PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables

Lucas Davi,* David Gens,† Christopher Liebchen,† Ahmad-Reza Sadeghi†
*University of Duisburg-Essen, Germany. lucas.davi@uni-due.de
†CYSEC/Technische Universität Darmstadt, Germany.
{david.gens,christopher.liebchen,ahmad.sadeghi}@trust.tu-darmstadt.de

***Abstract*—Kernel exploits constitute a powerful attack class allowing attackers to gain full control over a system. Various kernel hardening solutions have been proposed or deployed in practice to protect the kernel against code injection (e.g., DEP) or code-reuse exploits (e.g., CFI). However, the security of all these hardening techniques relies heavily on the assumption that kernel page tables cannot be manipulated, e.g., by means of data-only attacks. Ensuring kernel page tables integrity is not only essential for kernel security but also a challenging task in practice since existing solutions require hardware trust anchors, costly hypervisors, or inefficient integrity checks.**

In this paper, we first motivate the importance of protecting kernel page tables by presenting a data-only attack against page tables to bypass the recently released CFI-based (Linux) kernel hardening technique RAP. Thereafter, we present the design and implementation of PT-Rand, the first practical solution to protect kernel page tables that does not suffer from the mentioned deficiencies of previous proposals. PT-Rand randomizes the location of page tables and tackles a number of challenges to ensure that the location of page tables is not leaked. This effectively prevents the attacker from manipulating access permissions of code pages, thereby enabling secure enforcement of kernel exploit mitigation technologies such as CFI. We extensively evaluate our prototype implementation of PT-Rand for the current Linux kernel on the popular Linux distribution Debian and report a low overhead of 0.22% for common benchmarks. Moreover, we combine RAP with PT-Rand to protect RAP against data-only attacks on kernel page tables.

I. Introduction

Operating system kernels are essential components in modern computing platforms since they provide the interface between user applications and hardware. They also feature many important services such as memory and disk management. Typically, the kernel is separated from user applications by means of memory protection, i.e., less-privileged user applications can only access the higher-privileged kernel through well-defined interfaces, such as system calls. Attacks against kernels are gaining more and more prominence for two reasons: first, the kernel executes with high privileges, often allowing the attacker to compromise the entire system based on a single kernel exploit. Second, the kernel implements a major part of the security subsystem. Hence, to escalate execution privileges to *root* or escape from application sandboxes in browsers, it is often inevitable to compromise the kernel. Kernel exploits are leveraged in (i) all of the latest iOS jailbreaks, (ii) browser sandbox exploits against Chrome [37], and (iii) large-scale attacks by nation-state adversaries to obtain full control over the targeted system, as in the infamous case of Stuxnet [42].

Typical means for program code exploitation are *memory corruption* vulnerabilities. They allow attackers to alter control and data structures in memory to execute (injected) malicious code, or to launch code-reuse attacks using techniques such as return-oriented programming [25, 46]. One of the main reasons for the prevalence of memory corruption vulnerabilities is that a vast amount of software is programmed in unsafe languages such as C and C++. In particular, kernel code is typically completely written in these languages for better performance, legacy reasons, and hardware-close programming. The monolithic design of the commodity kernels and numerous device drivers increase the attack surface compared to user-mode applications. For instance, over the last 17 years 1526 vulnerabilities have been documented in the Linux kernel [14].

Various solutions have been proposed or deployed in practice to protect software systems against code-injection or code-reuse exploits: modern kernel hardening solutions like *Supervisor Mode Execution Protection* (SMEP) and *Supervisor Mode Access Protection* (SMAP) [27] prevent access to user-mode memory while the CPU executes code in kernel mode [3, 27]. This prevents the attacker from executing code with kernel privileges in user mode. The deployment of W$\oplus$X (Writable $\oplus$ Executable) prevents the adversary from executing code in the data memory. Indeed, W$\oplus$X has dramatically reduced the threat of code-injection attacks. However, attackers have already eluded to more sophisticated exploitation techniques such as code reuse to bypass these measures and to hijack the control flow of the targeted code. Mitigating *control-flow hijacking* attacks is currently a hot topic of research [51]. The most promising and effective defenses at the time of writing are control-flow integrity (CFI) [1], fine-grained code randomization [31], and code-pointer integrity (CPI) [30]. However, all defenses against control-flow hijacking are based on the following assumptions: firstly, they assume that code pages cannot be manipulated. Otherwise, the adversary can replace existing code with malicious code or overwrite CFI/CPI checks. Secondly, they assume that critical data structures containing code pointers (e.g., the shadow stack for CFI, the safe region for CPI) are isolated. Otherwise, the adversary can manipulate them by overwriting code pointers.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
 ISBN 1-891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23421

However, as observed by Ge et al. [19], defenses against control-flow hijacking in the kernel additionally require the protection of *page tables* against *data-only attacks*. Otherwise the assumptions mentioned above will not hold and these defenses can simply be bypassed by manipulating the page tables.

Data-only attacks do not change the control flow of the program. Instead they direct the control flow to certain nodes within the control-flow graph (CFG) of the underlying program by altering the input data. Hence, the executed path in the CFG is indistinguishable from any other benign execution. Page tables are data structures that map virtual addresses to physical addresses. They define *read-write-execute* permissions for code and data memory pages, where a page is simply a contiguous 4KB memory area. Hence, attackers can launch data-only attacks (based on memory corruption vulnerabilities in the kernel) to alter page tables, and consequently disable memory protection, manipulate code pages, and inject malicious code [38]. Recently industry researchers have presented several page-table based attacks [16] stressing that these attacks are possible because the attacker can easily determine the location of the page tables.

To tackle data-only attacks on page tables, previous work suggested kernel instrumentation to mediate any access to memory-management structures according to a security policy [4, 5, 13, 20, 43, 49]. However, as we elaborate on related work in Section VIII, all these solutions suffer from at least one of the following shortcomings: high performance overhead, require additional and higher privileged execution modes (e.g., hypervisors), or depend on architecture-specific hardware features. Recently, Microsoft released a patch for Windows 10 [28] that randomizes the base address used to calculate the virtual address of page table entries. However, this patch does not protect against crucial information disclosure attacks that have been frequently shown to circumvent any (even fine-grained) randomization scheme [15, 48].

Goal and Contributions. In this paper, we present the design and implementation of a novel memory protection scheme, PT-Rand, that prevents the attacker from manipulating page tables. We highlight the importance of page table protection by implementing a real-world exploit, based on a vulnerable kernel driver (CVE-2013-2595), to directly manipulate the code of a kernel function. Using this attack, we circumvent a recently released CFI kernel hardening scheme, Linux RAP [52], and execute arbitrary code with kernel privileges. In summary, our contributions are as follows:

Page Table Protection: We present a practical and effective protection of page tables against *data-only attacks* without requiring additional hardware or a hypervisor. Rather than applying expensive policy enforcement checks, we randomize page tables when they are allocated and ensure that no information related to the location of page tables is leaked. To achieve this, we need to tackle several challenges. (1) There are many data pointers that the attacker can exploit to locate page tables. (2) The physical memory (including page tables) is usually mapped 1:1 into the virtual address space. Hence, the attacker can easily locate and access this section. (3) The kernel still needs to efficiently access page tables, and distinguish between randomized and regular memory pages. As we will show in Section V, PT-Rand tackles all these challenges, while remaining compatible to existing software, like kernel drivers.

Prototype Implementation: We provide a fully working prototype implementation for a recent Linux kernel (v4.6). We also combine Linux kernel CFI protection (RAP) with PT-Rand to protect RAP against data-only attacks on page tables.

Performance Evaluation: We provide an extensive security and performance evaluation. In particular, we show that the attacker cannot bypass the randomization by means of guessing attacks. Our performance measurements for popular benchmarking suites SPEC CPU2006, LMBench, Phoronix, and Chromium browser benchmarks show that PT-Rand incurs almost no measurable overhead (0.22% on average for SPEC), successfully applies to many complex, modern system configurations, and is highly practical as it supports a variety of applications and kernel code.

PT-Rand effectively enables memory protection and paves the way for secure deployment of defenses to thwart code-reuse attacks on the kernel.

II. Background: Memory Protection and Paging

In this section, we recall the basic principles of memory protection and paging that are needed for the understanding of the following sections.

Memory protection ensures that (i) privileged kernel code is isolated from less-privileged user code, (ii) one process cannot access the memory space of another process, and (iii) read-only data memory cannot be tampered with by unauthorized write operations. To enforce memory protection, modern operating systems leverage a widely-deployed CPU feature called *paging*. Although the implementation details vary among different architectures, the basic principles are the same. Hence, without loss of generality, we focus our discussion on paging for the contemporary x86_64 architecture.

