LAZARUS: Practical Side-channel Resilient Kernel-Space Randomization

David Gens[1], Orlando Arias[2], Dean Sullivan[2], Christopher Liebchen[1], Yier Jin[2], and Ahmad-Reza Sadeghi[1]

[1] CYSEC/Technische Universität Darmstadt, Germany.
{david.gens,christopher.liebchen,ahmad.sadeghi}@trust.tu-darmstadt.de
[2] University of Central Florida, Orlando, FL, USA.
{oarias,dean.sullivan}@knights.ucf.edu,yier.jin@eecs.ucf.edu

Abstract. Kernel exploits are commonly used for privilege escalation to take full control over a system, e.g., by means of code-reuse attacks. For this reason modern kernels are hardened with kernel Address Space Layout Randomization (KASLR), which randomizes the start address of the kernel code section at boot time. Hence, the attacker first has to bypass the randomization, to conduct the attack using an adjusted payload in a second step. Recently, researchers demonstrated that attackers can exploit unprivileged instructions to collect timing information through side channels in the paging subsystem of the processor. This can be exploited to reveal the randomization secret, even in the absence of any information-disclosure vulnerabilities in the software.
In this paper we present *LAZARUS*, a novel technique to harden KASLR against paging-based side-channel attacks. In particular, our scheme allows for fine-grained protection of the virtual memory mappings that implement the randomization. We demonstrate the effectiveness of our approach by hardening a recent Linux kernel with LAZARUS, mitigating all of the previously presented side-channel attacks on KASLR. Our extensive evaluation shows that LAZARUS incurs only 0.943% overhead for standard benchmarks, and therefore, is highly practical.

Keywords: KASLR, Code-Reuse Attacks, Randomization, Side Channels

1 Introduction

For more than three decades memory-corruption vulnerabilities have challenged computer security. This class of vulnerabilities enables the attacker to overwrite memory in a way that was not intended by the developer, resulting in a malicious control or data flow. In the recent past, kernel vulnerabilities became more prevalent in exploits due to advances in hardening user-mode applications. For example, browsers and other popular targets are isolated by executing them in a sandboxed environment. Consequently, the attacker needs to execute a privilege-escalation attack in addition to the initial exploit to take full control over the

system [4, 17, 18, 19]. Operating system kernels are a natural target for attackers because the kernel is comprised of a large and complex code base, and exposes a rich set of functionality, even to low privileged processes. Molinyawe et al. [20] summarized the techniques used in the Pwn2Own exploiting contest, and concluded that a kernel exploit is required for most privilege-escalation attacks.

In the past, kernels were hardened using different mitigation techniques to minimize the risk of memory-corruption vulnerabilities. For instance, enforcing the address space to be writable or executable (W$\oplus$X), but never both, prevents the attacker from injecting new code. Additionally, enabling new CPU features like Supervisor Mode Access Prevention (SMAP) and Supervisor Mode Execution Protection (SMEP) prevents certain classes of user-mode-aided attacks. To mitigate code-reuse attacks, modern kernels are further fortified with kernel Address Space Layout Randomization (KASLR) [2]. KASLR randomizes the base address of the code section of the kernel at boot time, which forces attackers to customize their exploit for each targeted kernel. Specifically, the attack needs to disclose the randomization secret first, before launching a code-reuse attack.

In general, there are two ways to bypass randomization: (1) brute-force attacks, and (2) information-disclosure attacks. While KASLR aims to make brute-force attacks infeasible, attackers can still leverage information-disclosure attacks, e.g., to leak the randomization secret. The attacker can achieve this by exploiting a memory-corruption vulnerability, or through side channels. Recent research demonstrated that side-channel attacks are more powerful, since they do not require any kernel vulnerabilities [6, 8, 10, 13, 23]. These attacks exploit properties of the underlying micro architecture to infer the randomization secret of KASLR. In particular, modern processors share resources such as caches between user mode and kernel mode, and hence, leak timing information between privileged and unprivileged execution. The general idea of these attacks is to probe different kernel addresses and measure the execution time of the probe. Since the timing signature for valid and invalid kernel addresses is different, the attacker can compute the randomization secret by comparing the extracted signal against a reference signal.

The majority of side-channel attacks against KASLR is based on *paging* [8, 10, 13, 23]. Here, the attacker exploits the timing difference between an aborted memory access to an unmapped kernel address and an aborted memory access to a mapped kernel address. As we eloberate in the related work Section 7 the focus of the existing work is on attacks, and only include theoretical discussions on possible defenses. For instance, Gruss et al. [8] briefly discuss an idea similar to our implemented defense by suggesting to completely un-map the kernel address space when executing the user mode as it is done in iOS on ARM [16]. However, as stated by the authors [8] they did not implement or evaluate the security of their approach but only provided a simulation of this technique to provide a rough estimation of the expected run-time overhead which is around 5% for system call intensive applications.

Goal and Contributions The goal of this paper is to prevent kernel-space randomization approaches from leaking side-channel information through the pag-

ing subsystem of the processor. To this end, we propose *LAZARUS*, as a novel real-world defense against paging-based side-channel attacks on KASLR. Our software-only defense is based on the observation that all of the presented attacks have a common source of leakage: information about randomized kernel addresses is stored in the paging caches of the processor while execution continues in user mode. More specifically, the processor keeps paging entries for recently used addresses in the cache, regardless of their associated privilege level. This results in a timing side channel, because accesses for cached entries are faster than cache misses. Our defense separates paging entries according to their privilege level in caches, and provides a mechanism for the kernel to achieve this efficiently in software. LAZARUS only separates those parts of the address space which might reveal the randomization secret while leaving entries for non-randomized memory shared. Our benchmarks show that this significantly reduces the performance overhead. We provide a prototype implementation of our side-channel defense, and conduct an extensive evaluation of the security and performance of our prototype for a recent kernel under the popular Debian Linux and Arch Linux distributions.

