examples are compiled as *64-bit programs*. The specifications of the two x86 machines are as follows:

Intel-based PC: i7-4770 @ 3.40GHz, 4 cores, 16GB RAM
AMD-based PC: Ryzen 7 1800X @ 3.60GHZ, 8 cores, 32GB RAM
OS on both PC: Ubuntu 16.04 LTS, kernel version 4.13

6.1 Microbenchmarks

The privcall allows developers to invoke routines that access application secrets in the PrivUser layer. A certain amount of added latency is inevitable since we perform a chain of control transfers to securely invoke the privcall routines. In this experiment, we compare the latency of an empty privcall against the commonly used library calls to show that the added overhead is indeed a reasonable trade-off for the protection of application secrets. 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 a 1000 consecutive invocations of each call in a loop. These two setups represent the two situations where privcall is infrequently called and frequently called.

Figure 4a shows the experiment results on the Intel PC while Figure 4b shows the results from the same experiment on the AMD PC. The latency incurred by a privcall proves to be at a reasonable level. The single invocation performance is on par with the most basic library calls such as ioctl or gettid, consuming around 1000-1500 cycles on both PCs. It is noticeable that the latency of a privcall does not improve drastically as some of the other calls such as gettid whose number of cycles has dropped from 1575 to 1044 on Intel PC. As to this result, we surmise that the control flow transfer chain used in our architecture affect 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 future. However, LOTRx86 is the only portable solution on the x86 architecture that achieves in-process memory protection. Also, it is the most efficient solution among portable solutions. We present a comparison of LOTRx86 against the traditional memory protection techniques to support our claim. Moreover, we argue that the performance overhead of LOTRx86 is



(a) Micro-benchmark (Intel): privcall vs. common C library calls

(b) Micro-benchmark (AMD): privcall vs. common C library calls

(c) Execution time of LOTRx86 vs. traditional memory protection methods (Intel)

(d) Execution time of LOTRx86 vs. traditional memory protection methods (AMD)

Figure 4: Micro-benchmark `privcall` vs common library calls (a,b) and comparison against traditional memory protection methods (c,d)

at a reasonable level in an application scale. We present the macro benchmark results using our PoC (Nginx with LOTRx86).

6.2 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>`).

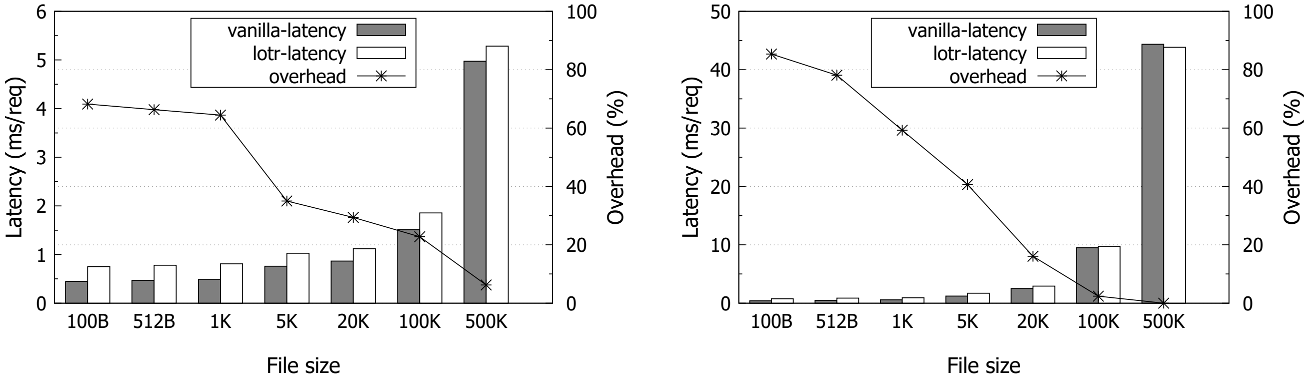
Test program. Our simple program first load a password from a file into the protected memory region, then it receives an input from the user via stdin to compare it against the protected password. In more detail, we implemented two functions `load_password` and `check_password` using the three protection 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 case of the RPC mechanism, we simply place the two measured functions and the password-storing buffer in a different process and make RPCs to 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 4. We averaged the results from 1000 trials (the y-axis is in log scale). The results show that LOTRx86 proves to be much faster than the two traditional methods by a large margin. On Intel PC, LOTRx86 greatly reduces the execution time (33051 cycles) of `load_password` by 83.11% (195661 cycles) and by 97.93% (1600291 cycles), compared to `mmap` and RPC-based implementation, respectively. This is because LOTRx86 does not require page table modifications that may cause system-wide performance overhead, nor the cost of communication with an external entity.