Paging creates an indirection layer to access *physical memory*. Once enabled, the CPU will only operate on *virtual memory* (VM), i.e., it can no longer access physical memory. The advantage of paging is that processes start working with large contiguous memory areas. However, physically, the memory areas are scattered throughout the RAM, or swapped out on hard disk. As a consequence, each access to a virtual memory address needs to be translated to a physical address. This is achieved by a dedicated hardware engine called Memory Management Unit (MMU). The translation is performed by means of *page tables* that operate at the granularity of pages, where a typical page size is 4KB. Specifically, the operating system stores mapping information from virtual to physical addresses into these page tables thereby enabling efficient translation. To isolate processes from each other, the kernel assigns each process to its own set of page tables. In addition, page tables maintain *read-write-execute* permissions for each memory page. These permissions are enforced at the time of translation, e.g., allowing the operating system to prevent write operations to code pages or executing data pages.

Figure 1 provides high-level insights into the translation process. First, the memory subsystem of the CPU receives the

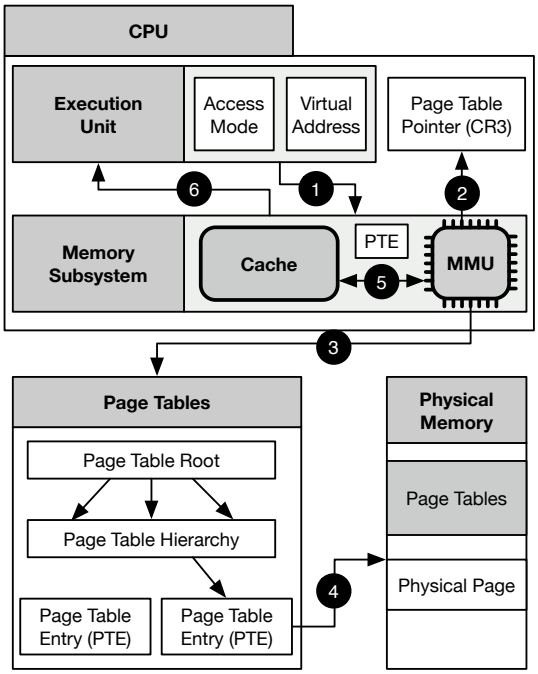

Figure 1: Paging - translation of virtual addresses to physical addresses.

access mode and a virtual memory address from the execution unit as input ❶. To access the page tables, the MMU reads out the pointer to the page table root which is always stored in the third control register (`CR3`) on x86_64 ❷. This pointer is already a physical memory address pointing to the root of the page table hierarchy ❸. That said, page tables are organized in a tree-like hierarchy for space optimization reasons. The MMU traverses the page table hierarchy until it reaches the page table entry (PTE) which contains the physical address for the given virtual memory address ❹. In addition, the PTE holds the access permissions and ownership (user or kernel) of the associated memory page. The memory subsystem leverages this information to validate whether the target operation (read, write, or execute) adheres to the permission set and ownership of the page. If validation is successful, the translation information is used to fetch the data from the physical memory slot and stored into the cache ❺. Note that the cache internally consists of a data and a instruction cache. For read and write operations the fetched data is stored into the data cache. In contrast, execute requests lead to a write of the fetched data to the instruction cache. Finally, the fetched data is forwarded to the execution unit of the CPU ❻. If the MMU either does not find a valid mapping in the page table hierarchy or observes an unauthorized access in ❹, the memory subsystem generates an exception ❻.

It is important to note that the page tables only contain physical addresses. This becomes a performance bottleneck when the kernel aims at changing the page permissions. As the kernel operates on virtual addresses, all the physical addresses belonging to a page would need to be mapped to virtual addresses dynamically before the permission update can be performed. To tackle this bottleneck, the kernel maintains a so-called *1:1 mapping* which permanently maps the whole physical memory to a fixed address into the virtual memory. To quickly translate a physical to a virtual address, the kernel adds the physical address to the start address of the 1:1 mapping, and can then use the resulting virtual address to access the memory.

III. On the Necessity of Page Tables Protection

In the adversary setting of kernel exploits the attacker has full control over the user mode, and hence, can execute arbitrary code with user-mode privileges, and interact with the kernel through system calls and driver APIs. The attacker's goal is to gain higher privilege level to be able to execute arbitrary code with kernel-mode privileges. To do so, the attacker needs to hijack a control-flow path of kernel code by overwriting a kernel code pointer, e.g., a return address or function pointer, using a memory-corruption vulnerability that is exposed either through the kernel itself or one of the loaded drivers.

In the following, we briefly provide an overview of the main kernel-related exploitation techniques as well as the defenses that are deployed or proposed against these attacks. To mitigate kernel code-injection and kernel code-reuse attacks, the kernel must be hardened with a variety of protection measures such as W⊕X and Control-Flow Integrity (CFI), fine-grained randomization or Code-Pointer Integrity (CPI). However, as we elaborate in the following the security of all these defenses relies on the integrity of page tables that can be attacked by means of data-only attacks – We show this using a real-world exploit that manipulates page tables against a kernel CFI protection.

Traditional Kernel Attacks. To escalate the attacker's privileges to kernel privileges, a common exploitation technique is as follows: first, the attacker allocates a new buffer in memory, writes malicious code into this buffer, and sets the memory page on which the buffer is located to executable. The latter can be achieved by common user space library functions such as *mprotect()* on Linux and *VirtualProtect()* on Windows. Recall that these actions are possible because the attacker has already gained control over the user space. Second, the attacker overwrites a kernel code pointer with the start address of the malicious code based on a memory corruption vulnerability inside the kernel. These vulnerabilities are typically triggered by abusing the kernel's interfaces such as system calls and driver APIs. Third, the attacker triggers the execution of a function that executes a branch on the corrupted kernel code pointer. As a result, the kernel's internal control flow will be dispatched to the previously injected, malicious code. Although this code resides in user space, it will be executed with kernel privileges because the control-flow hijacking occurred in the kernel mode. In a similar vein, the attacker can launch code-reuse attacks using the return-oriented programming (ROP) [46] technique. These attacks combine and chain short instruction sequences (called gadgets) that end in an indirect branch instruction. They are typically leveraged if the attacker cannot allocate new malicious code on an executable page. Thus, the user-mode buffer will hold a ROP payload consisting of code pointers to gadgets. Upon corruption of the kernel pointer, the ROP gadget chain will be executed under kernel privileges [18].

Code-injection and Code-reuse Attacks. Modern CPUs feature hardware extensions *Supervisor Mode Execution Protection* (SMEP) and *Supervisor Mode Access Protection* (SMAP) that prevent access to user-mode memory while the CPU executes code in the kernel mode [3, 27]. Alternatively, if these extensions are not present, the kernel can simply unmap the entire user space memory when kernel code is executed [34]. Such protections force the attacker to directly inject malicious code or the ROP payload into the kernel's memory space which is a challenging task since the attacker cannot directly write into kernel memory. However, several kernel functions accept and process user-mode buffers. A prominent example is the *msgsnd()* system call which allows exchange of messages. The attacker can exploit this function to cause the kernel to copy the user-mode exploit buffer (the message) into kernel memory. By leveraging a memory disclosure attack inside the kernel, the attacker can determine the address where the buffer is located in kernel memory and launch the exploit thereafter [40]. Several techniques are deployed or proposed to harden the kernel against these attacks: W$\oplus$X (Writable $\oplus$ Executable) is leveraged by many modern operating systems to prevent code to be executed from data memory. *Fine-grained code randomization* diversifies the code address layout to complicate code-reuse attacks [31]. Many modern operating systems apply Kernel Address Space Layout Randomization (KASLR) [17, 34]. *Control-flow integrity* (CFI) mitigates control-flow hijacking attacks by validating that the application's control flow remains within a statically computed control-flow graph [1]. CFI has been also adapted to kernel code [13, 19]. Recently a CFI-based protection for Linux kernel (RAP [52]) has been released. *Code pointer integrity* (CPI) [30] prevents control-flow hijacking by ensuring the integrity of code pointers.

Principally all these defenses significantly raise the bar. However, as observed in [13, 19] these defenses heavily rely on the assumption that the instrumented code cannot be manipulated, i.e., the attacker cannot compromise integrity checks or exploit information leakage against randomization schemes, and replace existing code with malicious code. On the other hand, this assumption is easily undermined by data-only attacks that tamper with the page tables as we describe next.

Data-only Attacks against Page Tables. In contrast to control-flow hijacking attacks, data-only attacks abstain from compromising code pointers. For example, the attacker can overwrite the `is_admin` variable of an application at runtime [10]. Although no code pointer has been compromised, the attacker can now execute benign functionality with higher privileges. In the context of the kernel, data-only attacks allow code injection attacks by modifying page table entries (PTEs) which we explained in Section II. To initiate data-only attacks, the attacker first exploits a memory-corruption vulnerability in the kernel or a device driver to gain read and write access to kernel memory. Since kernel memory contains references to page tables, the attacker can carefully read those references and locate them [38]. In particular, the attacker can disclose the virtual address of a PTE corresponding to a page that encapsulates a kernel function which can be triggered from the user space. Next, the attacker modifies the page permissions to writable and executable. For instance, the entire code of the kernel function could be replaced with malicious code. Finally, the attacker triggers the kernel function from user space to execute the injected code with kernel privileges.