To summarize, our contributions are as follows:

- **Novel side-channel defense.** We present the design of *LAZARUS*, a software-only protection scheme to thwart side-channel attacks against KASLR based on paging.
- **Protoype Implementation.** We provide a fully working and practical prototype implementation of our defense for a recent Linux kernel version 4.8.
- **Extensive Evaluation.** We extensively evaluate our prototype against all previously presented side-channel attacks and demonstrate that the randomization secret can no longer be disclosed. We re-implemented all previously proposed attacks on KASLR for the Linux kernel. We additionally present an extensive performance evaluation and demonstrate high practicality with an average overhead of only 0.943% for common benchmarks.

2 Background

In this section, we first explain the details of modern processor architectures necessary to understand the remainder of this paper. We then explain the different attacks on KASLR presented by related work.

2.1 Virtual Memory

Virtual memory is a key building block to separate privileged system memory from unprivileged user memory, and to isolate processes from each other. Virtual memory is implemented by enforcing an indirection between the address space of the processor and the physical memory, i.e., every memory access initiated by the processor is mediated by a piece of hardware called the Memory Management Unit (MMU). The MMU translates the virtual address to a physical address, and

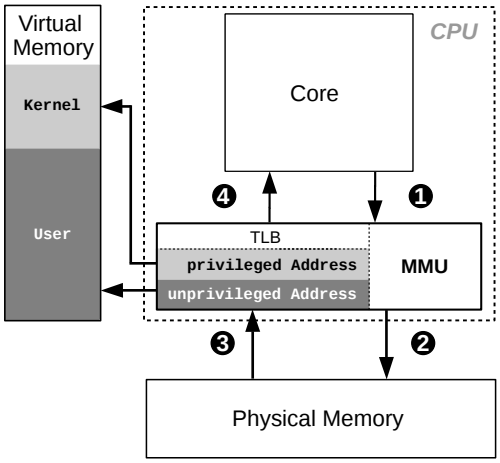

Fig. 1. When virtual memory is active, all memory accesses of the processor are mediated by the MMU ❶: it loads the associated page-table entry ❷ into the TLB from memory, checks the required privilege level ❸, and translates the virtual memory address into the corresponding physical memory address if and only if the current privilege level of the processor matches the required privilege level ❹.

enforces access control based on permissions defined for the requested address. The translation information as well as the access permissions are stored in a hierarchical data structure, which is maintained by the kernel, called the page table. The kernel isolates processes from each other by maintaining separate page tables for each process, and hence, different permissions. In contrast to processes, the kernel is not isolated using a separate page table but by setting the supervisor bit in page-table entries that translate kernel memory. In fact, each process page table contains entries that map the kernel (typically in the top part of the virtual address space). This increases the performance of context switches between the kernel and user applications because replacing the active page table forces the MMU to evict entries from its internal cache, called Translation Lookaside Buffer (TLB). The TLB caches the most recent or prominent page table entries, which is a sensible strategy since software usually exhibits (spatial or temporal) *locality*. Hence, all subsequent virtual-memory accesses, which are translated using a cached page-table entry, will be handled much faster.

Figure 1 shows the major components of virtual memory and their interaction. In the following we describe the MMU and the TLB in detail and explain their role in paging-based side-channel attacks.

The Central Processing Unit (CPU) contains one or more execution units (cores), which decode, schedule, and eventually execute individual machine instructions, also called operations. If an operation requires a memory access, e.g.,

load and store operations, and the virtual memory subsystem of the processor is enabled, this access is mediated by the MMU (Step ❶). If the page-table entry for the requested virtual address is not cached in the TLB, the MMU loads the entry into the TLB by traversing the page tables (often called a *page walk*) which reside in physical memory (Step ❷). The MMU then loads the respective page-table entry into the TLBs (Step ❸). It then uses the TLB entries to look up the physical address and the required privilege level associated with a virtual address (Step ❹).

2.2 Paging-based Side-channel Attacks on KASLR

All modern operating systems leverage kernel-space randomization by means of kernel code randomization (KASLR) [2, 11, 14]. However, kernel-space randomization has been shown to be vulnerable to a variety of side-channel attacks. These attacks leverage micro-architectural implementation details of the underlying hardware. More specifically, modern processors share virtual memory resources between privileged and unprivileged execution modes through caches, which was shown to be exploitable by an user space adversary.

In the following we briefly describe recent paging-based side-channel attacks that aim to disclose the KASLR randomization secret. All these attacks exploit the fact that the TLB is shared between user applications and the kernel (cf., Figure 1). As a consequence, the TLB will contain page-table entries of the kernel after switching the execution from kernel to a user mode application. Henceforth, the attacker uses special instructions (depending on the concrete side-channel attack implementation) to access kernel addresses. Since the attacker executes the attack with user privileges, the access will be aborted. However, the time difference between access attempt and abort depends on whether the guessed address is cached in the TLB or not. Further, the attacker can also measure the difference in timing between existing (requiring a page walk) and non-existing mappings (immediate abort). The resulting timing differences can be exploited by the attacker as a side channel to disclose the randomization secret as shown recently [8, 10, 13, 23].

Page Fault Handler (PFH) Hund, et al. [10] published the first side-channel attack to defeat KASLR. They trigger a page fault in the kernel from a user process by accessing an address in kernel space. Although this unprivileged access is correctly denied by the page fault handler, the TLBs are queried during processing of the memory request. They show that the timing difference between exceptions for unmapped and mapped pages can be exploited to disclose the random offset.

Prefetch Instruction Furthermore, even individual instructions may leak timing information and can be exploited [8]. More specifically, the execution of the `prefetch` instruction of recent Intel processors exhibits a timing difference, which depends directly on the state of the TLBs. As in the case of the other side-channel attacks, this is used to access privileged addresses by the attacker.

Since this access originates from an unprivileged instruction it will fail, and according to the documentation the processor will not raise an exception. Hence, its execution time differs for cached kernel addresses. This yields another side channel that leaks the randomization secret.

Intel's TSX Transactional memory extensions introduced by Intel encapsulate a series of memory accesses to provide enhanced safety guarantees, such as rollbacks. While potentially interesting for the implementation of concurrent software without the need for lock-based synchronization, erroneous accesses within a transaction are not reported to the operating system. More specifically, if the MMU detects an access violation, the exception is masked and the transaction is rolled back silently. However, an adversary can measure the timing difference between two failing transactions to identify privileged addresses, which are cached in the TLBs. This enables the attacker to significantly improve over the original page fault timing side-channel attack [13, 23]. The reason is that the page fault handler of the OS is never invoked, significantly reducing the noise in the timing signal.