6.3 LOTRx86-enabled web server

To develop a proof-of-concept LOTRx86-enabled web server, 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 web server'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) are thwarted. Only through the pre-defined `privcall` routines, the web server 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`.



(a) Nginx latency measurements on Intel

(b) Nginx latency measurements on AMD

Figure 5: SSL KeepAlive response latency with varying file sizes on LOTRx86-enabled Nginx

This function performs a series of operations to read the private key then parsing the contents into an ASN1 structure, then the function eventually produces an RSA structure that is used by LibreSSL 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`.

Performance measurements. We used the *ab* apache benchmark tool to perform a benchmark similar to the one performed in [35], a work that leverages hypervisor to achieve a similar objective to LOTRx86. Using the tool, we make 1000 KeepAlive requests to the server, then the server responds by sending a file back to the client. In the benchmark, we measured the average execution time from the socket connection to the last response from the server. We also varied the size of the requested file size to represent different configurations. (we used {1k, 5k, 20k, 50k, 100k, 500k} and [35] uses {5k,20k,50k}). The results are shown in Figure 5a and Figure 5b. Note that due to the difference of CPU performance, the range of y-axes is different.

The additional performance overhead due to LOTRx86 mainly comes from the execution mode transition (user mode to `PrivUser` 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. In case of 5K requested file size, a rather extreme case, LOTRx86 adds about 35% on Intel processor and 40% on AMD. However, the overhead becomes relatively irrelevant as the file size increases: 20K: 29.36% (Intel), 16.07% (AMD), 500K : 6.25% (Intel) 0% (AMD). This particular experiment tests the feasibility of LOTRx86 in latency-critical tasks, and the results show that our approach is feasible even in such cases.

7 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 [18] introduced a concept of inner-kernel that takes control of the hardware MMU by deprivileging the original kernel by disabling a subset of its Ring0 power; By removing all privileged instruction that may disable memory protection, Nested Kernel protects itself and the kernel memory mappings. Nested Kernel exports virtual MMU interface that allows the deprivileged kernel to request sensitive memory management operation explicitly.

Dune leveraged the Intel VT-x virtualization technology to provide user-level programs with privileged system functionalities that were only allowed to kernels [5]. 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 such as Intel's VT-x and AMD's SVM [2, 27], made use of the intermediate Rings and segmentation to achieve virtualization. Hypervisor implementations [3, 8, 17] deprivileged the operating system kernel by making them run in the intermediate Rings then enforced segmentation to protect the in-memory hypervisor. LOTR-x86 not only explores the use of the intermediate Rings on 64-bit operating systems but also for a different purpose. Our architecture inhabits the abandoned Rings into more privileged user-mode in which developers can place their sensitive application code and data.

Use of processor features: Some works employed the x86-32 segmentation feature for application memory protection [22, 47](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. In this regard, the attack model of these work differs from that of LOTR-x86. LOTR-x86 assumes that the host application is untrusted and we place sensitive code and data in the PrivUser-space. Also, The Nacl sandbox has adopted SFI to compensate for the lack of segmentation in x86-64 [42]. LOTR-x86 enables inescapable in-place segmentation enforcement to construct the `PrivUser`-mode in x86-64.

Processor architectures have been extended to support user-space memory protection. Intel has recently introduced Software Guard eXtensions (SGX) that creates an enclave which a predefined set of code and data can be protected [14]. 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 that will support dynamic memory management [26]. Intel also ships memory bound checking functionality called MPX [15], which provides hardware assist for bound checking that software fault isolation approaches advocated. Intel has also disclosed plans for domain-based memory partitioning that resembles the memory domain feature in the ARM architecture. More recently AMD has published a white paper describing the upcoming memory encryption feature to be added to its x86 processors [19].

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.

Process/thread level partitioning: Early work on application privilege separation focused on restructuring a program into separate processes [7, 29, 41, 47]. In essence, placing program components into process-level partitions aims to achieve complete address space separation. Process-level partitioning provides architecturally (processor and kernel implementation) enforced separation, 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, 24, 28, 44]. Chen et al. [11] pointed out that even the thread compartments are still too coarse-grained, introduce a high-performance overhead due to page table switches, and requires 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 has been deprecated on `AARCH64`.