Generic Bypass of Kernel CFI. To demonstrate the potential of data-only attacks against page tables, we first hardened the current Linux kernel with the open source version of RAP [52]. RAP is a state-of-the-art CFI implementation that instruments the Linux kernel during compile-time to enforce fine-grained CFI at run-time. In particular, RAP ensures that the attacker cannot overwrite code pointers (used for indirect branches) with arbitrary values. This is achieved by emitting CFI checks before all indirect branches that validate whether the program flow targets a valid destination. However, as mentioned before, a fundamental assumption of RAP is the integrity of the kernel code. If code integrity is not ensured, the attacker can simply overwrite the CFI checks with `NOP` instructions or directly overwrite existing kernel code with malicious code.

We undermine this assumption by using a data-only attack to first modify the page tables and change the memory permission of the kernel code to writable. Next, we overwrite an existing system call with our attack payload which elevates the privileges of the current process to root. After successfully overwriting the kernel code, we invoke the modified system call from user mode to eventually obtain root access. The details of this exploit are described in Section VI-A. While the impact of the attack itself is not surprising (CFI does not aim to prevent code-injection attacks), it highlights the importance of having an effective protection against data-only attacks that target page tables. We note that this attack is not limited to RAP but can also be applied to randomization or isolation-based defenses (CPI) against code-reuse attacks.

Existing Kernel Page Tables Protections. As we discuss in detail in the related work Section VIII, the existing proposals for protecting kernel page tables suffer from various shortcomings: they either require a hardware trust anchor, or privileged software (e.g., hypervisor), or are costly due to integrity checks.

Summary. All known exploit mitigation schemes strongly depend on memory protection to prevent the attacker from injecting code or corrupting existing code. Even with these schemes in place, page tables managing memory permissions can be compromised through data-only attacks. Hence, designing a defense against data-only attacks is vital and complements the existing mitigation technologies allowing their secure deployment for kernel code.

IV. PT-RAND: DESIGN OF OUR PAGE TABLE PROTECTION

In this section, we present the adversarial model, explain the high-level design of our scheme, and elaborate on challenges for implementing practical and secure page table protection.

A. Adversary Model and Assumptions

The adversary setting for our protection scheme PT-Rand against the corruption of page tables is based on the following assumptions (which are along the lines of the assumptions of related literature):

- **Memory Corruption:** There exists a memory corruption vulnerability in either the kernel or a driver. The attacker can exploit this vulnerability to read and write arbitrary memory (e.g., [38]).
- **Controlling User Space:** The attacker has full control over the user space, and consequently can execute arbitrary code in user space and call kernel API functions.
- **User Space Access:** User-mode pages are not accessible when the CPU is in the kernel mode. This is enforced by modern CPU features such as SMAP/SMEP [3, 27] or by simply unmapping the user space during kernel code execution [34].
- **W⊕X:** Kernel code pages are not per-se writable. This is enforced by W⊕X protection inside the kernel. As a consequence, the attacker needs to resort to a **data-only attack** to manipulate code page permissions, and inject code thereafter.
- **Code-reuse Defense:** A defense mechanism against kernel-related code-reuse attacks is enforced, such as control-flow integrity (CFI) [1, 19], fine-grained code randomization [12, 31], or code-pointer integrity (CPI) [30]. Specifically, our prototype implementation of PT-Rand incorporates RAP [52], a public state-of-the-art CFI implementation for the Linux kernel. As mentioned before, existing defenses against code-reuse attacks cannot prevent data-only attacks against the page tables. (Our solution serves as a building block to prevent these protection frameworks from being undermined by data-only attacks against page tables.)
- **DMA Protection:** Direct Memory Access (DMA) [44, 55] cannot be exploited to bypass virtual memory permissions because an IOMMU [27] is configured to prevent DMA to security-critical memory.
- **Safe Initialization:** The attacker cannot attack the kernel prior the initialization of PT-Rand. This is not a limitation because PT-Rand is initialized at the early boot phase during which the attacker cannot interact with the kernel.
- **Source of randomness:** A secure (hardware) random number generator is available [3, 27, 53].
- **Side-channels:** Timing and cache side channel attacks as well as hardware attacks, like rowhammer [29], are orthogonal problems, and hence, beyond the scope of this paper. Nevertheless, we discuss in Section VI-A how we can adopt known techniques from Apple's iOS to prevent practical side-channel attacks.

B. Overview of PT-Rand

Our goal is to mitigate data-only attacks against the kernel page tables in the adversary setting explained in Section IV-A. To do so, we introduce the design and implementation of a novel kernel extension called PT-Rand. The main idea of PT-Rand is to (i) *randomize* the location of page tables securely, i.e., *prevent* the leakage of the randomization secret, and (ii) *substitute* pointers that reference page tables with physical addresses to obfuscate these references and prevent their leakage.

Figure 2 depicts the overall architecture and workflow of PT-Rand. During the early boot phase, the kernel operates only on physical memory. To guarantee a successful switch to virtual memory, contemporary kernels allocate an initial set of page tables at a constant and fixed address. These page tables manage the kernel's core functions as well as data areas, and remain valid for the rest of the kernel's life-time. To prevent the attacker from tampering with page tables, PT-Rand generates a randomization secret ❶, and randomizes the location of the initial page tables ❷. The randomization secret is stored in a privileged CPU register which is neither used during normal operation of the kernel nor accessible from user mode. Recall from Section IV-A that the attacker can only access the kernel memory, but not the kernel's registers. The latter would require the attacker to either launch a code-injection attack (prevented by W⊕X) or a code-reuse attack (mitigated by CFI [1], code randomization [31] or CPI [30]). After relocating the initial page tables to a random address, the kernel can no longer access these page tables through the 1:1 mapping. In particular, PT-Rand relocates the initial page tables in an unused memory region. As we will evaluate in detail in Section VI-A, the entropy for this memory region is reasonably high for contemporary 64-bit systems rendering brute-force attacks infeasible Ⓐ.

Note that the kernel features dedicated allocator functions for page table memory. For PT-Rand, we instrument these functions to (i) move the initial page tables to a random address, and (ii) always return physical addresses for any page table related memory allocation. In contrast, the default allocators always return a virtual address as a reference to newly allocated page table memory. This small adjustment allows us to obfuscate the location of page tables from user-level attackers, because the kernel code operates on virtual addresses when accessing page tables. Hence, at this stage, neither the attacker nor the kernel itself can access the page tables. In order to allow benign kernel code to still access the page tables, we modify all kernel functions that access page table memory: for each of these functions we convert the physical address to a virtual address based on the randomization secret generated in ❶.

However, during the early boot phase, the kernel has already saved references to the initial page tables in various data structures. Since the initial tables were not allocated with our modified allocator, the references contain obsolete virtual addresses. To avoid a kernel crash, PT-Rand updates all these references (virtual addresses) with the new physical address ❸. To this end, every reference to page tables now contains a physical address rather than a virtual address. Thus, the attacker aiming to locate page tables by reading the designated places of page table pointers [38] only retrieves physical addresses. Since there is no direct correlation between physical and virtual addresses, the attacker cannot use any leaked references to infer the corresponding virtual address Ⓑ. We also implemented PT-Rand such that no intermediate computation result that includes the randomization secret is ever written into memory. Specifically, we instruct the compiler to keep intermediate and the end result that include the randomization secret in registers, and prevent them from getting spilled.

Our modified page table memory allocator also randomizes any future page table allocations into the PT-Rand memory

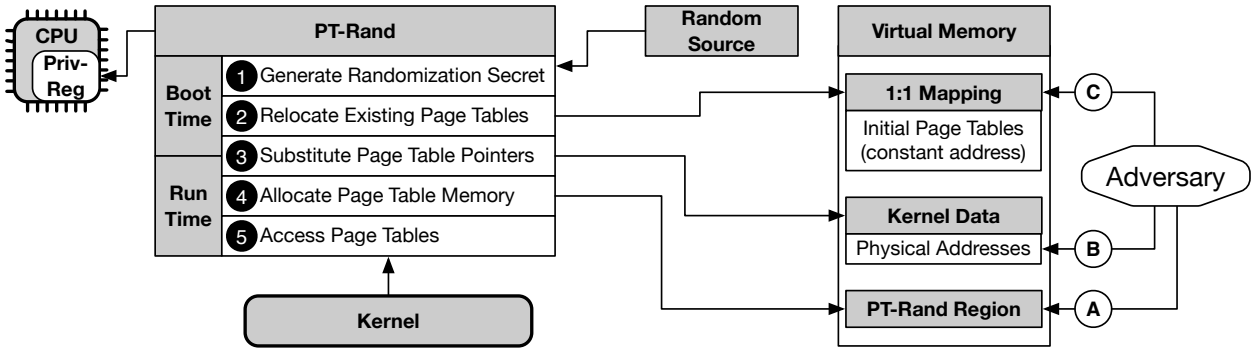

Figure 2: Overview of the different components of PT-Rand.

region ❹. Further, we ensure that every physical memory page that contains page table entries is unmapped from the 1:1 mapping. Hence, if the attacker discloses a physical address of a page table pointer, she cannot exploit the 1:1 mapping to read out page tables Ⓒ. Finally, PT-Rand provides an interface for the kernel to access and manage page tables ❺. In particular, PT-Rand translates the physical addresses of page table pointers to virtual addresses based on the randomization offset.