3 LAZARUS

In this section, we give an overview of the idea and architecture of LAZARUS, elaborate on the main challenges, and explain in detail how we tackle these challenges.

3.1 Adversary Model and Assumptions

We derive our adversary model from the related offensive work [6, 8, 10, 13, 23].

- **Writable ⊕ Executable Memory**. The kernel enforces Writable ⊕ Executable Memory (W⊕X) which prevents code-injection attacks in the kernel space. Further, the kernel utilizes modern CPU features like SMAP and SMEP [12] to prevent user-mode aided code-injection and code-reuse attacks.
- **Kernel Address Space Layout Randomization (KASLR)**. The base address of the kernel is randomized at boot time [2, 14].
- **Absence of Software-based Information-disclosure Vulnerability**. The kernel does not contain any vulnerabilities that can be exploited to disclose the randomization secret.
- **Malicious Kernel Extension**. The attacker cannot load malicious kernel extensions to gain control over the kernel, i.e., only trusted (or signed) extensions can be loaded.
- **Memory-corruption Vulnerability**. This is a standard assumption for many real-world kernel exploits. The kernel, or a kernel extension contains a memory-corruption vulnerability. The attacker has full control over a user-mode process from which it can exploit this vulnerability. The vulnerability

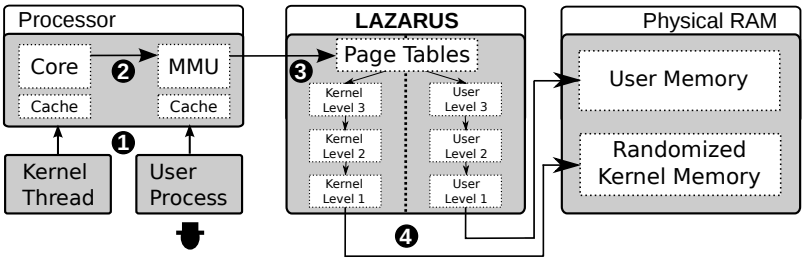

Fig. 2. The idea behind our side channel protection: An unprivileged user process (❶) can exploit the timing side channel for kernel addresses through shared cache access in the MMU paging caches (❷). Our defense mitigates this by enforcing (❸) a separation between different privilege levels for randomized addresses (❹).

enables the attacker to overwrite a code pointer of the kernel to hijack the control-flow of the kernel. However, the attacker cannot use this vulnerability to disclose any addresses.

While modern kernels suffer from software-based information-disclosure vulnerabilities, information-disclosure attacks based on side channels pose a more severe threat because they can be exploited to disclose information in the absence of software vulnerabilities. We address the problem of side channels, and treat software-based information-disclosure vulnerabilities as an orthogonal problem.

3.2 Overview

Usually, kernel and user mode share the same virtual address space. While legitimate accesses to kernel addresses require higher privilege, these addresses still occupy some parts of the virtual memory space that is visible to user processes. The idea behind our side-channel defense is to strictly and efficiently separate randomized kernel memory from virtual memory in user space.

Our idea is depicted in Figure 2. Kernel execution and user space execution usually share a common set of architectural resources, such as the execution unit (Core), and the MMU. The attacker leverages these shared resources in the following way: in step ❶, the attacker sets up the user process and memory setting that will leak the randomization secret. The user process then initiates a virtual memory access to a kernel address.

Next, the processor invokes the MMU to check the required privilege level in step ❷. Since a user space process does not possess the required privileges to access kernel memory, any such access will ultimately be denied. However, to deny access the MMU has to look up the required privileges in the page tables. These are structured hierarchically with multiple levels, and separate caches on every level. Hence, even denied accesses constitute a timing side-channel that directly depends on the last cached level.

We address ❸ the root of this side channel: we separate the page tables for kernel and user space. This effectively prevents side-channel information from kernel addresses to be leaked to user space, because the MMU uses a different page table hierarchy. Thus, while the processor is in user mode, the MMU will not be able to refer to any information about kernel virtual addresses, as shown in step ❹.

3.3 Challenges for Fine-grained Address Space Isolation

To enable LAZARUS to separate and isolate both execution domains a number of challenges have to be tackled: first, we must provide a mechanism for switching between kernel and user execution at any point in time without compromising the randomized kernel memory (**C1**). More specifically, while kernel and user space no longer share the randomized parts of privileged virtual memory, the system still has to be able to execute code pages in both execution modes. For this reason, we have to enable switching between kernel and user space. This is challenging, because such a transition can happen either through explicit invocation, such as a system call or an exception, or through hardware events, such as interrupts. As we will show our defense handles both cases securely and efficiently.

Second, we have to prevent the switching mechanism from leaking any side-channel information (**C2**). Unmapping kernel pages is also challenging with respect to side-channel information, i.e., unmapped memory pages still exhibit a timing difference compared to mapped pages. Hence, LAZARUS has to prevent information leakage through probing of unmapped pages.

Third, our approach has to minimize the overhead for running applications to offer a practical defense mechanism (**C3**). Implementing strict separation of address spaces efficiently is involved, since we only separate those parts of the address space that are privileged and randomized. We have to modify only those parts of the page table hierarchy which define translations for randomized addresses.

In the following we explain how our defense meets these challenges.

C1: Kernel-User Transitioning Processor resources are time-shared between processes and the operating system. Thus, the kernel eventually takes control over these resources, either through explicit invocation, or based on a signaling event. Examples for explicit kernel invocations are *system calls* and *exceptions*. These are synchronous events, meaning that the user process generating the event is suspended and waiting for the kernel code handling the event to finish.

On the one hand, after transitioning from user to kernel mode, the event handler code is no longer mapped in virtual memory because it is located in the kernel. Hence, we have to provide a mechanism to restore this mapping when entering kernel execution from user space.

On the other hand, when the system call or exception handler finishes and returns execution to the user space process, we have to erase those mappings again. Otherwise, paging entries might be shared between privilege levels. Since

all system calls enter the kernel through a well-defined hardware interface, we can activate and deactivate the corresponding entries by modifying this central entry point.