Our approach is fundamentally different from the previous work; the process and thread-level protection retrofit the protection mechanisms that are 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 address space switch nor run-time page table modifications. In our design, the *privilege* is what changes when granting access to the protected memory through `privcall`; the page tables that map the protected memory region as `S-pages` are intact whether the running context is in the normal user-mode or `PrivUser`-mode.

Hypervisor-based Approaches: A number of works have leveraged hypervisors to protect applications in virtualized systems. memory [10, 23, 31, 33, 35, 37, 46]. Hypervisor-based approaches leverage hypervisor-controlled page tables and other hypervisor control over the virtualized system to ensure trustworthiness of applications and system services. Similar to SGX, hypervisor-based approaches are designed on the premise that 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 nor virtualization technologies.

Address-based isolation. Software-based fault isolation techniques [21, 22, 36, 38, 42, 43, 47] 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 [22, 38, 42, 47]. The aforementioned Intel's MPX technology incorporates hardware support for address-based isolation techniques.

Address-based isolation techniques can protect application secrets by applying bound checking to program's load and store instructions that can potentially access the sensitive memory addresses [9]. On the other hand, LOTRx86 creates a 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 [9, 30] while LOTRx86 (and similar domain) includes a controlled control transfer between the two domains.

8 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 needs to be addressed.

SMEP/SMAP. Intel's SMEP and SMAP [27] prevents supervisor mode (Ring0-2) to access or execute `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 user-supplied pointer to the argument.

Argument passing. The current prototype of LOTRx86 requires the arguments to the `PrivUser` routines to be placed in the shared argument page in the 32-bit address space so that they can be accessed in `PrivUser` mode. Providing high-level APIs that facilitate convenient argument passing to the `PrivUser` of one of our most important future works. Another limitation that comes from the bitness difference is that the `PrivUser` execution mode is incapable of traversing the (64-bit) pointers. Nevertheless, we wish to point out that sharing complex objects across the untrusted and isolated domains is generally discouraged. That is, we advise developers to clearly define the necessary input to the `privcall` routines so that the routines do not need to traverse pointers any further than the ones that are passed as pass-by-reference arguments.

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

9 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 a 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 size. The average overhead is limited to 30.40% on the Intel processor and 20.19% on the AMD processor.

10 ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their valuable comments and feedback that helped us improve our paper. This work was supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA). This work was also supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2017R1A2B3006360).

REFERENCES

[1] 2018. musl libc. https://www.musl-libc.org. (2018). Last accessed Jan 23, 2018.

[2] AMD 2013. *AMD64 Architecture Programmer's Manual*. AMD.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*. ACM, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[4] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3, Article 8 (Aug. 2015), 26 pages. https://doi.org/10.1145/2799647

[5] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 335–348. http://dl.acm.org/citation.cfm?id=2387880.2387913

[6] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, Berkeley, CA, USA, 309–322. http://dl.acm.org/citation.cfm?id=1387589.1387611

[7] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1251375.1251380

[8] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. 2012. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst.* 30, 4, Article 12 (Nov. 2012), 51 pages. https://doi.org/10.1145/2382553.2382554

[9] Scott A. Carr and Mathias Payer. 2017. DataShield: Configurable Data Confidentiality and Integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*. ACM, New York, NY, USA, 193–204. https://doi.org/10.1145/3052973.3052983

[10] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. 2008. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. *SIGPLAN Not.* 43, 3 (March 2008), 2–13. https://doi.org/10.1145/1353536.1346284

[11] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. 2016. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy (SP)*. 56–71. https://doi.org/10.1109/SP.2016.12

[12] Jonathan Corbet. 2012. Supervisor mode access prevention. https://lwn.net/Articles/517475/. (2012).

[13] Jonathan Corbet. 2015. Memory protection keys. https://lwn.net/Articles/643797/. (2015).

[14] Intel Corperation. 2018. Intel® Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx. (2018). Last accessed Feb 27 , 2018,.

[15] Intel Corperation. 2018. 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,.

[16] Intel Corperation. 2018. System V Application Binary Interface. https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf. (2018). Last accessed Feb 21 , 2018,.