C. Challenges

To enable PT-Rand we had to tackle a number of challenges as we explain in the following. In Section V, we describe in detail how we address each challenge.

Page Table Allocation. Page tables are data objects that are dynamically allocated in the kernel. These objects are created by the page allocator, which is a central, low-level service in the kernel that manages physical pages. To randomize the memory pages where page tables are stored, we need to determine and instrument all kernel functions that allocate page tables.

Generating a Randomized Space. While the kernel needs to be able to locate randomized pages for performing benign changes, the attacker must not learn the new mapping. Consequently, we need to provide high entropy to avoid simple brute-force search. Furthermore, the new location of the page tables must not collide with other existing mappings in the virtual address space. This area must also be large enough to hold the page tables of all the processes running on the system.

Page Table References. Memory disclosure vulnerabilities allow the attacker to leak information about code and data pointers. Even fine-grained randomization schemes can be undermined if the attacker can map a single pointer to an instruction [48]. Hence, one of the main challenges in our design is to ensure that all references to page tables and the base address of the PT-Rand region are not leaked to the attacker. For this, we need to locate all page table references and replace them with physical addresses (❸ in Figure 2). Furthermore, we need to carefully handle benign page table changes by the kernel. Typically, the kernel processes page table pointers using virtual addresses on the kernel's stack. Since the stack is readable by the attacker, we need to provide a new mechanism to prevent leakage of these pointers.

Handling of 1:1 Mapping. As we discussed in Section II, the kernel maintains a 1:1 mapping for fast translation from virtual to physical addresses. ❹ in Figure 2 removes the page tables from this 1:1 mapping to prevent the attacker from learning the page table location. However, removal of page tables is not per-se possible. This is due to the fact that the 1:1 mapping deploys so-called large pages of 2MB by default. Hence, simply removing the page leads to deletion of adjacent data not related to page tables. In addition, we need to identify all functions that access page tables via the 1:1 mapping, and patch them to perform the translation based on the randomization secret.

Translation of Physical Addresses. At run-time, the kernel needs to repeatedly translate physical addresses to virtual addresses, e.g., during a page walk or when creating a page table entry. As we explained in Section II, this is efficiently performed based on the 1:1 mapping. However, when PT-Rand is in place, the kernel cannot use the 1:1 mapping anymore to translate physical addresses of page tables, because PT-Rand removed them from the 1:1 mapping. Consequently, the kernel has to distinguish between physical addresses of normal memory and physical addresses of page table memory as each of them needs a different translation mechanism. This distinction must be efficiently performed at run time to not impede the system's run-time performance.

V. Implementation

Our design as presented in Section IV-B requires low-level system modifications to the operating system kernel. We decided to prototype PT-Rand for the open-source Linux kernel. However, the concepts underlying our work on PT-Rand can be integrated into other contemporary operating systems. To this end, our kernel patch is comprised of 1382 insertions and 15 deletions across 45 source files.

Figure 3 shows how we integrate PT-Rand into the Linux kernel. We create wrapper functions for the page table allocator to randomize the virtual address of pages that contain page table entries. If the wrapper function is called to allocate memory which will be used to store page table entries, it allocates the memory at a random address in the PT-Rand region. The virtual address, pointing to this region, can only be computed by adding the randomization secret, which is stored in the third debug register. Pages for regular memory are still allocated in the 1:1 mapping and their virtual addresses within

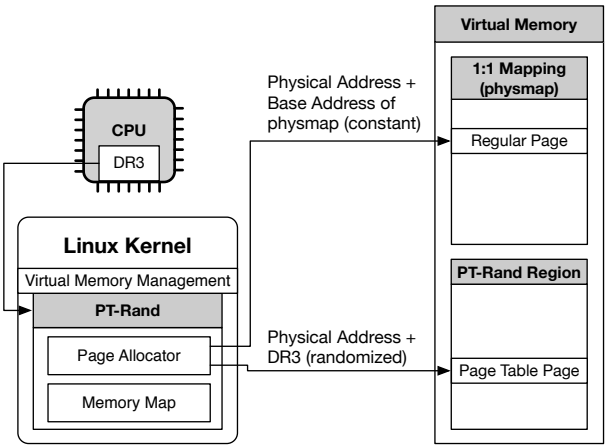

Figure 3: We modify the virtual memory location of page table pages by providing a randomization wrapper around the standard page allocator in the kernel. While randomized pages are removed from 1:1 mapping, regular page allocation requests still fall within this region.

Address	Size	Purpose
0x00000000000	47 Bits	User Space
hole caused by [48:63] sign extension		
0xffff8000000	43 Bits	Hypervisor
0xffff8800000	43 Bits	1:1 Mapping
0xffffc800000	40 Bits	**PT-Rand (Hole)**
0xffffc900000	45 Bits	vmalloc/ioremap
0xffffe900000	40 Bits	**Hole**
0xffffea00000	40 Bits	Memory Map
unused hole		
0xffffec00000	44 Bits	Kasan
unused hole		
0xffffff00000	39 Bits	Fixup Stacks
unused hole		
0xffffffff800	512M	Kernel Text
0xffffffffa00	1525M	Modules
0xffffffffff6	8M	vsyscalls
0xffffffffffe	2M	Hole

Figure 4: The x86_64 virtual memory map for Linux with four level page tables.

the 1:1 mapping are calculated by adding the base address of the 1:1 mapping, called *physmap* in Linux, to the physical address of the regular page.

We create wrapper functions for those kernel functions that need to access page table memory. When the kernel starts executing, the PT-Rand initialization function will first generate the randomization secret based on the standard kernel function `get_random_bytes()`. We enable the kernel to use the hardware-based random number generator (HW-RNG) to avoid low entropy during boot time. Note, that since version 3.16 the Linux kernel incorporates the output of HW-RNGs for generating random numbers by default[1].

In the following, we present the implementation details of PT-Rand according to the challenges we outlined in Section IV-C.

A. Page Table Allocations

The main task of PT-Rand is to map memory which contains page tables to a random location within the PT-Rand region. Page table allocation involves two steps: (i) randomization of the initial pages, and (ii) randomization of memory allocations which are used to store newly created page tables.

To complete the first step, we need precise knowledge of all existing references to the initial page tables, because after randomization these references need to be updated. The main challenge we faced is identifying all those references. To tackle this challenge, we followed a pragmatic approach: we reverse-engineered the kernel code execution after the location of the initial page tables have been randomized. Since every page table access based on an old reference leads to a kernel crash, we could determine the point of execution and associated kernel function which caused the crash. Thereafter, we inspected the kernel's source files and updated all references to use our new base address. After updating all references, kernel execution continued normally. In our extensive evaluation on different suites of benchmarks and complex software such as the Chrome browser (see Section VI-B) we have not experienced any kernel crashes.

To handle the second step, we extend the page table management functions in the kernel. Specifically, we create a wrapper function around the memory allocator for page tables. This allows us to modify their return values, i.e., they return physical addresses as a reference to the allocated memory rather than virtual addresses. Since there is no relation between physical and virtual memory addresses, the attacker cannot infer the location in the virtual memory by leaking the physical address.

We also create wrapper functions for every other kernel function that interacts with page tables to translate page table references (physical addresses) to virtual memory addresses before accessing the page tables.

B. Generating a Randomized Area

In order to provide sufficient protection against guessing attacks we require a high randomization entropy. While 64 bit architectures have a theoretical limit of 16EB of memory, current hardware is limited to support 256TB resulting in 48 bit randomization entropy.

The Linux kernel organizes the available virtual memory into different regions. Figure 4 is taken from the Linux kernel documentation [2] and reveals that the Linux kernel currently does not use all of the available virtual memory. In particular, we identified two memory *holes* of which each offers 1TB (40 Bit) free memory. Our proof-of-concept implementation of PT-Rand utilizes one of these holes for the PT-Rand region to store the page tables. Note that such large holes will always exist for 64 Bit systems due to the vast amount of available virtual memory.

[1] https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=be4000

C. Page Table References

As described in Section IV-A, the attacker can exploit kernel vulnerabilities to read from and write to kernel memory. However, these vulnerabilities do not allow the attacker to access content stored in registers. Hence, we can securely store the randomization secret into a dedicated register. For our proof-of-concept, we chose the fourth debug register `DR3`. We selected this register since it is only used for debugging purposes. It is noteworthy to mention that application debugging is still supported under PT-Rand. Typically, debuggers can use software and hardware breakpoints. The former are the default breakpoints and not affected by PT-Rand. For the latter, we only use one of the four available hardware breakpoints. Note that exploiting debugging facilities is a widely-accepted strategy when building defenses, e.g., TRESOR [36] or kBouncer [39]. Alternatively, we are currently exploring the feasibility of deploying any of the so-called model-specific registers (MSRs).