Transitions between kernel and user space execution can also happen through *interrupts*. A simple example for this type of event is the timer interrupt, which is programmed by the kernel to trigger periodically in fixed intervals. In contrast to system calls or exceptions, interrupts are asynchronously occurring events, which may suspend current kernel or user space execution at any point in time.

Hence, interrupt routines have to store the current process context before handling a pending interrupt. However, interrupts can also occur while the processor executes kernel code. Therefore, we have to distinguish between interrupts during user or kernel execution to only restore and erase the kernel entries upon transitions to and from user space respectively. For this we facilitate the stored state of the interrupted execution context that is saved by the interrupt handler to distinguish privileged from un-privileged contexts.

This enables LAZARUS to still utilize the paging caches for interrupts occuring during kernel execution.

C2: Protecting the Switching Mechanism The code performing the address space switching has to be mapped during user execution. Otherwise, implementing a switching mechanism in the kernel would not be possible, because the processor could never access the corresponding code pages. For this reason, it is necessary to prevent these mapped code pages from leaking any side-channel information. There are two possibilities for achieving this.

First, we can map the switching code with a different offset than the rest of the kernel code section. In this case an adversary would be able to disclose the offset of the switching code, while the actual randomization secret would remain protected.

Second, we can eliminate the timing channel by inserting dummy mappings into the unmapped region. This causes the surrounding addresses to exhibit an identical timing signature compared to the switching code.

Since an adversary would still be able to utilize the switching code to conduct a code-reuse attack in the first case, LAZARUS inserts dummy mappings into the user space page table hierarchy.

C3: Minimizing Performance Penalties Once paging is enabled on a processor, all memory accesses are mediated through the virtual memory subsystem. This means that a page walk is required for every memory access. Since traversing the page table results in high performance penalties, the MMU caches the most prominent address translations in the Translation Lookaside Buffers (TLBs).

LAZARUS removes kernel addresses from the page table hierarchy upon user space execution. Hence, the respective TLB entries need to be invalidated. As a result, subsequent accesses to kernel memory will be slower, once kernel execution is resumed.

To minimize these perfomance penalties, we have to reduce the amount of invalidated TLB entries to a minimum but still enforce a clear separation between

kernel and user space addresses. In particular, we only remove those virtual mappings, which fall into the location of a randomized kernel area, such as the kernel code segment.

4 Prototype Implementation

We implemented LAZARUS as a prototype for the Linux kernel, version 4.8 for the 64 bit variant of the x86 architecture. However, the techniques we used are generic and can be applied to all architectures employing multi-level page tables. Our patch consists of around 300 changes to seven files, where most of the code results from initialization. Hence, LAZARUS should be easily portable to other architectures. Next, we will explain our implementation details. It consists of the initialization setup, switching mechanism, and how we minimize performance impact.

4.1 Initialization

We first setup a second set of page tables, which can be used when execution switches to user space. These page tables must not include the randomized portions of the address space that belong to the kernel. However, switching between privileged and unprivileged execution requires some code in the kernel to be mapped upon transitions from user space. We explicitly create dedicated entry points mapped in the user page tables, which point to the required switching routines.

Fixed Mappings Additionally, there are kernel addresses, which are mapped to fixed locations in the top address space ranges. These *fixmap* entries essentially represent an address-based interface: even if the physical address is determined at boot time, their virtual address is fixed at compile time. Some of these addresses are mapped readable to user space, and we have to explicitly add these entries as well.

We setup this second set of page tables only once at boot time, before the first user process is started. Every process then switches to this set of page tables during user execution.

Dummy Mappings As explained in Section 3, one way of protecting the code pages of the switching mechanism is to insert dummy mappings into the user space page table hierarchy. In particular, we create mappings for randomly picked virtual kernel addresses to span the entire code section. We distribute these mappings in 2M intervals to cover all third-level page table entries, which are used to map the code section. Hence, the entire address range which potentially contains the randomized kernel code section will be mapped during user space execution using our randomly created dummy entries.

4.2 System Calls

There is a single entry point in the Linux kernel for system calls, which is called the system call handler. We add an assembly routine to execute immediately after execution enters the system call handler. It switches from the predefined user page tables to the kernel page tables and continues to dispatch the requested system call. We added a second assembly routine shortly before the return of the system call handler to remove the kernel page tables from the page table hierarchy of the process and insert our predefined user page tables.

However, contrary to its single entry, there are multiple exit points for the system call handler. For instance, there is a dedicated error path, and fast and slow paths for regular execution. We instrument all of these exit points to ensure that the kernel page tables are not used during user execution.

4.3 Interrupts

Just like the system call handler, we need to modify the interrupt handler to restore the kernel page tables. However, unlike system calls, interrupts can occur when the processor is in privileged execution mode as well. Thus, to handle interrupts, we need to distinguish both cases. Basically we could look up the current privilege level easily by querying a register. However, this approach provides information about the current execution context, whereas to distinguish the two cases we require the privilege level of the interrupted context.

Fortunately, the processor saves some hardware context information, such as the instruction pointer, stack pointer, and the code segment register before invoking the interrupt handler routine. This means that we can utilize the stored privilege level associated with the previous code segment selector to test the privilege level of the interrupted execution context. We then only restore the kernel page tables if it was a user context.

We still have to handle one exceptional case however: the non-maskable interrupt (NMI). Because NMIs are never maskable, they are handled by a dedicated interrupt handler. Hence, we modify this dedicated NMI handler in the kernel to include our mechanism as well.

4.4 Fine-grained Page Table Switching

As a software-only defense technique, one of the main goals of LAZARUS is to offer practical performance. While separating the entire page table hierarchy between kernel and user mode is tempting, this approach is impractical.

In particular, switching the entire page table hierarchy invalidates all of the cached TLB entries. This means, that the caches are reset every time and can never be utilized after a context switch. For this reason, we only replace those parts of the page table hierarchy, which define virtual memory mappings for randomized addresses. In the case of KASLR, this corresponds to the code section of the kernel. More specifically, the kernel code section is managed by the last of the 512 level 4 entries.

