PAGE TABLE RANDOMIZATION: MEMORY PROTECTION RESILIENT TO DATA-ONLY ATTACKS

DAVID GENS

TECHNISCHE UNIVERSITÄT DARMSTADT

Informatik

System Security Lab

Prof. Dr.-Ing. Ahmad-Reza Sadeghi
Dr.-Ing. Lucas Vincenzo Davi
M.Sc. Christopher Liebchen

February 8, 2016 – version 1.0

DECLARATION

I declare that I have produced this document without the prohibited assistance of third parties and without making use of aids other than those specified. Notions taken over directly or indirectly from other sources have been identified as such. This document has not previously been presented in identical or similar form to any other German or foreign examination board.

Darmstadt, February 8, 2016

David Gens

ABSTRACT

Security is a huge challenge for today's computer systems. The software stack is vulnerable to a vast number of runtime attacks. For this reason, many defenses have been proposed in the past. These defenses are implemented and enforced by the operating system kernel. As a result, exploitation techniques initially aimed at application level code have been extended to also target system level software. Such attacks threaten to bypass these defenses by disabling memory protection in the kernel through exploitation of memory-corruption vulnerabilites. Memory protection is implemented through paged virtual memory, a concept that relies on data objects in kernel memory, which are called page tables. We propose a modification to the kernel, that ensures the integrity of page tables without requiring additional hardware capabilities, expensive runtime checks, or increasing TCB complexity. Our approach incurs almost zero overhead and is therefore highly practical.

ZUSAMMENFASSUNG

Sicherheit ist eine der großen aktuellen Herausforderungen für heutige Rechnersysteme. Diese Systeme sind für eine Reihe von Laufzeitangriffen anfällig, für die eine Vielzahl von Defensivmaßnahmen vorgeschlagen und implementiert wurden. Die implementierten Kontrollmechanismen werden im Kern des Betriebssystems umgesetzt. Dieser wird somit zum attraktiven Angriffsziel, die hierbei verwendeten Techniken sind angelehnt an bekannte Laufzeitangriffe auf Anwendungssoftware. Die erfolgreiche Ausnutzung von Schwachstellen im Betriebssystem ermöglicht das Umgehen des Zugriffsschutzes und somit das Ausschalten der implementierten Defensivtechniken. Kern dieser Arbeit ist eine Modifikation der virtuellen Speicherverwaltung zum probabilistischen Schutz von Seitentabellen, die den Zugriffsschutz realisieren. Das vorgeschlagene Konzept kommt ohne zusätzliche Anforderungen an Hard- oder Software aus. Erste Experimente bestätigen die hohe praktische Relevanz des erarbeiteten Konzeptes durch günstiges Laufzeitverhalten.

CONTENTS

1 INTRODUCTION

Computer systems have come a long way since the early days of electromechanical computing devices developed during World War II. The advent of the third industrial revolution transforms the way our modern societies create, use, and exchange information. Most of our data is digital [36], ready to be processed automatically by computer systems. We already depend on these systems. They are used to control and coordinate our electricity networks, transportation and shipping, financial markets, and military.

What these systems do, is defined by the software they run, i.e., the algorithms and data processed by the hardware. Users directly interact with application-level software, whereas the computing system itself is managed and controlled by system-level software. One important aspect of an operating system kernel is security. Apart from the traditional security goals confidentiality, integrity, and availability, operating system design faces additional challenges. For instance, it must provide separation of individual processes, and ensure strong isolation between kernel and userland [75].

To address these challenges, hardware support is needed. Additionally, integrity of critical data structures has to be maintained by the kernel. This includes data objects describing process memory, access properties, user capabilities, the file system, network- and interprocess communication, and many more. However, kernel code continuously suffers from programming errors, logical flaws, or configurational inconsistencies, which can lead to different classes of vulnerabilities. Information disclosure, denial of service, privilege escalation, and code execution are serious threats that an operating system kernel faces.

These are common weak spots which can typically be found during extensive test runs, or laborious audits. Many of these errors may have only subtle to practically no consequences for the system within a benign environment. However, in the presence of an adversary these errors expose the system to a range of possible runtime attacks. In addition, the majority of systems deployed in the real world does alas not undergo the required, rigorous inspections.

Memory-corruption vulnerabilities allow an adversary to read or write the contents of main memory at some point during the execution, through exploitation of type errors, buffer overflows, use-after-free, or dangling pointers. These kinds of bugs represent one major starting point for runtime exploitation. The successful exploitation of one of these bugs grants adversaries the possibility to modify control flow in the kernel, modify critical data structures without violating its control-flow graph, and disclose information like data-pointers, code-pointers, or system credentials.

As a consequence, a range of countermeasures have been proposed in the past [2, 14, 50, 60] to limit the damage an adversary could cause to the system. Implementations of these approaches can be roughly categorized as kernel monitors, that enforce certain policies against the system at runtime. So far, kernel monitor implementations needed to resort to additional levels of hardware support, like virtualization [26, 70], trusted execution environments [5], or runtime checks [3, 86], to enforce these policies against a kernel level adversary. These techniques come with their own set of problems and limitations, such as additional runtime penalties, required changes in system setup and environment, or limited applicability.

CONTRIBUTIONS We propose Page Table Randomization, a novel technique in virtual memory management that allows for a probabilistic defense of page tables. Our approach strengthens the position of the operating system kernel in the presence of a strong local adversary, without requiring any additional hardware support, trusted execution environments, or runtime checks. It comes with the beneficial runtime properties of randomization techniques, and also provides strong leakage resilience. This concept can be used to realize highly efficient kernel monitors. We also present a proof-of-concept implementation of this approach for the AMD64 architecture under the free and widely used operating system kernel Linux. We extensively test and evaluate our research prototype using the popular distribution Debian.

OUTLINE The remainder of this document is structured as follows. In Chapter 2 we introduce basic conceptual and technical notions that are needed to understand the following chapters. We also look at memory protection mechanisms and how an adversary can bypass them. In Chapter 3 we describe our adversarial model, state the problem definition, and briefly analyze different solution approaches. In Chapter 4 we present the design of our solution on a conceptual level. In Chapter 5 applicability of our solution is demonstrated on the basis of our proof-of-concept implementation. Chapter 6 presents extensive tests and evaluation of our research prototype in terms of performance and security. We discuss these results in Chapter 7. In Chapter 8 we give a brief overview of related concepts and technologies, and finally conclude in Chapter 9.

2 BACKGROUND

In this Chapter we give a brief introduction into the basic definitions, and technical background to aid understanding of the concepts we refer to throughout this document. We also look at these concepts from the perspective of an adversary.

2.1 HISTORY OF OPERATING SYSTEM SECURITY MECHANISMS

Early operating systems had little to no separation of processes or users. This was mainly due to the fact, that there was only one user running a single program, that had to be installed physically, e.g., in the form of punch cards [75]. When operating systems supporting multiple users were developed, process separation needed to be introduced. Multiple users using the same machine requires their programs to run simultaneously. To achieve this on uniprocessor systems, time slices are assigned to individual processes, creating the impression of multiple programs executing in parallel for a human observer. One of the earliest examples of this approach was MULTICS, the Multiplexed Information and Computing Service [63].

Processes have to be isolated from each other. The resources assigned to one process must not be used by another process, otherwise the input and output of the running programs might become corrupted. Yet, there needs to be a possibility for processes to interconnect, allowing them to work together through well defined inter process communication (IPC) channels. The operating system also should prevent a process from harvesting and using all the resources, to preserve the functionality and execution of other processes. Additionally, separation between the operating system and user processes is essential in enforcing these properties at runtime. Thus, the kernel requires elevated privileges and access rights. It usually also abstracts the hardware and manages devices,[1] which requires system privileges.

At first, different privilege levels were incorporated into systems via software, but later processors implemented this model of protection in hardware. In order to also control memory accesses, the concept of virtual memory was developed. We will now look into these two concepts in greater detail. Although both are implemented in most major operating systems in some form, with hardware support from many different processors, we focus on the AMD64 architecture and the Linux kernel for the rest of this document.