However, even though we store the base address in a privileged register, certain events (e.g., function calls) can spill temporary registers for several cycles to memory. As recently shown, this short time window can be exploited to undermine CFI checks [11]. PT-Rand tackles the attack by instructing the compiler to never spill registers which contain a randomized address. This is enabled by a GCC feature, called explicit register variables, which will always keep local variables in registers. However, given the complexity and many optimization techniques deployed by modern compilers, we can only guarantee that the above GCC compiler feature never leaks accordingly flagged variables, but not any intermediate calculation results. As a consequence, we are currently working on a GCC compiler extension that explicitly clears any intermediate results held in other registers.

D. Handling of 1:1 Mapping

The typical page size is 4KB. However, the kernel also supports page sizes of 2MB or 1GB. In particular, for the Linux kernel, the 1:1 mapping is configured to use 2MB pages by default.

In PT-Rand, we rely on unmapping memory that contains page tables from the 1:1 mapping. This becomes challenging when 2MB pages are used because the page might contain other data than page table memory that should not be removed from the 1:1 mapping. We tackle this challenge by reconfiguring the page size to 4KB pages at run time. However, in order to split a 2MB page into 4KB pages, we need to previously allocate 512 (i.e., 2MB divided by 4KB) new page table entries within the 1:1 mapping. Note that the 4KB split up only affects memory that contains page tables. For other memory parts, the kernel will continue to use large pages. Our performance evaluation in Section VI-B indicates that this change has no impact on the overall performance. Next, we configure each entry to map the corresponding memory of the 2MB page, and adopt the permissions and other metadata. Finally, we update the page table hierarchy to use the 4KB page tables entries instead of the one 2MB entry. After the conversion, we can relocate and delete only those 4KB pages that contained page table entries.

E. Translation of Physical Addresses

Since the page tables are relocated by PT-Rand, they are no longer accessible through the 1:1 mapping. Hence, as described in Section IV-C, the kernel has to utilize two different mechanisms when translating physical addresses to virtual addresses, namely one for physical addresses of pages that contain page table entries, and another one to translate physical addresses for non-page table related memory. Fortunately, the kernel already keeps track of the properties of each individual physical page in a dedicated data structure called `memory map`. When we analyzed this structure, we noticed that certain bits of the `flag` field are not used. This allows us to quickly distinguish among the two different types of pages. Specifically, we reserve one bit to mark if a physical page has been removed from the 1:1 mapping by PT-Rand. In other words, if the bit is set, the kernel knows that the requested access is a page table related access which requires handling based on the PT-Rand region.

At run-time, kernel functions that need to translate a physical to a virtual memory address will check the `flag` field of the memory map. If the physical page is not accessible through the 1:1 mapping, the kernel function will use the randomization secret provided by PT-Rand to determine the virtual memory address. Otherwise, the function uses the default translation through the 1:1 mapping. Hence, PT-Rand preserves the high efficiency for the majority of the page requests through the 1:1 mapping. In particular, we modified the `__va` macro to perform the check on the `flag` field. This function is the central point for translating physical to virtual addresses. PT-Rand does not cause any problems for external drivers, since external kernel drivers (e.g., graphic card drivers) are supposed to use these kernel functions to translate addresses.

During the implementation, we encountered that modifying `__va` raises another challenge: in the early boot phase, i.e., before PT-Rand relocates the initial page tables, a few kernel functions already invoke the modified macro. However, at this point of system state, the memory map is not yet initialized. Hence, the macro cannot yet access the `flag` field. We solved this problem by utilizing an unused macro called `__boot_va` which performs the same operation as the uninstrumented version of the `__va` macro. We patched all functions that are executed before the memory map is initialized to use the unmodified `__boot_va` macro.

VI. Evaluation

In this section, we present the evaluation results for PT-Rand. We first analyze security aspects such as randomization entropy and leakage resilience. Thereafter, we present a thorough investigation of the performance overhead incurred by PT-Rand. For this, we conducted micro-benchmarks on hot code paths, measure performance overhead based on SPEC CPU industry benchmarks, and quantify the impact on complex applications such as browsers.

A. Security Considerations

Our main goal is to prevent data-only attacks against the kernel page tables at run time. For this, we randomize the location of page tables per boot. In general, any randomization-based scheme must resist the following attack

vectors: (i) guessing attacks, (ii) memory disclosure through code and data pointers, and (iii) memory disclosure through spilled registers. In the following, we discuss each attack vector to demonstrate the effectiveness of PT-Rand. We also include an exploit in our study to demonstrate that exploit hardening mechanisms at the kernel-level can be bypassed when PT-Rand is not applied.

Guessing Attacks. Low randomization entropy allows the attacker to guess the randomization secret with high probability [47]. The randomization entropy of PT-Rand depends on: (1) the number of guesses, (2) the size of the region where the page tables are allocated, and (3) the overall size of memory that is required to store all page tables.

We limit the number of attacker's guesses by configuring the kernel to initiate a shutdown in case of an invalid memory access in kernel memory. Note that this has no impact on the kernel's execution. In fact, this was the default behavior of previous versions of the Linux kernel. As described in Section V-B, we utilize an unused memory region of 1TB (40 Bit) to randomize the memory allocations for the page tables. However, the smallest memory unit in paging is a 4KB (12 Bit) page. This means when one page table entry is placed randomly into the PT-Rand region, 4KB of memory become readable. Hence, the attacker does not have to guess the correct address of a particular page table entry but only the start address of the page which contains the entry. As a consequence, the total randomization entropy available for PT-Rand is 28 Bit.

For a deterministic attack, the attacker has to manipulate a specific page table entry S that protects a specific function of the kernel. Alternatively, it might be sufficient for the attacker to corrupt an *arbitrary valid* entry A of the page table. However, it is not guaranteed that this modification will allow the attacker to compromise the kernel, thus, the attack success is probabilistic. Hence, we calculate the success probability that the attacker can correctly guess the address of the page which contains S. We denote this probability with $p(x)$ which depends on the number of pages, denoted by x that contain page table entries.

We can reduce the problem of calculating the success probability $sp(x)$ to a classical urn experiment without replacement and with three different colored balls: black, red, and green. The *black* balls represent the unmapped pages. The attacker loses the experiment by drawing a black ball (because accessing an unmapped page crashes the operating system). The *red* balls represent the valid pages, however, they do not contain the attacker's target page table entry S. The attacker is allowed to continue and draw another ball, as long as the attacker draws a red ball (access to a valid page). A *green* ball represents the page containing the page table entry S that the attacker aims to modify. With SG we denote the event that the attacker draws the green ball eventually without drawing a black ball (guessing the correct address of S without accessing an unmapped page). Hence, the probability of SG is the sum of the probabilities that the attacker draws the green ball in the first try plus the probability that the attacker draws the green ball after drawing the i-th red ball where $i \geq 1$. The resulting probability of SG is computed as follows:

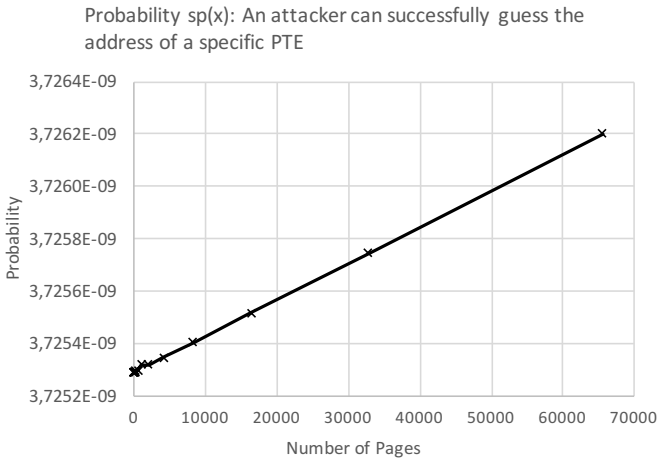

Figure 5: Probability for guessing attacks based on the number of mapped pages in the PT-Rand region.

$$Pr[SG] = p(x) = \frac{1}{2^{28}} + \sum_{i=1}^{x} \frac{\binom{x}{i}}{\binom{2^{28}}{i}} * \frac{1}{2^{28} - i}$$

Figure 5 plots the probability that the attacker can succeed in guessing a specific page table entry if up to 2^{16} memory pages for page tables are allocated. The graph shows that even if a high number of page table entries (PTEs) are allocated, the attacker's success probability is still very low ranging from $3.725 * 10^{-9}$ to $3.726 * 10^{-9}$. We measured the number of page tables for a variety of different systems and configurations. For a normal desktop system, we observed that between 2,000 and 4,000 PTE pages were allocated. If we start a virtual machine up to 16,000 pages for PTEs are allocated. Lastly, our server (24 cores and 64GB RAM) running 9 virtual machines in parallel allocates up to 33,000 pages for PTEs. As shown in Figure 5, the probability grows linearly. Therefore, even if the attacker attempts to decrease the entropy by forcing the operating system to allocate more pages that contain page table entries[2] the attacker's success probability is very low. Further, PT-Rand can prevent attacks on the entropy by limiting the amount of page tables to a number that will guarantee a user configurable amount of entropy.