Thus, we replace only this entry during a context switch between privileged and unprivileged execution. As a result, the caches can still be shared between different privilege levels for non-randomized addresses. As we will discuss in Section 5, this does not impact our security guarantees in any way.

5 Evaluation

In this section we evaluate our prototypical implementation for the Linux kernel. First, we show that LAZARUS successfully prevents all of the previously published side-channel attacks. Second, we demonstrate that our defense only incurs negligible performance impact for standard computational workloads.

5.1 Security

Our main goal is to prevent the leakage of the randomization secret in the kernel to an unprivileged process through paging-based side-channel attacks. For this, we separate the page tables for privileged parts of the address space from the unprivileged parts. We ensure that this separation is enforced for randomized addresses to achieve practical performance.

Because all paging-based exploits rely on the timing difference between cached and uncached entries for privileged virtual addresses, we first conduct a series of timing experiments to measure the remaining side channel in the presence of LAZARUS.

In a second step, we execute all previously presented side-channel attacks on a system hardened with LAZARUS to verify the effectiveness of our approach.

Effect of LAZARUS on the timing side-channel To estimate the remaining timing side-channel information we measure the timing difference for privileged virtual addresses. We access each page in the kernel code section at least once and measure the timing using the `rdtscp` instruction. By probing the privileged address space in this way, we collect a timing series of execution cycles for each kernel code page. The results are shown in Figure 3. [3]

The timing side channel is clearly visible for the vanilla KASLR implementation: the start of the actual code section mapping is located around the first visible jump from 160 cycles up to 180 cycles. Given a reference timing for a corresponding kernel image, the attacker can easily calculate the random offset by subtracting the address of the peak from the address in the reference timing.

In contrast to this, the timing of LAZARUS shows a straight line, with a maximum cycle distance of two cycles. In particular, there is no correlation between any addresses and peaks in the timing signal of the hardened kernel. This indicates that our defense approach indeed closes the paging-induced timing

[3] For brevity, we display the addresses on the x-axis as offsets to the start of the code section (i.e., `0xffffffff80000000`). We further corrected the addresses by their random offset, so that both data series can be shown on top of each other.

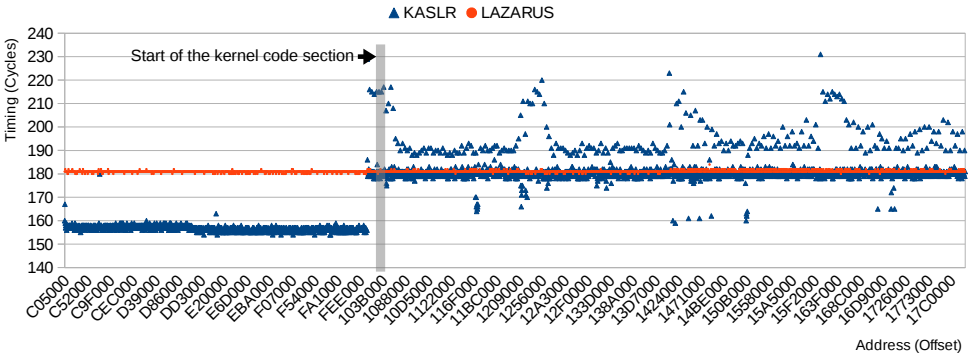

Fig. 3. Timing side-channel measurements.

channel successfully. We note, that the average number of cycles depicted for LAZARUS are also in line with the timings for cached page table entries reported by related work [8, 13]. To further evaluate the security of our approach, we additionally test it against all previous side-channel attacks.

Real-world side-channel attacks We implemented and ran all of the previous side-channel attacks against a system hardened with LAZARUS, to experimentally assess the effectiveness of our approach against real-world attacks.

Page-fault handler The first real-world side-channel attack against KASLR was published by Hund et al. [10]. They noted that the execution time of the page fault handler in the OS kernel depends on the state of the paging caches. More specifically, they access kernel addresses from user space which results in a page fault. While this would usually terminate the process causing the access violation, the POSIX standard allows for processes to handle such events via *signals*. By installing a signal handler for the segmentation violation (`SIGSEGV`), the user process can recover from the fault and measure the timing difference from the initial memory access to the delivery of the signal back to user space. In this way, the entire virtual kernel code section can be scanned and each address associated with its corresponding timing measurement, allowing a user space process to reconstruct the start address of the kernel code section. We implemented and successfully tested the attack against a vanilla Linux kernel with KASLR. In particular, we found that page fault handler exhibits a timing difference of around 30 cycles for mapped and unmapped pages, with an average time of around 2200 cycles. While this represents a rather small difference compared to the other attacks, this is due to the high amount of noise that is caused by the execution path of the page fault handler code in the kernel. [4] When we applied LAZARUS to the kernel the attack no longer succeeded.

[4] This was also noted in the original exploit [10].

Prefetch Recently, the `prefetch` instruction featured on many Intel x86 processors was shown to enable side-channel attacks against KASLR [8]. It is intended to provide a benign way of instrumenting the caches: the programmer (or the compiler) can use the instruction to provide a hint to the processor to cache a given virtual address.

Although there is no guarantee that this hint will influence the caches in any way, the instruction can be used with arbitrary addresses in principle. This means that a user mode program can prefetch a kernel virtual address, and execution of the instruction will fail siltently, i.e., the page fault handler in the kernel will not be executed, and no exception will be raised.

However, the MMU still has to perform a privilege check on the provided virtual address, hence the execution time of the `prefetch` instruction depends directly on the state of the TLBs.

We implemented the prefetch attack against KASLR for Linux, and successfully executed it against a vanilla system to disclose the random offset. Executing the attack against a system hardened with LAZARUS we found the attack to be unsuccessful.

TSX Rafal Wojtczuk originally proposed an attack to bypass KASLR using the Transactional Synchronization Extension (TSX) present in Intel x86 CPUs [23], and the attack gained popularity in the academic community through a paper by Jang et al. [13]. TSX provides a hardware mechanism that aims to simplify the implementation of multi-threaded applications through lock elision. Initially released in Haswell processors, TSX-enabled processors are capable of dynamically determining to serialize threads through lock-protected critical sections if necessary. The processor may abort a TSX transaction if an *atomic* view from the software's perspective is not guaranteed, e.g., due to conflicting accesses between two logical processors on one core.