1 Microkernel designs differ from monolithic kernels in this respect.

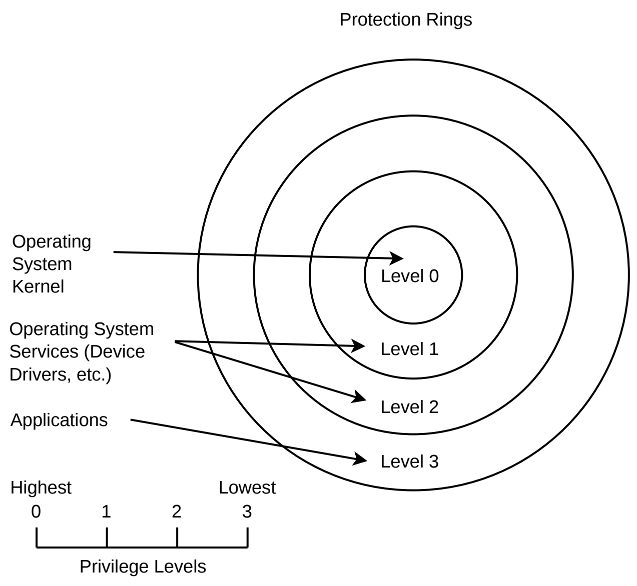

Figure 2.1: Processors implement different privilege levels [43, Fig. 6-3].

2.2 PRIVILEGE LEVELS

In order to separate execution in the kernel from execution in user space the processor implements different privilege levels, as illustrated in Figure 2.1. For the x86 architecture there are four distinct levels, named Ring 0 to Ring 3. Lower levels have access to a larger set of instructions. This means, that privileged instructions may only be executed in Ring 0, which is also called *supervisor mode*, thus the kernel runs on this level. For instance, privileged registers may only be accessed through privileged instructions, which require Current Privilege Level (CPL) 0. Rings 1 and 2 are not used in 64 bit operation, which leaves Ring 3 for userland execution, hence these Rings are also called the *user modes*. For running trivial user programs this may be sufficient, but already basic I/O operations require privileged instructions and therefore cannot be executed by a user space program. In order to request privileged services from the kernel, user space programs issue *system calls*.

An example of this is given in Figure 2.2. The user program issues the *open* system call in order to open a file, which is depicted in ❶. It places the system call number 2 for *open* in the `rax` register and executes the `syscall` instruction, which causes a transition of the CPL from 3 to 0 in the processor. The system then enters the kernel by continuing execution at the address for the system call handler in ❷. The system call handler saves the current process context and redirects execution once more using the system call table. In ❸, the kernel uses the supplied system call number as an offset into the system call table, which lists all the available system calls for the platform. The

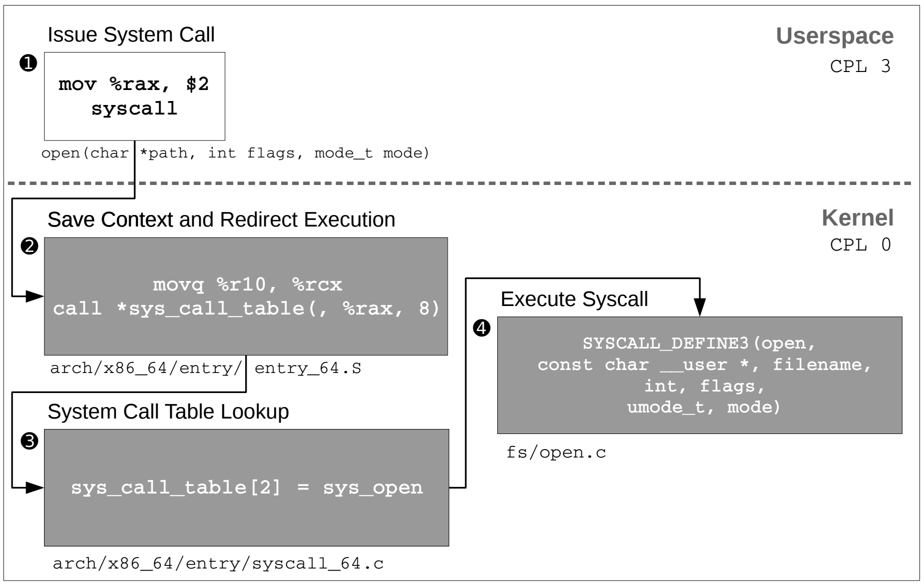
Issue System Call
❶
mov %rax, $2
syscall
open(char *path, int flags, mode_t mode)
Userspace
CPL 3
Save Context and Redirect Execution
❷
movq %r10, %rcx
call *sys_call_table(, %rax, 8)
arch/x86_64/entry/entry_64.S
Kernel
CPL 0
Execute Syscall
❹
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
fs/open.c
System Call Table Lookup
❸
sys_call_table[2] = sys_open
arch/x86_64/entry/syscall_64.c

Figure 2.2: A user program issues the open system call (on AMD64).

entry at index 2 points to a function that implements the open system call in the kernel. In ❹ the *open* system call is executed, i.e., a file is opened or created, and a file descriptor returned to user space.

Usually user programs do not issue the required assembly instructions themselves, but rather use wrapper libraries, such as libc, for convenience. The system call API represents the platform's interface to user space. Bugs in this part of the code open up the kernel for exploitation from user space, often resulting in possible denial of service, information disclosure, and bypass of kernel mechanisms, or worse, privilege escalation and code execution[2]. Around 30 vulnerabilities of these two latter types have been reported in 2014 for Linux, at least five of them in the system call API [92]. It is important to note that the system call interface only represents the *local kernel boundary*, but there are also possibilities for an remote attacker to target the kernel directly without going to user space, e.g., through the network stack[3], or through vulnerable drivers for external devices, since these pieces of code are also part of the monolithic kernel. Finally, there may be known vulnerabilities, which are deliberately not made public [79].

2.3 VIRTUAL MEMORY

The kernel not only has to maintain separate execution contexts, but also manage which parts of main memory are assigned to which pro-

2 For instance, CVE-2013-2094, and CVE-2014-3153, to name two popular examples.
3 Such as, e.g., CVE-2015-3036, and CVE-2015-4001.

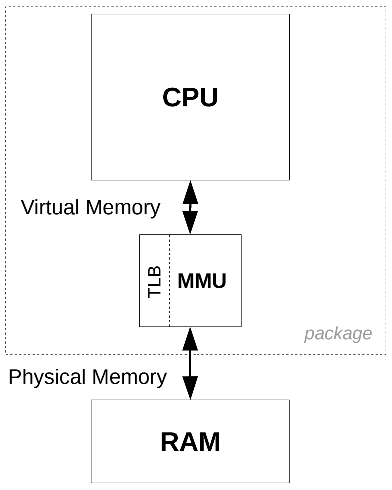

Figure 2.3: CPU memory accesses are translated by the MMU and cached by the TLB.

cess. There may also be parts which are shared between processes and certain parts which no process should be allowed to access. The kernel achieves this by maintaining physical memory through an indirection, a concept which is called *virtual memory*. In order to access physical memory from a virtual memory addresses it first has to be translated.