For this reason, even if the attacker tries to decrease the randomization entropy by forcing PT-Rand to allocate a large amount of memory within the PT-Rand region, e.g., by spawning new processes, the success probability will not increase significantly before such an attack can be detected, e.g., by only allowing a fixed number of allocated pages.

Memory References. Memory disclosure is another severe threat to any kind of randomization scheme. For PT-Rand, we assume that the attacker can disclose any kernel data structure, and therefore, possible references to page tables. Hence, we obfuscate the references to page tables in all kernel data structures by substituting the virtual addresses with physical

[2]the attacker can force the operating system to create new page table entries by starting new processes.

addresses. Note, there is no correlation between virtual and physical addresses. Therefore, the attacker does not gain any information about the real location of the page tables by disclosing obfuscated addresses. Since our modified memory allocator for page-table memory only returns obfuscated references, the attacker cannot access page tables by reading those pointers. The remaining potential source of leakage are functions that did not use our modified allocator. Recall, all functions that access the page tables now expect a physical address. Hence, if these functions receive a virtual memory address of a page table entry, they will automatically try to translate them using the randomization secret. The result is very likely an invalid address which will lead to a kernel crash.

Spilled Registers. As recently demonstrated in [11], even temporarily spilled registers which contain a security-critical value can compromise PT-Rand. To prevent *any* access to the debug register (`DR3`) that contains the randomization secret, we patched the Linux kernel code to never access `DR3`, i.e., `DR3` cannot be accessed through any kernel API. Note that the CPU *does not* spill debug registers during interrupts [27]. Further, we prevent the compiler from writing the randomization secret to the stack by performing all computations in registers and never save or spill the result to memory. However, there might be cases, where a register that contains an intermediate value is spilled on the stack due to a hardware interrupt. In contrast to software interrupts, which we disable during page walks, hardware interrupts cannot be disabled. This opens a very small time window that may enable the attacker to use a concurrent thread to disclose register values, and potentially recover parts of the randomization secret. We performed preliminary experiments with a setting that favors the attacker to implement this attack, and did not succeed. Nevertheless, we are currently exploring two different strategies to mitigate such attacks. The first strategy is to further decrease the already small time window where register values could potentially be leaked. In particular, we envision to instrument the page table reads, by rewriting them with inline assembly, such that the de-obfuscated address is only present in the register for a couple of instructions. After accessing the page-table memory all registers that contain (intermediate values of) the randomization secret are set to zero. Alternatively, the second strategy ensures that the attacker cannot use a concurrent thread to access the stack of a victim thread that got interrupted and whose registers got temporarily spilled to memory. This can be achieved by using different page tables per kernel thread. Specifically, this allows us to assign stack memory per kernel thread which cannot be accessed by other (concurrent) threads. Therefore, even if intermediate values are spilled to memory, the attacker cannot leak them using concurrent threads. A simpler version of this technique, where the kernel uses a different page table per CPU, is already deployed in the grsecurity patch [50].

Real-world Exploit. We evaluated the effectiveness of PT-Rand against a set of real-world vulnerabilities. In particular, we use an information disclosure vulnerability in the Linux kernel to bypass KASLR[3], and a vulnerable driver which does not sanitize pointers provided by a user-mode application (CVE-2013-2595) to read and write arbitrary kernel memory. Based on these attack primitives, we develop an attack which allows us to execute arbitrary code in the kernel, despite having the kernel protected with state-of-the-art CFI for the kernel. The goal of our attack is to (i) change the memory permissions of a page that contains the code of a pre-defined kernel function to writable, (ii) overwrite the function with our shellcode, and (iii) finally trigger the execution of this function to instruct the kernel to execute our shellcode with kernel privileges.

To retrieve the KASLR offset, we use the aforementioned information disclosure vulnerability. The vulnerability allows the attacker to disclose the absolute address of a kernel function. Since we can determine the relative offset of this function to the start address of the kernel code section, we can compute the absolute address of the kernel after KASLR. Based on this address, we can compute the address of every function or global variable of the kernel since KASLR only shifts the whole kernel by a randomized offset during boot. In an offline analysis of the kernel image, we discovered a global variable that holds a reference to the `task_struct` of the initial process. The `task_struct` is a kernel data structure in which the kernel maintains information about each process, like id, name and assigned virtual memory. Specifically, it contains a pointer to the `mm_struct` which maintains information about the memory that is assigned to the process. Within this structure, we discovered a virtual memory pointer to the root of the page table of the corresponding process.

Using the arbitrary read capability and the 1:1 mapping, we traverse the page table to the entry that maintains the permissions for the system call `sys_setns`. Next, we set this page to writable and overwrite the beginning of `sys_setns` with our shellcode. In our proof-of-concept exploit, we re-write the function to elevate the current process' privileges to root. Naturally, other payloads are possible as well, like installing a kernel rootkit. After we modified the system call function, we set the corresponding page table entry again back to readable and executable, and invoke the system call to trigger the execution of our shellcode.

As explained in detail, this attack does not involve changing any code pointer. Hence, it resembles a data-only attack that cannot be mitigated by defenses against control-flow hijacking. However, after hardening the kernel with PT-Rand, this attack fails since we cannot reliably locate the correct page table entry for system call `task_struct`.

Side-channel Attacks. As stated in Section IV-A, preventing side-channel attacks is beyond the scope of this paper. However, since side-channel attacks have the potential to undermine the security guarantees of PT-Rand, we will shortly discuss how these attacks work and how the kernel could be extended to prevent them.

Through side channels the attacker can disclose information about the kernel memory layout. In particular, the attacker discloses whether a kernel memory page is mapped. Hence, the attacker, in user mode, will attempt to read or write to a kernel memory page. Since kernel memory cannot be accessed by the user-mode programs such an attempt will result in an access violation. However, the time elapsing between the attempted access and the access violation depends on whether the page is mapped. Hund et al. [26] first demonstrated the feasibility

[3]. This vulnerability was silently fixed by the Linux kernel maintainers which is why there was no official CVE number assigned: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=b2f739

of this attack by measuring the different timings the page fault handler needs to deliver an exception to the user mode to bypass kernel ASLR. Wojtczuk [56] improved this attack by using Intel's Transactional Synchronization Extensions (TSX) which provides new instructions for hardware-aided transactional memory. The advantage of using TSX instructions to access kernel memory is that the faulting access does not invoke the page fault handler, and hence, allows to execute the previous attack of Hund et al. faster and with higher precision.

These timing-side channels exist because the user and kernel mode share the same address space, i.e., they use the same page tables. Hence, we can prevent such attacks by ensuring that the user and kernel mode use different page tables similar to Apple's iOS [34].

Code-reuse attacks. PT-Rand is complementary to defenses against code-reuse attacks, like CFI [1, 19], CPI [30], or fine-grained randomization [12, 31]. We applied the open-source version of the CFI kernel protection for Linux RAP [52] to prevent the attacker from hijacking the control flow. Hence, the attacker cannot use code-reuse attacks like ROP to leak the randomization secret.

B. Performance

We measured the performance overhead incurred by PT-Rand based on SPEC CPU2006, LMBench, Phoronix, and Chromium benchmarks. All measurements are taken on an Intel Core i7-4790 CPU running at 3.60GHz with 8 GB RAM using Debian 8.2 with a recent Linux kernel, version 4.6.

SPEC CPU 2006. The SPEC CPU 2006 benchmarks measure the performance impact of PT-Rand on CPU-intensive applications. We executed the entire benchmark suite with the default parameters for reference tests: three iterations with reference input. We did neither encounter any problems during the execution (i.e., crashes) nor strong deviations in the results of the benchmarks. Figure 6 summarizes the performance impact of PT-Rand compared to a non-modified kernel. The average performance overhead of PT-Rand is only 0.22% with worst-case overhead of only 1.7%. This confirms the high efficiency and practicality of PT-Rand for contemporary systems.

Note that a few benchmarks perform marginally better when PT-Rand is applied. Such deviations have been also observed in prior work, and can be attributed to negligible measurement variances.

LMBench. Most of our modifications affect the launch and termination phase of an application's lifecycle. This is due to the fact that PT-Rand needs to randomize the page tables at program start and remove them from the 1:1 mapping. When an application terminates, PT-Rand needs to de-randomize its page tables and make this memory again accessible through the 1:1 mapping. Hence, we additionally tested our approach using the popular LMBench micro benchmark suite [35] to assess the overhead for these critical phases. Specifically, LMBench collects timing information for process launch, fork, and exit. We measured an absolute overhead of less than 0.1 ms on average which is hardly noticeable to the end-user of a PT-Rand-hardened system

Benchmark Name	Relative Overhead
IOZone	1.0%
PostMark	1.8%
OpenSSL	-2%
PyBench	-0.9%
PHPBench	-0.2%
Apache	0.8%

Table I: Phoronix benchmark results.

LMBench also includes other benchmarks, e.g., performance impact on memory accesses, system calls or floating point operations. We successfully executed all benchmarks and observed no measurable impact on the performance.

Phoronix. Besides SPEC CPU2006 and LMBench we measured the performance impact of PT-Rand with the Phoronix benchmark suite [41] which is widely used to benchmark the performance of operating systems. Table I[4] summarizes the results which are consistent with the results of the SPEC CPU2006 benchmarks.