TSX will suppress any faults that must be exposed to software if they occur within a transactional region. Memory accesses that cause a page walk may abort a transaction, and according to the specification *will not be made architecturally visible through the behavior of structures such as TLBs* [12]. The timing characteristics of the abort, however, can be exploited to reveal the current state of the TLBs. By causing a page walk inside a transactional block, timing information on the aborted transaction discloses the position of kernel pages that are mapped into a process: first, the attacker initiates a memory access to kernel pages inside a transactional block, which causes (1) a page walk, and (2) a segmentation fault. Since TSX masks the segmentation fault in hardware, the kernel is never made aware of the event and the CPU executes the abort handler provided by the attacker-controlled application that initiated the malicious transaction. Second, the attacker records timing information about the abort-handler execution. A transaction abort takes about 175 cycles if the probed page is mapped, whereas it aborts in about 200 cycles or more if unmapped [23]. By probing all possible locations for the start of the kernel code section, this side channel exposes the KASLR offset to the unprivileged attacker in user space.

Probing pages in this way under LAZARUS reveals no information, since we unmap all kernel code pages from the process, rendering the timing side channel useless as any probes to kernel addresses show as unmapped. Only the switching code and the surrounding dummy entries are mapped. However, these show identical timing information, and hence, are indistinguishable for an adversary.

5.2 Performance

We evaluated LAZARUS on a machine with an Intel Core i7-6820HQ CPU clocked at 2.70GHz and 16GB of memory. The machine runs a current release of Arch Linux with kernel version 4.8.14. For our testing, we enabled KASLR in the Linux kernel that Arch Linux ships. We also compiled a secondary kernel with the same configuration and LAZARUS applied.

We first examine the performance overhead with respect to the industry standard SPEC2006 benchmark [9]. We ran both the integer and floating point benchmarks in our test platform under the stock kernel with KASLR enabled. We collected these results and performed the test again under the LAZARUS kernel. Our results are shown in Figure 4.

The observed performance overhead can be attributed to measurement inaccuracies. Our computed worst case overhead is of 0.943%. We should note that SPEC2006 is meant to test computational workloads and performs little in terms of context switching.

To better gauge the effects of LAZARUS on the system, we ran the system benchmarks provided by LMBench3 [22]. LMBench3 improves on the context switching benchmarks by eliminating some of the issues present in previous versions of the benchmark, albeit it still suffers issues with multiprocessor machines. For this reason, we disabled SMP during our testing. Our results are presented in Figure 5.

We can see how a system call intensive application is affected the most under LAZARUS. This is to be expected, as the page tables belonging to the kernel must be remapped upon entering kernel execution. In general, we show a 47% performance overhead when running these benchmarks. We would like to remind the reader, however, that these benchmarks are meant to stress test the performance of the operating system when handling interrupts and do not reflect normal system operation.

In order to get a more realistic estimate of LAZARUS, we ran the Phoronix Test Suite [15], which is widely used to compare the performance of operating systems. The Phoronix benchmarking suite features a large number of tests which cover different aspects of a system, and are grouped according to the targeted subsystem of the machine. Specifically, we ran the system and disk benchmarks to test application performance. Our results are shown in Figure 6. We show an average performance overhead of 2.1% on this benchmark, which is in line with our results provided by the SPEC and LMBench benchmarks. The worst performers were benchmarks that are bound to read operations. We speculate that this is due to the amount of context switches that happen while the read