This indirection enables the kernel to create a virtual address space for every process. The virtual address space consists all possible virtual addresses a process might use. By making use of this indirection, memory accesses in virtual memory can be controlled before being translated to an actual access to physical memory. Virtual memory is implemented on AMD64 through paging, a mechanisms which is supported by a piece of hardware, called the Memory Management Unit (MMU). This is depicted in Figure 2.3. Because programs typically work with a relatively small set of consecutive addresses, which they access frequently,[4] it is sensible to cache the most prominent translations in a fast, small, associative storage, called the Translation Lookaside Buffer (TLB). The basic unit of translation is not an individual address, but a *page*. Pages are contiguous chunks of memory of a fixed size in virtual address space. These pages are mapped to actual physical chunks of memory, which are called *page frames*. The MMU maps the addresses of virtual pages to addresses of physical page frames by looking them up in the *page tables*. It also associates basic access properties with an individual mapping, for instance, if

virtual memory pages correspond to physical page frames

4 More formally, programs exhibit spatial and or temporal *locality*.

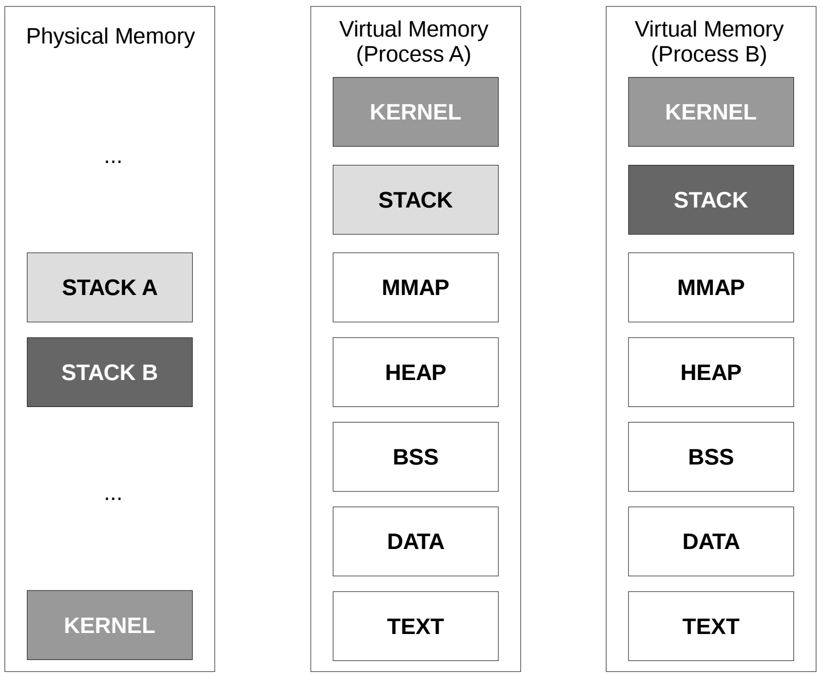

Figure 2.4: The kernel isolates process memory through virtual address spaces, which may contain multiple different VMAs.

the virtual page is mapped as *readable*, *writable*, *executable*, or if the page belongs to the supervisor or a user mode.

Page tables are data objects in the kernel, which are setup and maintained for every process. The way in which the kernel manages the virtual address space of two separate processes is depicted in Figure 2.4. The kernel is mapped as part of the virtual address space of each process. This means, that userland and kernel share the same address space. The user processes A and B also comprise multiple Virtual Memory Areas (VMAs). There are different VMAs for stack, shared mappings (MMAP), heap, static data (BSS), dynamic data (DATA), and code (TEXT) of a process. Each VMA contains a potentially large number of virtual memory pages. The kernel maintains access properties for every page that is mapped into the address space of the current process within its page tables. For instance, the kernel enforces the code pages of a process to be readable and executable, but not writable, by configuring the page table entries for code pages accordingly. In addition to that, processes should not be allowed to access each other's stack memory, or other exclusive, non-shared memory. Hence, the address space of Process A does not include the stack of Process B, and vice versa. More specifically, the page tables of Process A do not contain a mapping for the physical pages of the stack of Process B. This means, that the page tables have to be exchanged when a task switch occurs.

To achieve this, the current execution context is saved upon each task switch, stored in memory as one of the properties of the process, and restored when the process is scheduled again. This context information includes the virtual memory mappings for the user process, i.e., the page tables, which are structured in hierarchical form. In particular, there is one single entry point to the page tables per process. In theory, maintaining a single flat page table for an entire 64 bit virtual address space would require 2^{52} page table entries, because a page table entry manages access properties of an individual page and default 4K pages contain 2^{12} addresses. In practice, only small parts of the overall 64 bit address space are ever accessed by a process. For this reason, the page table is implemented hierarchically, using multiple levels. The number of levels is architecture dependent, x86 deploys four page table levels in 64 bit mode. The current number of levels supported by the Linux kernel is four or less.

page tables are organized as a hierarchy

Figure 2.5 illustrates how the page tables are structured under AMD64. The top level page table is called the Page Global Directory (PGD). The lower levels are connected to this top level in a tree-like fashion. Only the address of the PGD is written to memory as part of the context information of the process, and restored when needed. The physical address of the PGD is loaded into control register `CR3`, which is used by the MMU to locate the page tables in memory. The AMD64 architecture supports three different virtual page sizes, 1G, 2M, and 4K. Only the cases of 4K and 2M mappings are depicted at the top and the bottom respectively. The address at the top belongs to a 4K page and the address at the bottom belongs to a 2M page. The page table structures on each level are 4K in size. This means, that every page table structure contains 512 individual entries, each being 64 bits in width.[5] Translation of a virtual address to its physical counterpart starts at the PGD. This level also represents the entry point to the page table hierarchy of each process, there is exactly one PGD page per process.

To traverse into a lower level, part of the virtual address to be translated is used as an offset into the current level. There can be 512 different Page Upper Directories (PUDs) for a process, and to locate the one responsible for the address currently being translated, the PGD offset is used. This offset represents the index within the PGD, where the address of the PUD for our virtual address is stored. Locating the next lower level, the Page Middle Directory (PMD), is done in the same way using the PUD and PUD offset. Within the page table hierarchy the Page Table Entries (PTEs) represent the lowest level. Each PTE struc-