[17] Oracle Corporation. 2017. VirtualBox Technical documentation. https://www.virtualbox.org/wiki/Technical_documentation. (2017). Last accessed Aug 23, 2017.

[18] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 2015. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 191–206. https://doi.org/10.1145/2786763.2694386

[19] Tom Woller David Kaplan, Jeremy Powell. 2016. *White Paper: AMD Memory Encryption*. AMD.

[20] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC '14)*. ACM, New York, NY, USA, 475–488. https://doi.org/10.1145/2663716.2663755

[21] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 75–88. http://dl.acm.org/citation.cfm?id=1298455.1298463

[22] Bryan Ford and Russ Cox. 2008. Vx32: Lightweight User-level Sandboxing on the x86. In *USENIX 2008 Annual Technical Conference (ATC'08)*. USENIX Association, Berkeley, CA, USA, 293–306. http://dl.acm.org/citation.cfm?id=1404014.1404039

[23] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. InkTag: Secure Applications on an Untrusted Operating System.

SIGPLAN Not. 48, 4 (March 2013), 265–278. https://doi.org/10.1145/2499368.2451146

[24] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 393–405. https://doi.org/10.1145/2976749.2978327

[25] NGINX Inc. 2018. Nginx. https://www.nginx.com. (2018). Last accessed Feb 27 , 2018,.

[26] INTEL 2014. *Intel Software Guard Extensions Programming Reference.* INTEL.

[27] Intel Corporation. 2016. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Number 325462-061US.

[28] Seny Kamara, Payman Mohassel, and Ben Riva. 2012. Salus: A System for Server-aided Secure Function Evaluation. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 797–808. https://doi.org/10.1145/2382196.2382280

[29] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications.. In *USENIX Annual Technical Conference, FREENIX Track* (2003-09-03). USENIX, 273–284. http://dblp.uni-trier.de/db/conf/usenix/usenix2003f.html#Kilpatrick03

[30] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 437–452. https://doi.org/10.1145/3064176.3064217

[31] Youngjin Kwon, Alan M. Dunn, Michael Z. Lee, Owen S. Hofmann, Yuanzhong Xu, and Emmett Witchel. 2016. Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services. *SIGOPS Oper. Syst. Rev.* 50, 2 (March 2016), 277–290. https://doi.org/10.1145/2954680.2872372

[32] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. 2017. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 523–539. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk

[33] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. 2014. MiniBox: A Two-Way Sandbox for x86 Native Code. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 409–420. https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_yanlin

[34] ARM Limited. 2009. Building a Secure System using TrustZoneÂő Technolog. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. (2009).

[35] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1607–1619. https://doi.org/10.1145/2810103.2813690

[36] Stephen McCamant and Greg Morrisett. 2006. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15 (USENIX-SS'06)*. USENIX Association, Berkeley, CA, USA, Article 15. http://dl.acm.org/citation.cfm?id=1267336.1267351

[37] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation. In *2010 IEEE Symposium on Security and Privacy.* 143–158. https://doi.org/10.1109/SP.2010.17

[38] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 395–404. https://doi.org/10.1145/2254064.2254111

[39] OpenBSD. 2017. LibreSSL. http://www.libressl.org. (2017). Last accessed Feb 27 , 2018,.

[40] Linux Kernel Organization. 2018. The Linux Kernel Archives. https://www.kernel.org. (2018). Last accessed April 2 , 2018,.

[41] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 16–16. http://dl.acm.org/citation.cfm?id=1251353.1251369

[42] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the 19th USENIX Conference on Security (USENIX Security'10)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1929820.1929822

[43] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM, New York, NY, USA, 203–216. https://doi.org/10.1145/168619.168635

[44] Jun Wang, Xi Xiong, and Peng Liu. 2015. 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 (USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 361–373. http://dl.acm.org/citation.cfm?id=2813767.2813794

[45] D. A. Wheeler. 2014. Preventing Heartbleed. *Computer* 47, 8 (Aug 2014), 80–83. https://doi.org/10.1109/MC.2014.217

[46] Jisoo Yang and Kang G. Shin. 2008. 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 (VEE '08)*. ACM, New York, NY, USA, 71–80. https://doi.org/10.1145/1346256.1346267

[47] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP '09)*. IEEE Computer Society, Washington, DC, USA, 79–93. https://doi.org/10.1109/SP.2009.25