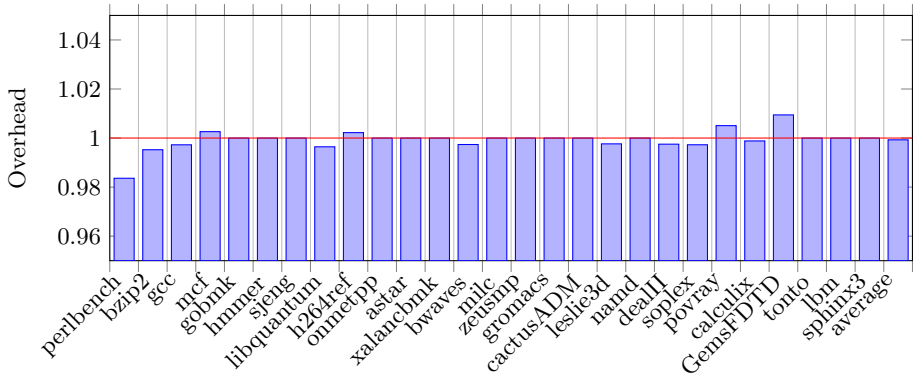

Fig. 4. SPEC2006 Benchmark Results

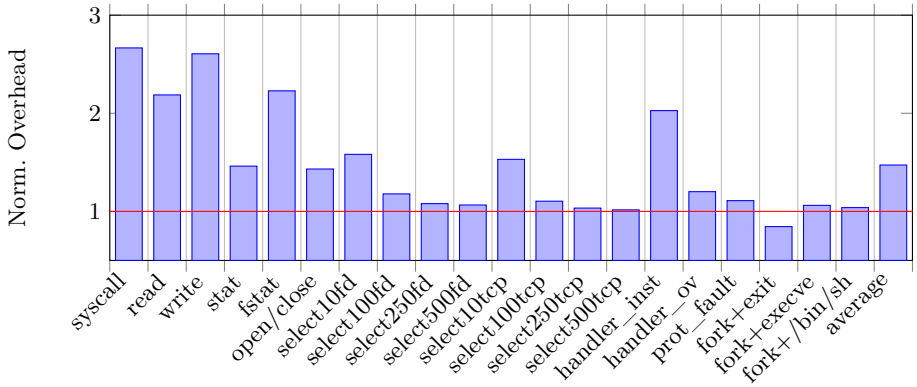

Fig. 5. LMBench3 Benchmark Results

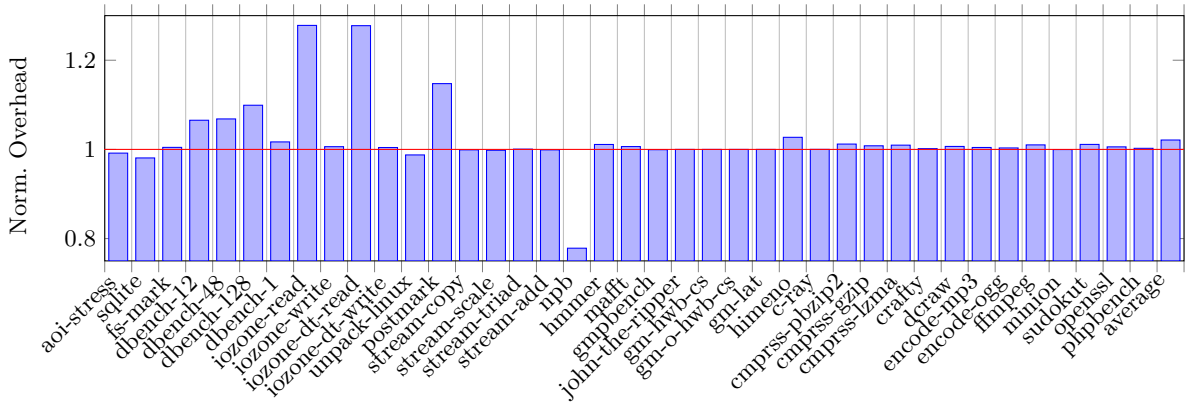

Fig. 6. Phoronix Benchmark Suite

operation is taking place, as a buffer in kernel memory needs to be copied into a buffer from user space or remapped there.

Lastly, we ran the `pgbench` benchmark on a test PostgreSQL database and measured a performance overhead of 2.386%.

6 Discussion

6.1 Applying LAZARUS to different KASLR implementations

Relocation of kernel code is an example of how randomization approaches can be used as a defense building block which is implemented by practically all real-world operating systems [2, 11, 14]. While a kernel employing control-flow integrity (CFI) [1, 3, 21] does not gain security benefit from randomizing the code section, it might still randomize the memory layout of other kernel memory regions: for instance, it can be applied to the module section, to hide the start address of the code of dynamically loadable kernel modules. Further, randomization was recently proposed as a means to protect the page tables against malicious modification through data-only attacks [5].

Since all of the publicly available attacks focus on disclosing the random offset of the kernel code section, we implemented our proof of concept for KASLR as well. Nonetheless, we note that LAZARUS is not limited to hardening kernel code randomization, but can be applied to other randomization implementations as well. In contrast to the case of protecting KASLR, our defense does not require any special treatment for hiding the low-level switching code if applied to other memory regions.

6.2 Other side-channel attacks on KASLR

As explained in Section 2, almost all previously presented side-channel attacks on KASLR exploit the paging subsystem. LAZARUS isolates kernel virtual memory from user processes by separating their page tables. However, Evtyushkin et al. [6] recently presented the branch target buffer (BTB) side-channel attack, which does not exploit the paging subsystem for virtual kernel addresses.

In particular, they demonstrated how to exploit collisions between branch targets for user and kernel addresses. The attack works by constructing a malicious chain of branch targets in user space, to fill up the BTB, and then executing a previously chosen kernel code path. This evicts branch targets previously executed in kernel mode from the BTB, thus their subsequent execution will take longer.

While the BTB attack was shown to bypass KASLR on Linux, it differs from the paging-based side channels by making a series of assumptions: 1) the BTB has a limited capacity of 10 bits, hence it requires KASLR implementations to deploy a low amount of entropy in order to succeed. 2) it requires the attacker to craft a chain of branch targets, which cause kernel addresses to be evicted from the BTB. For this an adversary needs to reverse engineer the hashing algorithm

used to index the BTB. These hashing algorithms are different for every micro architecture, which limits the potential set of targets. 3) the result of the attack can be ambiguous, because any change in the execution path directly effects the BTB contents.

There are multiple ways of mitigating the BTB side-channel attack against KASLR. A straightforward approach is to increase the amount of entropy for KASLR, as noted by Evtyushkin et al. [6]. A more general approach would be to introduce a separation between privileged an unprivileged addresses in the BTB. This could be achieved by offering a dedicated flush operation, however this requires changes to the hardware. Alternatively, this flush operation can emulated in software, if the hashing algorithm used for indexing the BTB has been reverse engineered. We implemented this approach against the BTB attack by calling a function which performs a series of jump instructions along with our page tables switching routine and were unable to recover the correct randomization offset through the BTB attack in our tests.

7 Related Work

In this section we discuss software and hardware mitigations against side-channel attacks that were proposed, and compare them to our approach.

7.1 Hardware Mitigations

Privilege Level Isolation in the Caches Eliminating the paging side channel is also possible by modifying the underlying hardware cache implementation. This was first noted by Hund et al. [10]. However, modern architectures organize caches to be optimized for performance. Additionally, changes to the hardware are very costly, and it takes many years to widely deploy these new systems. Hence, it is unlikely that such a change will be implemented, and even if it is, existing production systems will remain vulnerable for a long time. Our software-only mitigation can be deployed instantly by patching the kernel.

Disabling Detailed Timing for Unprivileged Users All previously presented paging side-channel attacks rely on detailed timing functionality, which is provided to unprivileged users by default. For this reason, Hund et al. [10] suggested to disable the `rdtsc` instruction for user mode processes. While this can be done from software, it effectively changes the ABI of the machine. Since modern platforms offer support for a large body of legacy software, implementing such a change would introduce problems for many real-world user applications. As we demonstrate in our extensive evaluation, LAZARUS is transparent to user-level programs and does not disrupt the usual workflow of legacy software.

7.2 Software Mitigations

Separating Address Spaces Unmapping the kernel page tables during user-land execution is a natural way of separating their respective address spaces, as suggested in [8, 13]. However, Jang et al. [13] considered the approach impractical,

due to the expected performance degradation. Gruss et al. [8] estimated the performance impact of reloading the entire page table hierarchy up to 5%, by reloading the top level of the page table hierarchy (via the `CR3` register) during a context switch, but did not provide any implementation or detailed evaluation of their estimated approach. Reloading the top level of the page tables results in a higher performance overhead, because it requires the processor to flush all of the cached entries. Address space separation has been implemented by Apple for their iOS platform [16]. Because the ARM platform supports multiple sets of page table hierarchies, the implementation is straightforward on mobile devices. For the first time we provide an improved and highly practical method of implementing address space separation on the x86 platform.