5 Note, that not all the entries of a level have to be filled.

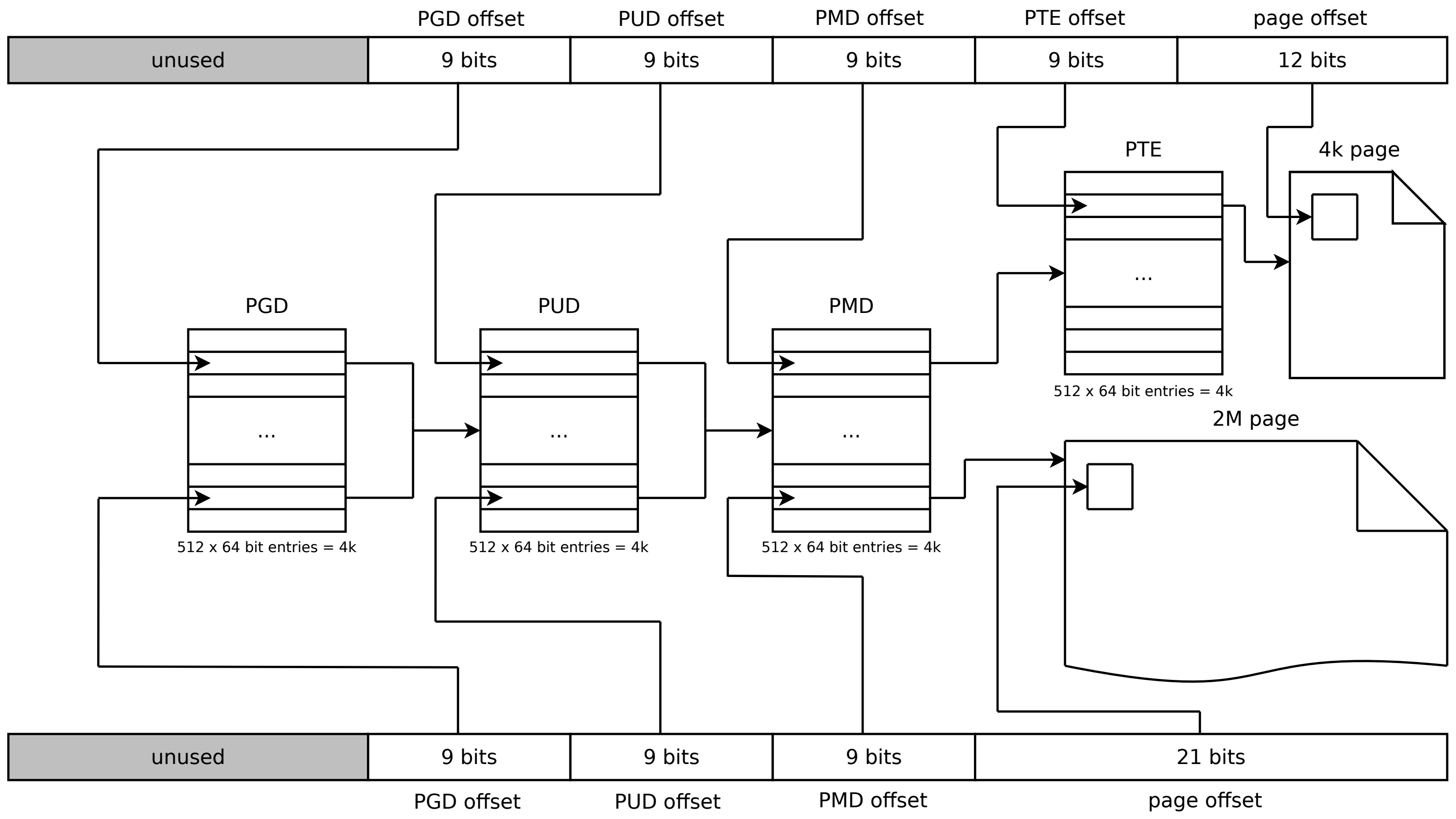

Figure 2.5: 4K and 2M paging on AMD64 [54].

ture manages one page of the virtual memory address space. In the case of a 2M mapping, the PMD page constitutes the lowest level. The contents of the page tables are physical addresses, which the MMU uses for traversing the page tables through direct memory access. In the next section, we give a detailed analysis of how an adversary can walk and subvert these, exploiting the way process virtual memory is managed in the kernel.

2.4 RUNTIME ATTACKS AND EXISTING DEFENSE APPROACHES

Although being known for nearly three decades, runtime attacks in general are a continuous threat to modern software, i.e., application- and system-level code. These attacks are based on the exploitation of various software bugs, such as format string vulnerabilities, stack- and heap-based buffer overflows, user-after-free, dangling pointers, and integer overflows. They can be classified into code-injection [8, 24], code-reuse [72, 81], and data-only attacks [13, 17]. Consequently, a range of architectural defensive measures like NX [58], IOMMU [1], SMEP [33], and SMAP [43] have been introduced to modern processors. Increasing deployment of these countermeasures requires adversaries to resort to more sophisticated techniques in kernel exploitation, such as kernel-level Return Oriented Programming (ROP) [38] and ret2dir [45] to achieve their goals. As introduced in Chapter 1, existing defenses that aim to strengthen kernel integrity and mitigate runtime attacks include kernel monitors. However, an attacker can compromise the page tables through a sophisticed data-only attack on the kernel, bypassing all memory protection. This is why existing kernel defenses [3, 5, 21, 26, 70, 86] require additional hardware capabilities, such as hypervisors, or trusted execution environments and resource-heavy runtime checks. A more detailed overview of existing kernel defenses is provided in Chapter 8.

2.5 A DATA-ONLY ATTACK ON PAGE TABLES

We will now focus on why manipulating kernel memory management is a worthwhile effort from the perspective of an adversary, and how this also affects the design of kernel monitors. In particular, we describe how an adversary can exploit kernel data objects and the direct mapping, to subvert the page tables of a process. This allows the attacker to overcome and completely bypass all the access restrictions the kernel enforces. Note, that we present just one particular example, of how this exploit can be constructed. In the following, we assume an adversary located in user space, who is equipped with a memory-corruption vulnerability in the kernel. This allows the attacker to read and write arbitrary kernel memory. While different exploit strategies are possible in this scenario, we will show how to construct a data-only attack on the page tables. This means, that the attacker does not inject malicious code, or violate the kernel's control-flow graph, also bypassing potentially deployed kernel monitors, such as kernel-level Control-Flow Integrity (CFI).

2.5.1 *Exploiting the Direct Mapping*

As briefly explained in the previous section, each user process is associated with its own hierarchy of page tables. The kernel page tables are linked into the page table hierarchy of each process at the PGD level,[6] thus the virtual mappings for the kernel are the same for every execution context. In contrast to the user processes, the kernel has to manage parts of memory that processes are not allowed to access. For instance, it must access physical memory to allocate and deallocate memory dynamically. For this reason, the kernel maintains the direct mapping, which is also called *one-to-one-*, *id-* or *physical-* mapping (cf., Figure 2.6). In this area in virtual memory, all physical memory is mapped as one contiguous, linearly addressable block. The direct mapping allows the kernel to address physical memory, while operating from virtual memory. This is required, because once paging is enabled during boot, the kernel only has access to virtual memory and cannot access physical memory otherwise.

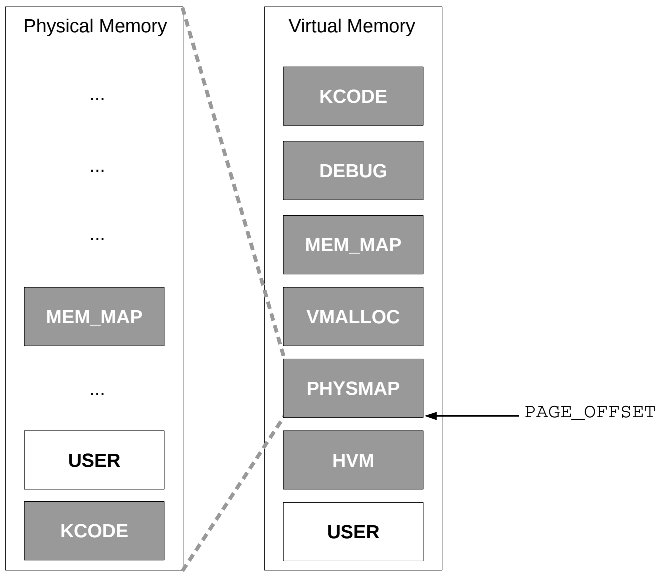

Figure 2.6: The kernel maintains the direct mapping to manage physical memory from virtual address space using a fixed base address.

6 Just like the execution context of a user process, the kernel context operates in virtual memory.

Using the direct mapping, physical memory accesses can be translated to virtual memory accesses by adding the base address of the physmap area (i.e., `PAGE_OFFSET`) to the respective physical address. The attacker can exploit this mapping to translate physical memory addresses to virtual addresses. On AMD64, every physical page frame is mapped within the physmap area at all times. This also means, that there may be multiple additional virtual mappings for the same page frame, possibly using different access properties. For instance, while process code pages may be mapped as readable and executable in user space, the same page frames are mapped as writable in the physmap area within kernel memory.[7] Moreover, the page tables are mapped as writable in the physmap area as well.

2.5.2 *Leaking Page Table Addresses*

We now know how the kernel uses the direct mapping to access physical addresses from virtual address space. Because all of physical RAM is mapped within the direct mapping, this is also the case for the page tables of a process. For an adversary, locating the page tables means finding the entry point to the page table hierarchy of the target process. As explained in Section 2.3, this represents the address of the PGD of that process, which is kept in control register `CR3`. This register is privileged and requires CPL 0. Therefore, the attacker would have to redirect kernel execution to spill the register to memory, or pass it to user space. However, this would violate the control-flow graph of the kernel.

Luckily, the virtual counterpart to the physical address that `CR3` holds for every process is also stored in kernel memory directly. This is, because the kernel must be able to restore the page tables of a process, before a process should be scheduled. Processes are represented as data objects of type `thread_info` in the kernel. This is depicted in Figure 2.7. More specifically, the `pgd` field of the `mm_struct` stores the virtual address of the PGD of a process. The `mm_struct` pointer is stored within the `task_struct`, wich in turn is referenced by the `thread_info` struct of the process. This means, that in order to find the page tables for a process, an adversary needs to locate the `thread_info` struct. The `thread_info` struct can be located from one of the stack addresses of the target process by masking it with a fixed value.[8] Generally speaking, there is a lot of additional information contained in these structs, e.g., the address of the task's stack is also part of the Task State Segment (TSS) for loading the register of

7 This address aliasing is exploited in the ret2dir kernel exploit technique [45] to bypass Supervisor Mode Execution Prevention (SMEP), which forbids to redirect execution to a user space address when in supervisor mode.