Chromium. Finally, we measured the performance overhead for Google's Chromium in two scenarios: 1) we ran the popular browser benchmarking frameworks JetStream, Octane, and Kraken, to measure the run-time overhead for daily usage, and 2) we modified Chromium such that it terminates directly after loading to measure the load-time overhead. We repeated both experiments three times and determined the median to account for small variances.

For the Chromium web browser, we report a run-time overhead of -0.294% and a load-time overhead of 9.1%. The run-time overhead represents the arithmetic mean of 0.76% for JetStream, 1.183% for Kraken, and -2.825% for Octane.

The browser frameworks measure browser engine latency and load, with a focus on JavaScript execution. While these tests do not accurately measure the direct performance overhead of PT-Rand, they provide a first estimation of the performance impact on the popular end-user applications such as a web browser. Given only -0.294% overhead, we confirm that PT-Rand does not negatively impact performance of user applications.

To measure the load-time overhead, we simply added a `return` instruction in the main function of Chromium. This ensures that Chromium immediately terminates after it is completely loaded. We measured the elapsed time based on the Unix tool `time`. With less than 1 ms load-time overhead we assert that PT-Rand does not impair the user experience. We find these results to be in line with our LMBench test results for process creation and termination.

C. Robustness

To evaluate the robustness of PT-Rand we executed a large number of popular user-mode applications, and the three aforementioned benchmarking suites. We did not encounter

[4] Note that we excluded some of the benchmarks because we got errors when executing them on a vanilla system.

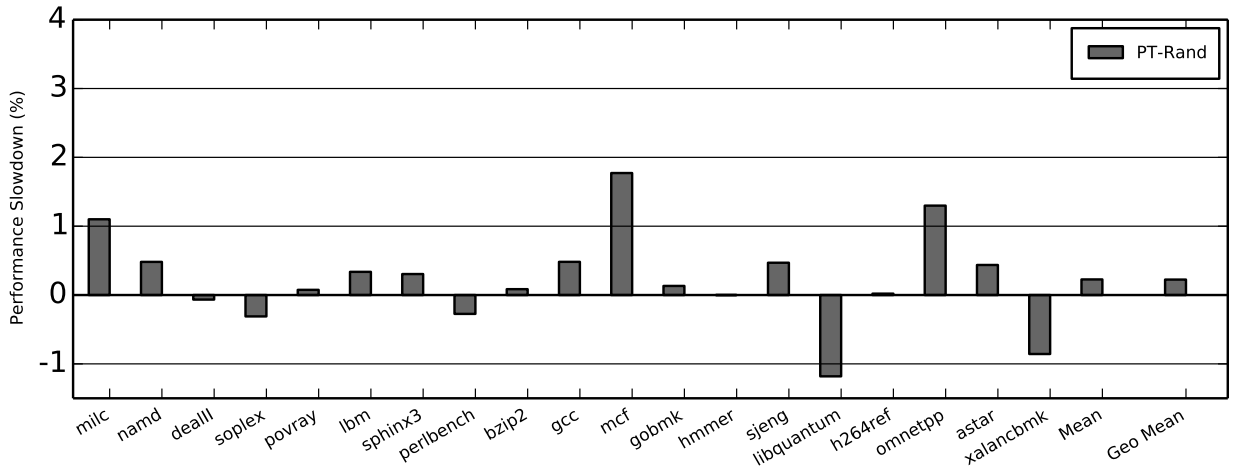

Figure 6: Overhead of page table randomization on SPEC CPU2006

any crashes during these tests, and all applications behaved as expected. To further stress test our implementation we executed the Linux Test Project (LTP) [32]. The LTP is comprised of different stress tests that can be used to evaluate the robustness and stability of the Linux kernel. We executed the most important tests under PT-Rand, and did not encounter any deviation in the behavior compared to the vanilla kernel. Finally, we did not encounter any compatibility issues or crashes when combining PT-Rand with RAP [52].

VII. Discussion

Choice of 64-bit. The choice of 64 bit architectures is not a conceptual limitation. PT-Rand can be ported to 32 bit architectures. However, similar to ASLR, PT-Rand relies on the available randomization entropy which is known to be low for 32 bit systems [47]. Hence, we focused our efforts on hardening 64 bit-based architectures because nearly all commodity desktops and servers feature 64 bit CPUs. Even mobile devices are increasingly deploying 64 bit CPUs. As of 2013, Apple's iPhone embeds a 64 bit processor and iOS 9 runs exclusively on 64 bit processors. In a similar vein, Google runs 64 bit processors for their latest Nexus smartphone.

Malicious Drivers. Our threat model does not consider injection of malicious drivers. These would allow the attacker to execute arbitrary code in kernel mode without requiring exploitation of a memory corruption vulnerability. As such, malicious drivers could access and leak the randomization secret. However, note that all modern operating systems support driver signing to prevent the loading of such malicious drivers thereby ensuring that the randomization secret is not leaked to the attacker.

Physical Attacks. Similar to previous work [13], the main focus of this work is to prevent remote attacks against the kernel. As a result, attacks that rely on physical access to the victim system are beyond the scope of this work. For instance, several attacks in the past utilized special hardware (e.g., FireWire [44]) to create a snapshot of the physical memory [22]. Such snapshots can be analyzed by means of forensic tools to identify critical data structures such as the page tables in the case of PT-Rand. However, they require physical access to the RAM. Creating a memory snapshot remotely to detect the location of page tables is not feasible because the remote attacker has only access to virtual memory, i.e., linearly scanning virtual memory will eventually lead to a system crash since we move the page tables to a memory region where the majority of surrounding pages are not mapped.

No Disturbance of Workflow. As described in Section V, we use a debug register to store the randomization secret. This prevents the attacker from leaking the secret by means of a memory disclosure. As a consequence, the chosen debug register is no longer available for debugging purposes. However, debug registers (`DR0-DR2`) are still available. Furthermore, these debug registers are only used for hardware breakpoints. Software breakpoints, which are far more common during debugging, are not affected by our change.

Lastly, it is noteworthy to mention that PT-Rand does not depend on any specific operating system features and can be ported to other operating systems.

VIII. Related Work

A. Data-only Attack and Defenses

Chen et al. [10] demonstrated the effectiveness of data-only attacks against server applications. In particular, they showed how to bypass authentication checks, and escalate privileges to root without the need to hijack the application's control flow. Hu et al. [23] created FLOWSTITCH to automatically generate such data-only attacks. This is achieved by first analyzing the execution flow that leads to the memory-corruption vulnerability, and then exploring how this vulnerability can be exploited to manipulating existing data flows of the application to leak sensitive information or escalate privileges. In a follow-up work, Hu et al. [24] introduced the notion of data-oriented programming (DOP) which extends these attacks to a Turing-complete machine.

To mitigate data-only attacks a number of data-randomization approaches have been proposed. Cadar et al. [9]

and Bhatkar et al. [7] apply static analysis to divide data accesses into equivalence classes. Next, they instrument all data accesses to use a xor key per equivalence-classes for reading and writing data from/to memory. This prevents the attacker to exploit a memory-corruption vulnerability to access arbitrary data. However, the instrumentation of data accesses is expensive with up to 30% run-time overhead.

Giuffrida et al. constantly re-randomize the addresses of data at run-time [21]. They implemented a live randomization system for Minix3 and report a modest overhead of 5%. However, between each randomization the attacker has a small time window where the attacker can leak the address, and manipulate content of the targeted data. Bigelow et al. [8] avoid this time windows in the context of server applications by re-randomizing the address space after the data was sent over the network. Hence, all potentially leaked information are re-randomized when they reach the attacker.

B. Register-based Information Hiding

Many defenses rely on a secret value that should not be disclosed to the attacker. To do so, one possibility is to store the secret value in a register. TRESOR [36] uses a debug register of x86 to hide an AES key from attackers. Similar to PT-Rand, Oxymoron [6], Code-Pointer Integrity [30], and ASLR-Guard [33] use a register to hide the base address of a memory region that contains sensitive data. In particular, these defenses use a segmentation register to hide their trusted computing base from the attacker.

C. Kernel and Page Table Protection

Several kernel defenses have been proposed that also protect the page table against malicious manipulations [4, 13, 20, 45, 54]. In general, existing approaches are based on a dedicated kernel monitor that enforces a set of pre-defined policies at run time, including integrity policies for page tables. To the best of our knowledge, PT-Rand is the first approach which follows a randomization-based approach to defend against data-only attacks against page tables.