Increasing KASLR Entropy Some of the presented side-channel attacks benefit from the fact that the KASLR implementation in the Linux kernel suffers from a relatively low entropy [6, 10]. Thus, increasing the amount of entropy represent a way of mitigating those attacks in practice. While this approach was suggested by Hund et al. [10] and Evtyushkin et al. [6], it does not eliminate the side channel. Additionally, the mitigating effect is limited to attacks which exploit low entropy randomization. In contrast, LAZARUS mitigates all previously presented paging side-channel attacks.

Modifying the Page Fault Handler Hund et al. [10] exploited the timing difference through invoking the page fault handler. They suggested to enforce its execution time to an equal timing for all kernel addresses through software. However, this approach is ineffective against attacks which do not invoke the kernel [8, 13]. Our mitigation reorganizes the cache layout in software to successfully stop the attacks, that exploit hardware features to leak side channel information, even for attacks that do not rely on the execution time of any software.

KAISER Concurrently to our work Gruss et al. implemented strong address-space separation [7]. Their performance numbers are in line with our own measurements, confirming that separating the address spaces of kernel and userland constitutes a practical defense against paging-based side-channel attacks. In contrast to LAZARUS, their approach does not make use of dummy mappings to hide the switching code, but separates it from the rest of the kernel code section (as outlined in 3.3.C2).

8 Conclusion

Randomization has become a vital part of the security architecture of modern operating systems. Side-channel attacks threaten to bypass randomization-based defenses deployed in the kernel by disclosing the randomization secret from unprivileged user processes. Since these attacks exploit micro-architectural implementation details of the underlying hardware, closing this side channel through a software-only mitigation efficiently is challenging. However, all of these attacks

rely on the fact that kernel and user virtual memory reside in a shared address space. With LAZARUS, we present a defense to mitigate previously presented side-channel attacks purely in software. Our approach shows that side-channel information exposed through shared hardware resources can be hidden by separating the page table entries for randomized privileged addresses from entries for unprivileged addresses in software. LAZARUS is a necessary and highly practical extension to harden kernel-space randomization against side-channel attacks.

Acknowledgment

This work was supported in part by the German Science Foundation (project S2, CRC 1119 CROSSING), the European Union's Seventh Framework Programme (609611, PRACTICE), and the German Federal Ministry of Education and Research within CRISP.

Dean Sullivan, Orlando Arias, and Yier Jin are partially supported by the Department of Energy through the Early Career Award (DE-SC0016180). Mr. Orlando Arias is also supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. 1144246.

Bibliography

[1] Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Transactions on Information System Security 13 (2009)

[2] Cook, K.: Kernel address space layout randomization. `http://selinuxproject.org/~jmorris/lss2013_slides/cook_kaslr.pdf` (2013)

[3] Criswell, J., Dautenhahn, N., Adve, V.: Kcofi: Complete control-flow integrity for commodity operating system kernels. In: 35th IEEE Symposium on Security and Privacy. S&P (2014)

[4] CVEDetails: CVE-2016-4557. `http://www.cvedetails.com/cve/cve-2016-4557` (2016)

[5] Davi, L., Gens, D., Liebchen, C., Ahmad-Reza, S.: PT-Rand: Practical mitigation of data-only attacks against page tables. In: 24th Annual Network and Distributed System Security Symposium. NDSS (2017)

[6] Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: Jump over aslr: Attacking branch predictors to bypass aslr. In: IEEE/ACM International Symposium on Microarchitecture (MICRO) (2016)

[7] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., Mangard, S.: Kaslr is dead: Long live kaslr. In: International Symposium on Engineering Secure Software and Systems. ESSoS (2017)

[8] Gruss, D., Maurice, C., Fogh, A., Lipp, M., Mangard, S.: Prefetch side-channel attacks: Bypassing smap and kernel aslr. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 368–379. ACM (2016)

[9] Henning, J.L.: Spec cpu2006 benchmark descriptions. SIGARCH Comput. Archit. News 34(4), 1–17 (Sep 2006), http://doi.acm.org/10.1145/1186736.1186737
[10] Hund, R., Willems, C., Holz, T.: Practical timing side channel attacks against kernel space ASLR. In: 34th IEEE Symposium on Security and Privacy. S&P (2013)
[11] Inc., A.: Os x mountain lion core technologies overview. http://movies.apple.com/media/us/osx/2012/docs/OSX_MountainLion_Core_Technologies_Overview.pdf (2012)
[12] Intel: Intel 64 and IA-32 architectures software developer's manual. http://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (2017)
[13] Jang, Y., Lee, S., Kim, T.: Breaking kernel address space layout randomization with intel TSX. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 380–392. ACM (2016)
[14] Johnson, K., Miller, M.: Exploit mitigation improvements in windows 8. https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf (2012)
[15] Larabel, M., Tippett, M.: Phoronix test suite. h ttp://www. phoronix-test-suite. com (2011)
[16] Mandt, T.: Attacking the ios kernel: A look at "evasi0n". http://www.nislab.no/content/download/38610/481190/file/NISlecture201303.pdf (2013)
[17] MITRE: CVE-2015-1328. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1328 (2015)
[18] MITRE: CVE-2016-0728. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-0728 (2016)
[19] MITRE: CVE-2016-5195. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195 (2016)
[20] Molinyawe, M., Hariri, A.A., Spelman, J.: $hell on earth: From browser to system compromise. In: Blackhat USA. BH US (2016)
[21] PaX Team: RAP: RIP ROP (2015)
[22] Staelin, C.: lmbench: an extensible micro-benchmark suite. Software-Practice and Experience 35(11), 1079 (2005)
[23] Wojtczuk, R.: Tsx improves timing attacks against kaslr. https://labs.bromium.com/2014/10/27/tsx-improves-timing-attacks-against-kaslr/ (2014)