8 That is, `someStackAddress & ~(THREAD_SIZE-1)` yields the current `thread_info` address [16, Line 612].

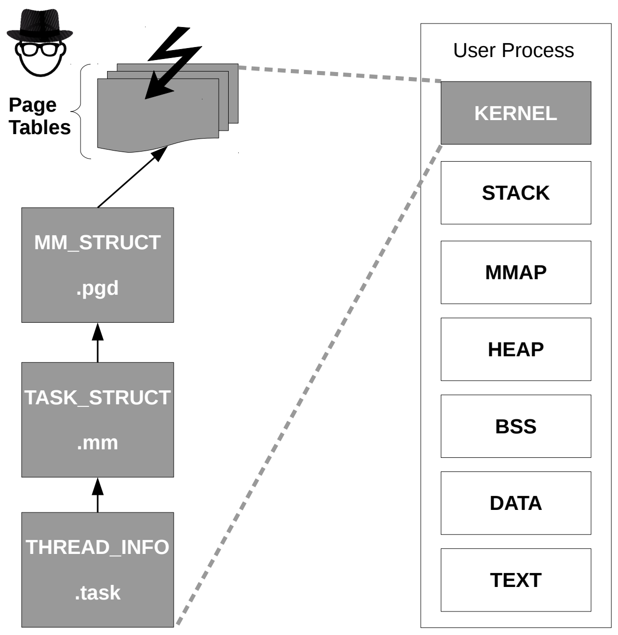

Figure 2.7: An adversary can locate and modify the page tables for a user process in the kernel through a sophisticated data-only attack.

the stack pointer `RSP0`. But for locating the page tables the depicted information is sufficient.

2.5.3 *Maliciously Modifying Page Tables*

At this point, the attacker is equipped with read and write access to the kernel, and located the page tables via the `thread_info` struct. The `pgd` field of the `mm_struct` contains a virtual address mapped from within the physmap area to the physical page frame containing the PGD of the target process. Therefore, the attacker is now able to traverse the page tables in software for a page to modify its access permissions. The values of individual entries within the page tables represent physical addresses combined with flags describing access properties. Thus, an adversary needs to make use of the direct mapping while performing the page walk. For every physical address encountered, the corresponding physmap address can be calculated by adding `PAGE_OFFSET` as a base address. In contrast to the physical address, this results in a virtual address, which the attacker can access directly. This manual translation is necessary at every level of the hierarchy, to descend into a lower level. In the end, it will yield the page table entry responsible for the target page. The attacker is then able to maliciously modify the access properties of the page directly by setting its permission bits at will.

2.6 CONCLUSION

In this Chapter we explained in detail, how a kernel-level adversary can exploit kernel memory management data to subvert and completely bypass its security guarantees. There are examples of real-world exploits and attacks presented in academia, which are using this approach [16, 31, 53, 76]. It is important to note, that this attack scenario is possible even if the kernel is build as relocatable and its base address is randomized. The problem is, that randomizing the kernel image's base address does not eliminate the direct mapping from physical to virtual addresses and the base address of this mapping is never randomized. Being able to write to kernel space and locate the page tables enables an adversary to modify the access properties of any page frame on the system. This includes removing the write protection of code pages, setting the executable flag on data pages, and creating new entries to gain access to previously unmapped pages.

Existing kernel monitors, that are based on enforcing control flow, or code integrity, cannot mitigate such data-only attacks on the kernel. Furthermore, these defenses require the integrity of page tables to enforce their policies. This is why many of the proposed solutions leverage additional hardware support like virtualization, trusted execution environments, costly runtime checks, or combinations thereof. In the case of using a hypervisor to ensure integrity of kernel page tables, the hypervisor page tables still need to be protected. Because integrity of page tables cannot be guaranteed by existing kernel monitors, we will state an explicit problem defintion in the next Chapter to systematically address this open problem.

3 PROBLEM DEFINITION AND ANALYSIS

In the previous Chapter, we described how a user space adversary can completely bypass the enforcement of memory protection, given a memory-corruption vulnerability that grants read and write access in the kernel. In this Chapter, we present a detailed adversarial model in Section 3.1. We investigate the root causes of the problems of memory protection and discuss open challenges in Section 3.2. We explain possible defense approaches to ensure integrity of page tables natively in the kernel in Section 3.3. Finally, we state the requirements for an efficient and effective mitigation scheme in Section 3.4.

3.1 ADVERSARY MODEL

For our work, we consider a strong adversary model which is consistent with previous defensive work in this field [5, 26, 70, 86]. We assume access restrictions to be enforced by the MMU, and architectural defenses to be active. More specifically:

- We consider an adversary who has control over a user process.
- The kernel is assumed to contain at least one memory corruption vulnerability, which allows arbitrary read and write accesses within (kernel) memory.
- We assume the access restrictions, such as a non-executable direct mapping, or a read-only kernel code section, to be enforced by the MMU. This requires the adversary to modify page tables to overcome these restrictions.
- The adversary does not have access to the system during boot.
- We assume Supervisor Mode Protections, such as SMAP [43] and SMEP [33], to be present and activate.
- The adversary does not have access to physical memory from software. This is the case for current hardware with appropriate configuration [1].

Timing-, cache-based, and other side-channel attacks are out of scope for our work. We will now look at the open challenges to mitigate malicious modification of page tables by such an adversary.

3.2 OPEN CHALLENGES

In Section 2.5, we demonstrated an attack on kernel data objects, which results in malicious modification of the page tables. However, this only represents one example of a whole class of attacks, that aim for the subversion of memory protection. These data-only attacks use a combination of malicious read and write accesses within kernel memory to manipulate memory protection of individual pages. The overall challenge is to keep an adversary, who is equipped with read and write access to kernel memory, from modifying the page tables, which are a critical building block for the enforcement of memory protection. At the same time, benign changes must be possible by the operating system. As briefly introduced in Section 2.4, previous work needed to rely on additional hardware requirements or runtime checks to achieve this.

Our analysis of the attack scenario shows that there are two technical challenges, which can be resolved within the operating system kernel to disallow compromise of the page tables through data-only attacks. First, the entry point to the page tables of a process is present within the address space of a process. In particular, we explained how an adversary can disclose the entry point to the page table hierarchy of a process with the help of different kernel objects. Second, the direct mapping of the kernel makes physical memory, and thus addresses used in entries of page table structures, available to an adversary. Knowing that the direct mapping always starts at a fixed address, the corresponding virtual address in kernel space can be easily calculated. This allows an adversary to walk the page tables, retrieve the PTEs, and modify access properties, or add mappings of any virtual address deemed useful.

A scheme providing a solution to these challenges will enhance kernel memory protection in the presence of a strong local adversary. Such a mechanism also serves as an enabling technology for the implementation of highly efficient kernel monitors, because it provides a secure memory domain for the monitor within the kernel. Note, that kernel monitors like control-flow integrity aim to solve the challenges posed by code-reuse attacks, whereas the challenges we identified are caused by data-only attacks. This means, that monitor implementations enforcing, e.g., control-flow integrity in the kernel cannot mitigate these attacks, because they do not violate control flow. Nonetheless, a design solving the identified challenges can be used in combination with a kernel monitor, to also provide a defense against code-reuse attacks. We now look into possible defense approaches to solve the open challenges we identified and briefly assess their advantages and drawbacks.