SecVisor [45] and HyperSafe [54] follow a hypervisor-based approach. SecVisor enforces W$\oplus$X for the kernel space to ensure the integrity of the kernel code. This is done by using memory virtualization to allow only certain physical pages to be executable. SecVisor provides an interface to the kernel to allow new physical pages to be marked as executable. These requests are checked against a user-provided policy which is not further specified. HyperSafe protects its page tables by marking them read-only, and checks before updating the page tables if the update conforms to a immutable set of policies which should prevent malicious changes of page tables. Since the hypervisor maintains its own memory domain, virtualized guests cannot compromise its integrity by means of data-only attacks. However, the page tables maintained in the hypervisor itself can be compromised by the attacker. For instance, evasion attacks can be deployed to attack the hypervisor from a virtualized guest system [55]. Another practical shortcoming of hypervisor approaches is the incurred performance overhead. SecVisor reports 14.58% average overhead (SPECInt) and HyperSafe 5% overhead (custom benchmarks). In contrast, PT-Rand only incurs 0.22% for SPEC CPU benchmarks. Some of the extra overhead of SecVisor and HyperSafe can be attributed to additional checks that go beyond table protection. However, the hypervisor itself will always add some extra execution overhead. In addition, these approaches rely on extra hardware features such as virtualization extensions.

Another recent example for a hypervisor-based approach is KCoFI [13] which enables full-system CFI enforcement for an operating system kernel. It also securely stores the policies for safeguarding its virtualized guests inside a memory region that is only accessible by the hypervisor. However, this solution also comes with significant overhead of up to 200%, and suffers from the constraints mentioned already above, i.e., requiring virtualization extensions and deployment of a hypervisor.

SPROBES and TZ-RKP both leverage hardware trust anchors [5, 20]. In particular, both issue run-time checks for the kernel's memory management functions. These checks are executed inside the hardware-enabled secure environment ARM TrustZone. This secure environment cannot be tampered with by any other software. The overhead of TZ-RKP is up to 7.56%. In addition to the higher overhead, SPROBES and TZ-RKP rely on dedicated hardware trust anchors to protect page tables.

SKEE implements similar run-time checks to SPROBES and TZ-RKP [4]. It utilizes the fact that ARM provides two registers for paging. This enables SKEE to isolate the run-time checks from the kernel. The overhead for protecting memory management varies between 3% and 15%.

Policy-based approaches like HyperSafe [54] and SPROBES/TZ-RKP [5, 20] mark pages that contain the page table structures as read-only to prevent malicious modifications. However, when the operating system needs to update the page tables these defenses mark the corresponding pages temporarily writable which opens a time window in which the attacker can concurrently modify page tables entries on the same page.

PaX/Grsecurity [50] provide a patch with various techniques to further harden the Linux kernel. Amongst others the patch aims to prevent information leaks, and randomizes important data structures at compile time. However, it does not deploy any techniques to explicitly prevent data-only attacks against the page table.

Windows 10 [28] recently released an update to randomize the base address which is used to compute the address of page table entries. However, the randomized base address is not protected against information disclosure attacks which is why the attack we implemented in Section VI-A will also work against Windows 10. In contrast, PT-Rand mitigates information-disclosure attacks by keeping the randomization secret in a register, which cannot be accessed by the attacker, and by obfuscating all pointers to the page tables.

IX. Conclusion

Exploitation of software is a pre-dominant attack vector against modern computing platforms. In particular, exploits against the kernel are highly dangerous as they allow the attacker to execute malicious code with operating system privileges. The research community has introduced several

classes of exploit mitigation techniques that significantly raise the bar of such attacks. However, these defenses build on the assumption that the attacker cannot alter the kernel's page tables which is the main place to manage access permissions of code and data memory. For the first time, we introduce a highly-efficient randomization technique that enables effective protection against page table corruption attacks for a contemporary Linux-based system. Our open-source solution, called PT-Rand, randomizes the location of all page tables, and obfuscates all references to the page tables without requiring extra hardware, costly hypervisors, or inefficient integrity checks. PT-Rand is a practical and necessary extension to complement existing mitigation technologies such as control-flow integrity, code randomization, and code pointer integrity.

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.

REFERENCES

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

[2] Andi Kleen. x86_64 Linux Virtual Memory Map. http://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt, 2004.

[3] ARM. ARM architecture reference manual. http://silver.arm.com/download/ARM_and_AMBA_Architecture/AR150-DA-70000-r0p0-00bet9/DDI0487A_h_armv8_arm.pdf, 2015.

[4] A. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning. Skee: A lightweight secure kernel-level execution environment for arm. In *23rd Annual Network and Distributed System Security Symposium*, NDSS, 2016.

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

[6] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

[7] S. Bhatkar and R. Sekar. Data space randomization. In *5th Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, DIMVA, 2008.

[8] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely rerandomization for mitigating memory disclosures. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[9] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro. Data randomization. Technical Report MSR-TR-2008-120, Microsoft Research, 2008.

[10] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium*, USENIX Sec, 2005.

[11] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[12] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

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

[14] CVE Details. Linux kernel: Vulnerability statistics. http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html, 2016.

[15] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (Just-In-Time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium*, NDSS, 2015.

[16] N. A. Economou and E. E. Nissim. Getting physical extreme abuse of intel based paging systems. https://www.coresecurity.com/system/files/publications/2016/05/CSW2016%20-%20Getting%20Physical%20-%20Extended%20Version.pdf, 2016.

[17] J. Edge. Kernel address space layout randomization. http://lwn.net/Articles/569635, 2013.

[18] S. Esser. iOS kernel exploitation. In *Blackhat Europe*, BH EU, 2011.

[19] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *1st IEEE European Symposium on Security and Privacy*, Euro S&P, 2016.

[20] X. Ge, H. Vijayakumar, and T. Jaeger. SPROBES: Enforcing kernel code integrity on the trustzone architecture. In *Mobile Security Technologies*, MoST, 2014.

[21] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *21st USENIX Security Symposium*, USENIX Sec, 2012.

[22] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin. Os-sommelier: Memory-only operating system fingerprinting in the cloud. In *3rd ACM Symposium on Cloud Computing*, SoCC, 2012.

[23] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium*, USENIX Sec, 2015.

[24] H. Hu, S. Shinde, A. Sendroiu, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *37th IEEE Symposium on Security and Privacy*, S&P, 2016.

[25] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *18th USENIX Security Symposium*, USENIX Sec, 2009.

[26] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[27] 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, 2015.

[28] A. Ionescu. Owning the image object file format, the compiler toolchain, and the operating system: Solving intractable performance problems through vertical engineering. www.alex-ionescu.com/?p=323, 2016.

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

[30] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[31] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[32] LTP developer. The linux test project. https://linux-test-project.github.io/, 2016.

[33] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[34] T. Mandt. Attacking the ios kernel: A look at "evasi0n". http://www.nislab.no/content/download/38610/481190/file/NISlecture201303.pdf, 2013.

[35] L. McVoy and C. Staelin. Lmbench: Portable tools for performance

analysis. In *USENIX Technical Conference*, ATEC, 1996.

[36] T. Müller, F. C. Freiling, and A. Dewald. Tresor runs encryption securely outside ram. In *20th USENIX Security Symposium*, USENIX Sec, 2011.

[37] MWR Labs. MWR Labs Pwn2Own 2013 write-up - kernel exploit. http://labs.mwrinfosecurity.com/blog/2013/09/06/mwr-labs-pwn2own-2013-write-up---kernel-exploit, 2013.

[38] MWR Labs. Windows 8 kernel memory protections bypass. http://labs.mwrinfosecurity.com/blog/2014/08/15/windows-8-kernel-memory-protections-bypass, 2014.

[39] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *22nd USENIX Security Symposium*, USENIX Sec, 2013.

[40] Perception Point Research Team. Analysis and exploitation of a linux kernel vulnerability (cve-2016-0728). http://perception-point.io/2016/01/14/analysis-and-exploitation-of-a-linux-kernel-vulnerability-cve-2016-0728/, 2016.

[41] Phoronix. Phoronix test suite. http://www.phoronix-test-suite.com/, 2016.

[42] S. Renaud. Technical analysis of the windows win32k.sys keyboard layout stuxnet exploit. http://web.archive.org/web/20141015182927/http://www.vupen.com/blog/20101018.Stuxnet_Win32k_Windows_Kernel_0Day_Exploit_CVE-2010-2743.php, 2010.

[43] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *11th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID, 2008.

[44] F. L. Sang, V. Nicomette, and Y. Deswarte. I/O attacks in Intel PC-based architectures and countermeasures. In *SysSec Workshop*, SysSec, 2011.

[45] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. *ACM SIGOPS Operating Systems Review*, 41(6):335–350, 2007.

[46] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2007.

[47] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2004.

[48] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[49] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *23rd Annual Network and Distributed System Security Symposium*, NDSS, 2016.

[50] B. Spengler. Grsecurity. *Internet [May, 2016]. Available on: http://grsecurity.net*, 2016.

[51] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[52] P. Team. RAP: RIP ROP. https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf, 2015.

[53] Trusted Computing Group. Tpm 1.2 protection profile. https://www.trustedcomputinggroup.org/tpm-1-2-protection-profile/, 2016.

[54] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395. IEEE, 2010.

[55] R. Wojtczuk. Subverting the xen hypervisor. In *Blackhat USA*, BH US, 2008.

[56] R. Wojtczuk. Tsx improves timing attacks against kaslr. https://labs.bromium.com/2014/10/27/tsx-improves-timing-attacks-against-kaslr/, 2014.