3.3 POSSIBLE DEFENSE APPROACHES

There are multiple ways of resolving the identified challenges by adjusting the current design of virtual memory management in the kernel. We present three basic alternative strategies to strengthen the kernel's memory protection mechanism.

read-only mappings

First, the pages containing page table structures could be mapped as read-only. For this, the memory management functions in the kernel would have to be instrumented to make the pages writeable prior to any modification and mark them as read-only again afterwards. This would prevent direct, malicious modification to page tables. But it would still allow an adversary to perform a page walk and leak information about the page tables. Additionally, this approach has the disadvantage that page table mappings have to be modified for every operation involving the page tables.

encrypted page table structures

A second approach is to encrypt the page table structures contained in the direct mapping and dynamically decrypt individual structures when they need to be read or updated. This would prevent malicious modifications and additionally prevent an adversary to leak information about the page tables. However, there are several issues. The key and encryption process would have to be managed outside of main memory, so that its code and data cannot be compromised. While this is in theory possible, this approach requires the MMU to decrypt pages as well. Even if current MMUs included this functionality, this approach adds runtime overhead of decryption to every page table operation in the kernel, i.e., read and write operations.

randomized page table pages

A third approach is to randomize the location of page tables. This would render malicious modification infeasible and also prevent disclosure of any information related to page tables. Additionally, this would not add any runtime overhead to memory management operations involving page tables, regardless if these are read or write operations. We therefore select this third approach to secure page table integrity. We will now list the requirements for a design based on randomizing page tables.

3.4 DESIGN REQUIREMENTS

Solving the open challenges we identified, to provide enhanced security guarantees, means that some specific design requirements have to be met. There are also several technical challenges when randomizing page tables, which need to be addressed. A design incorporating randomized page tables in the operating system kernel must meet requirements to compatibility, performance, and security.

COMPATIBILITY An implementation of the defense approach must be possible within the existing virtual memory management infrastructure the kernel provides. If the location of page tables is randomized, the kernel still needs to be able to locate the page tables to perform benign changes. While using the direct mapping will still be possible for regular pages, we need to provide the kernel with means to perform a translation from physical to virtual page table addresses. Additionally, 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.

PERFORMANCE Page table management is one of the critical tasks the operating system has to perform, when creating or terminating processes, setting up shared memory regions, or allocating memory dynamically. These tasks should not suffer from severe performance degradation. The translation from physical to virtual addresses without requiring a page walk in software is a key technique to perform maintenance work efficiently. In order to achieve realistic performance, our approach has to provide the kernel with some means to achieve this as well. Additionally, the runtime overhead for scheduled processes must be kept at an absolute minimum.

SECURITY A solution must provide an environment, where a kernel-level adversary is not able to modify page tables directly, and should prevent disclosing information about the page tables. An adversary must not be allowed to learn their new, randomized mapping. While randomization approaches are well known for their high efficiency, they have been shown to be vulnerable to a range of attacks [25, 71, 73, 74].

In particular, memory disclosure vulnerabilities and brute-force attacks represent two main threats to these approaches. Memory-disclosure vulnerabilities allow an adversary to leak information about code- or data-pointers [73]. If randomization is applied to a range of addresses in memory, a single leaked randomized addresses potentially compromises all other addresses in this area.

Brute-force attacks threaten randomization approaches, that suffer from low entropy [71]. This means, that randomization approaches with low entropy are ineffective, because the offset used for randomization can be guessed with high probability. Therefore, a defense mechanism using randomization has to provide resilience against memory disclosure, and enable a high level of entropy.

3.5 SUMMARY AND CONCLUSION

In this Chapter we analyzed the problems of kernel memory protection mechanisms and identified the key challenges not yet tackled by existing defense approaches. We also explained three different basic approaches to ensure the integrity of page tables and selected the randomization approach for our design. Finally, we stated the requirements for an efficient and effective defense mechanism against malicious modification of page tables through data-only attacks. Based on the principle idea of randomizing page tables, we propose a novel, probabilistic defense approach to guard memory management in the kernel, without having to resort to hardware virtualization or policy-based runtime checks.

Instead of raising the requirements towards hardware capabilities, or the assumption of the completeness of its policies, its protection guarantees directly depend on resilience to memory disclosures, and the amount of entropy available. Like other randomization approaches it therefore offers potentially low overhead and high practicality. However, there are multiple design requirements, that have to be met. We address the listed specifications and how we realize them in the following chapter.

4 PAGE TABLE RANDOMIZATION

In this chapter we give detailed insight into the design of our solution to solve the challenges we identified in the previous chapter.

4.1 CONCEPTUAL OVERVIEW

The basic idea underlying our solution involves three steps and is depicted in Figure 4.1. The attacker has read and write access to kernel memory, and our goal is to mitigate data-only attacks on the page tables in this scenario. Hence, we disable the one-to-one mapping between physical and virtual addresses in the kernel for page tables in Step ❶.

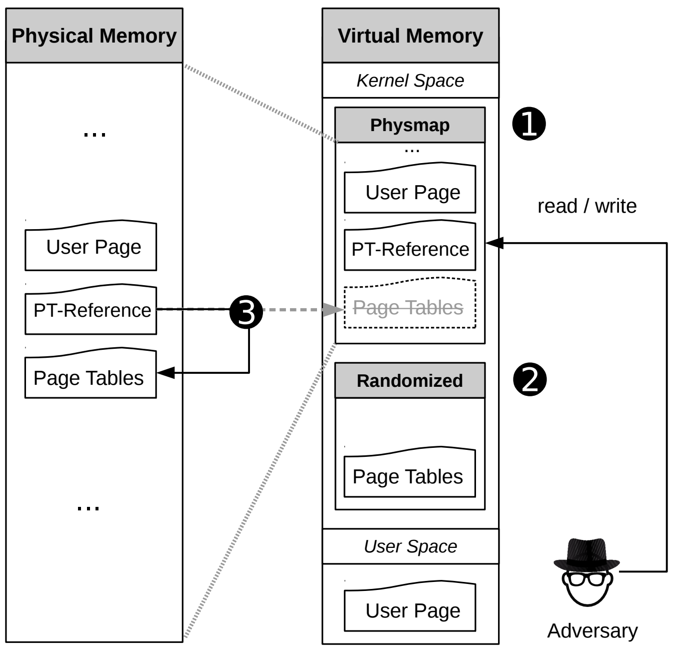

Figure 4.1: Our approach removes the page tables from the direct mapping and randomizes their new location in virtual address space, effectively hiding them from an adversary.

This allows us to relocate the corresponding parts of physmap to a different location in virtual memory. Instead of including the physical pages that contain page table structures in the physmap area, we devise another large memory area for these pages in virtual memory in Step ❷. Because we relocate all the page table pages to this new area, we are able to randomize their addresses by changing the base address of this area on every boot. This base address will be chosen perfectly at random prior to launching the first process.

Finally, we replace all references to the page table hierarchy with their physical addresses in Step ❸. Because we randomize the base address for the new virtual mapping of the page tables, an attacker is not able to translate these physical addresses to their virtual counterparts anymore. Hence, these references can be written to main memory without violating our security goals. The benefit of this solution is that there is practically no overhead involved, because we only modify an offset in the translation of physical to virtual addresses. Enabling Page Table Randomization involves changes to virtual memory management in the Operating System kernel. We describe these changes in the next sections.

4.2 RANDOMIZING PAGE TABLE PAGES

Page tables are data objects that are dynamically allocated in the kernel at runtime. These objects are provided by the page allocator, which is a central, low-level facility in the kernel that manages physical pages. However, at early boot time this facility is not yet available, hence the kernel includes static page tables, which are used during this phase. To achieve randomization of page tables, our design includes modifications to the kernel, that allow the dynamic allocation of individually randomized pages, and the relocation of existing page tables, such as the static boot time page tables.

This is also depicted in Figure 4.2. First, we generate our randomization secret, i.e., the base address of the randomized region at boot time, which we store in a privileged register. Second, we relocate all existing static page tables. Afterwards, our modified page allocator can provide regular pages or randomized pages dynamically. In the case of regular requests, the virtual address of the page will be located in the physmap area. If the allocation request is a page table allocation, the virtual address of the page will fall into the randomized area. In both cases the virtual address is given by the physical address of the page added to the base address of the respective memory region. We instrument the kernel to replace all page table references in memory with their physical addresses, and translate these accesses to virtual addresses using the privileged register which holds our base address.

Note, that randomizing the start address of the entire direct mapping might seem like an alternative to only randomizing the page tables. However, there are multiple subsystems in the kernel that would expose the location of known kernel objects and leak the randommization secret. A randomized physical mapping is therefore trivially vulnerable to information disclosure. This is why we need to make sure that only page table allocations are randomized. All other allo-

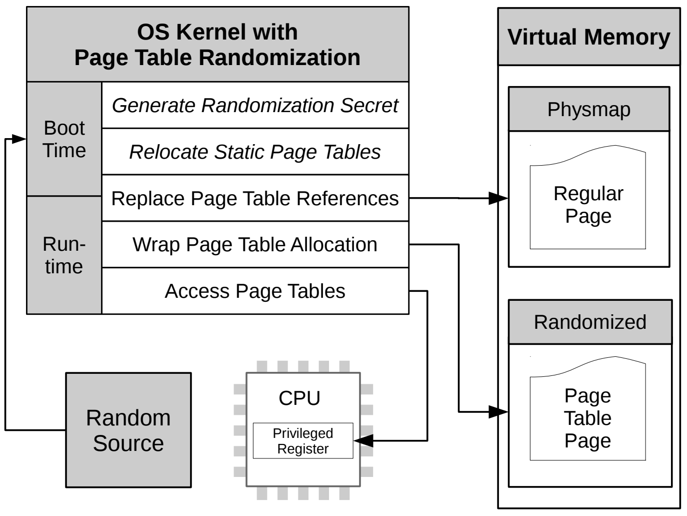

Figure 4.2: We initialize Page Table Randomization during early boot, relocate static page tables, and distinguish between regular and page table pages for dynamic allocation in the kernel. We keep our randomization secret in a privileged register, and replace references to randomized virtual addresses with their physical addresses in memory.

cation requests in the kernel should remain unchanged. Additionally, freed, randomized pages should not be reused for other purposes. There are multiple ways to achieve this, e.g., the kernel can use two separate allocators, one for the page table pages and one for all other pages. A second way is to provide a wrapper around the standard kernel allocator, that randomizes and derandomizes pages, and use this wrapper for page table allocations. A third way is to extend the standard allocator to provide regular and randomized pages upon request.

4.3 ACCESSIBILITY OF PAGE TABLES

Without the one-to-one mapping of the page table pages, the kernel will no longer be able to locate them when performing a page walk in software. As explained in Section 2.5.3, it would usually add the base address of the direct mapping to a physical address to retrieve its virtually mapped counterpart. Using the direct mapping will still work for regular pages. However, for randomized pages we need to provide the kernel with means to perform this translation efficiently as well. This is necessary, because the translation of physical to virtual addresses is a low-level operation that is performed frequently during maintenance work. We modify virtual memory management code in the kernel, to read the randomized base address from our privileged register when calculating the virtual address of a page table page. This requires the kernel to differentiate between pages containing page table structures and all other pages. Thus, we mark such physical pages in the kernel.

4.4 RESILIENCE TO INFORMATION DISCLOSURE

It is not possible to read or write hardware registers by accessing memory. Thus, the contents of our base register cannot be leaked through a data-only attack. Additionally, an adversary operating from user space is not able to read the contents of privileged registers. That is why we keep the base address to our randomized area within a privileged register. This enables us to allow benign kernel code to access the page tables, but at the same time prevent a kernel-level adversary from accessing them.[1] We need to ensure that the randomized base address is never leaked. To achieve this we avoid writing it to main memory. This also means that our solution is not compatible with hibernation (suspend-to-disk) and sleep-mode (suspend-to-RAM). Both kernel features cause the randomized base address to be written to main memory, which would create a potential leak.

Furthermore, we need to make sure that the base address of our new region is not placed on the stack, e.g., through local variables. This can be achieved by instructing the compiler to prevent spilling local variables that we flag accordingly. Whenever the base address is needed to read, create, modify, or clear page table entries, we keep the resulting randomized addresses in such flagged local variables, which are then held within registers during function execution. We also make sure, that the contents of the register can be read by the kernel, but not be trivially leaked by a ROP-adversary. To achieve this, the kernel image must not contain a gadget spilling the register containing our randomized base address. Therefore, we avoid placing such a gadget in the code, by making use of inline assembly routines.

4.5 ENTROPY FOR RANDOMIZATION

In order to avoid collisions with existing virtual mappings, we populate one of the currently unused regions in the 64 bit virtual address space for the relocation of page table pages. The area we choose additionally has to be large enough to enable a high level of entropy and withstand brute-force attacks.

Providing enough entropy for randomization is closely related to the way the kernel manages the overall virtual memory layout of the platform, which depends on the kernel implementation. However, for

1 The idea to hide information through utilization of privileged registers was leveraged before, e.g., by TRESOR [59] for implementing AES encryption outside of main memory to thwart cold-boot attacks.

64 bit architectures this should not represent a problem. For instance, unused areas exist within the memory layout of the Linux kernel under AMD64 [4] and AArch64 [57, page 12], which we can utilize to perform randomization providing high levels of entropy. For 64 bit platforms without such unused memory regions, we could reuse occupied regions usually used by hypervisors or debugging facilities.

Note, that we use one randomized base address for all processes at runtime. This means that we do not have to change the contents of our base register during a context switch. It also has the advantage of not having to store the base addresses in main memory as part of the task context. This is a design choice and does not represent a limitation in terms of entropy. It would be possible in theory to have different offsets per process while still only using one register, e.g., through a table look-up. However, this would also add one further indirection and thus potentially introduce additional runtime overhead, which is why we focus on the simpler approach presented here.

4.6 summary

In this chapter we presented the conceptual idea and overall design of our solution to the integrity protection of page tables. We propose several modifications to include our design in the virtual memory management subsystem in the kernel. Our approach randomizes the location of page table pages to prevent an attacker from locating any page table entries and modifying access properties of memory pages through data-only attacks. The presented design incorporates a high level of entropy by leveraging a source for random numbers and the 64 bit address space. We make use of a single, privileged register to provide protection against information disclosure by a local adversary. In the next Chapter we will describe our proof-of-concept implementation for the AMD64 architecture under Linux. We also take a look at some technical challenges that have to be solved for implementing Page Table Randomization.

5 IMPLEMENTATION

In this Chapter we describe the implementation of our research prototype for Page Table Randomization.

5.1 OVERVIEW

For our proof-of-concept implementation the randomization facility in the kernel was designed as a wrapper around the standard page allocator, although other implementations are of course possible. The relevant parts of the kernel are depicted in Figure 5.1, new components are highlighted in gray. An advantage of the wrapper approach is, that the new functionality is only used from the call sites we place in the code. We can therefore easily verify that only page table allocations use the randomization wrapper.

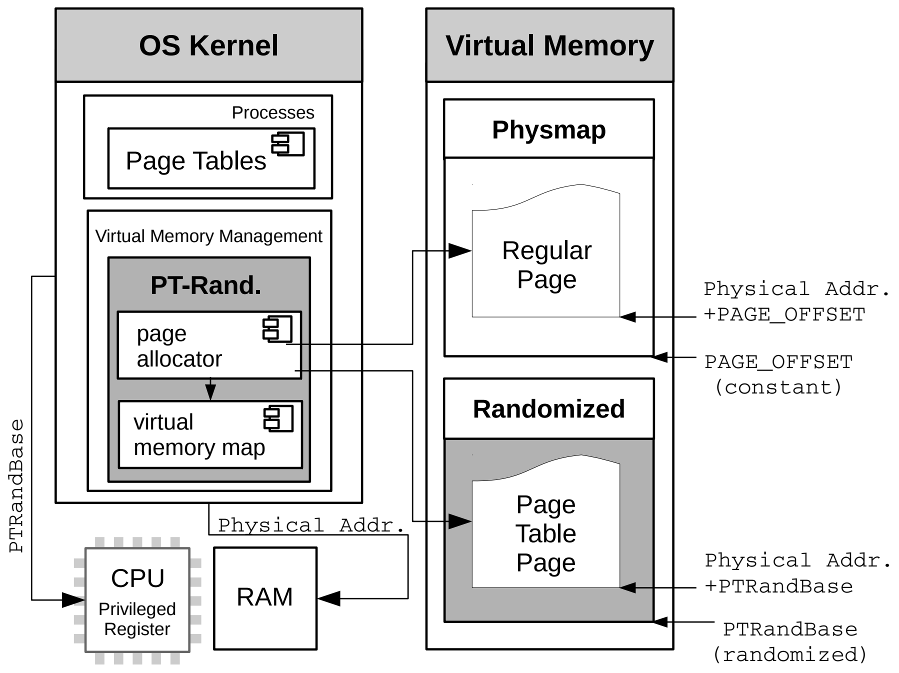

Figure 5.1: Key components in our implementation of Page Table Randomization.

Additionally, existing code must be modified to write the physical instead of the virtual addresses of page table pages to memory and to translate these physical addresses to their virtual addresses using the base register. We modify kernel code for process creation and virtual memory management. Moreover, randomized pages are marked in the virtual memory map to differentiate them from regular pages.

The memory map is an array of `struct page` metadata objects, each describing properties of a physical page. This array is used by several kernel subsystems, such as the page allocator. This marking is necessary for calculating the virtual address from a physical address at runtime, because for regular pages physmap is still used, while for randomized pages the base register is read. Overall, our patch includes changes to 63 files, with 1586 insertions, and 179 deletions.

5.2 INFRASTRUCTURE AND KERNEL ENVIRONMENT

We conduct our research based on the Linux kernel, i.e., Linus' tree in version 4.2.[1] The main directory for development is `arch/x86`, although some of the changes also affect more general parts, such as `kernel`, `fs`, or `mm`. The virtual memory subsystem of the kernel and the `arch/x86` specific memory management code are rather stable, still there are occasional changes, e.g., to include optimizations, improve the structure, or documentation. One of these changes included the configuration feature `X86_DIRECT_GBPAGES`, which allows making use of 1G pages in the direct mapping.[2] By default, the kernel makes use of 2M pages for the direct mapping. Recall, that this maps all of physical memory, which potentially represents a huge amount of memory (cf., Section 2.5.1). The more memory mapped, the more PTEs needed to set up the page tables for this mapping, thus using 1G or 2M pages instead of the standard 4K reduces the space overhead for maintaining the direct mapping.

5.3 CONFIGURATION CHANGES

There are several configuration requirements, that we need for our patch to take effect. Some of the important configuration settings are given in Listing A.1. In particular, we make use of kASLR, which relocates the base address of the kernel code and static initialization data in physical memory. This randomizes the boot time kernel page tables, which become part of the kernel page table hierarchy later at runtime. Additionally, the `/dev/mem` and `/dev/kmem` character devices exposing physical and kernel memory to user space are not included in our build, as they would violate our assumption, that the attacker has no access to physical memory from software (cf., Section 3.1). Yet, it might be necessary to include `/dev/mem` for setups requiring a graphical interface through Xorg. The `STRICT_DEVMEM` setting allows

1 The repository at `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`, commit `64291f7db5bd8150a74ad2036f1037e6a0428df2`.

2 These 1G mappings are sometimes called *huge pages*, 2M mappings being called *large pages*. Support for these has been in the kernel for a long time.

including the `/dev/mem` device, while restricting accesses to the necessary parts.

Furthermore, we built the kernel with the `PANIC_ON_OOPS` option enabled, which causes the machine to halt, if a kernel error is detected. This is important, because our adversarial model allows read and write accesses to kernel memory. Enabling this option disallows the adversary to systematically scan kernel memory, because a single bad access stops execution.

We disable `WANT_PAGEVIRTUAL` as well. If enabled, this causes the kernel to include a pointer to the virtual memory page in the meta data of the physical page frame. This would make our randomization efforts meaningless. Yet, for architectures making use of high memory, such as x86, this setting might be necessary, and implementing our approach for these architectures would require some modifications of this feature. However, for our current implementation we simply avoid it.

In addition, there are several settings related to huge page support we need to change. These changes are located in `arch/x86/Kconfig`. Our modifications are listed in Listing A.2. Note, that `PHYSICAL_ALIGN` and `RANDOMIZE_BASE_MAX_OFFSET` together directly control the amount of entropy available for kASLR. Maximizing kASLR entropy with respect to these two values means decreasing the alignment restrictions, while increasing the randomization offset. All other settings are related to the huge page support in the kernel, which we disable for our implementation to avoid unnecessary splitting of huge pages upon randomization.

5.4 RANDOMIZING INDIVIDUAL PAGES

Listing 5.1 shows part of the official documentation of the Linux kernel, that has an overview of the current memory layout on AMD64 from low to high addresses. The memory area used for randomization should provide enough space for the page tables of all processes and a reasonable amount of entropy for randomization of the base address. Therefore, we select one of the 40 bit holes as our region of choice. This provides us with 1 TiB of address space to work with for the allocation of page table pages. The total amount of entropy is limited to 28 bits when using this region, because we implement the randomization on a per-page basis. Our wrapper implementation performs randomization and derandomization. Allocation and freeing on the other hand is delegated to the page allocator, which is also called the *zoned buddy allocator*.

Randomizing a page in virtual memory involves three steps. First, we choose and set up the new location. The address of the new location will be given by the physical address of the page, added to the base address used for randomization. Mapping this new location

Listing 5.1: The AMD64 virtual memory map for Linux with four level page tables [4].

```
0000000000000000 - 00007fffffffffff (=47 bits) user space
hole caused by [48:63] sign extension
ffff800000000000 - ffff87ffffffffff (=43 bits) hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB)   direct mapping
ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
ffffea0000000000 - ffffeaffffffffff (=40 bits) vmemmap
... unused hole ...
ffffec0000000000 - fffffc0000000000 (=44 bits) kasan
... unused hole ...
ffffff0000000000 - ffffff7fffffffff (=39 bits) esp fixup stacks
... unused hole ...
ffffffff80000000 - ffffffffa0000000 (=512 MB)  kernel text
ffffffffa0000000 - ffffffffff5fffff (=1525 MB) modules
ffffffffff600000 - ffffffffffdfffff (=8 MB)    vsyscalls
ffffffffffe00000 - ffffffffffffffff (=2 MB)    hole
```

requires walking the page tables for the new address and modifying or creating the PTEs to point to the intended physical page frame. Second, we remove the old virtual mapping within physmap. Otherwise, the page will be doubly mapped, which we need to avoid for randomized pages. Thus, a second page walk is necessary to retrieve the entry for the old virtual address and clear its mapping. Third, we replace all the references using the page with their new virtual addresses. This is necessary for all addresses in the range of the old virtual mapping. Listing 5.2 shows an example routine from our implementation achieving the first two steps. Here, `_dgs_calc_randmap` calculates the new, randomized location from the old virtual address of the page, `_dgs_vaddr_alias` sets up an additional mapping in virtual memory for the given `kaddr` at the new `raddr`, and `__SetPagePGT` sets the page's randomization bit flag in the memory map. Finally, the old mapping is cleared and the TLB flushed.

Remapping an existing page in this way during runtime requires knowledge of all the references to the page that is to be moved, which is generally non-trivial. Fortunately, for page table pages the number of references is limited, because they are organized hierarchically, using physical addresses internally. In addition, the only pages with existing references we need to move are the kernel page tables, because the process page tables will be randomized at the time of creation. However, we still need to move pages that have no references, i.e., free pages that we request from the page allocator. The page allocator

Listing 5.2: Randomizing a memory page

```
static inline void *_ptrand_randomize(void *p)
{
        struct page *ppage;
        pte_t *old_pte, *new_pte;
        unsigned long kaddr, raddr;

        if (is_pgt_rand(p))
                return p;
        ppage = virt_to_page(p);
        kaddr = (unsigned long)page_address(ppage);
        raddr = _dgs_calc_randmap(kaddr);
        if (_dgs_split_large(kaddr))
                return NULL;
        if (_dgs_vaddr_alias(kaddr,raddr,&old_pte,&new_pte))
                return NULL;
        __SetPagePGT(ppage);
        native_ptep_get_and_clear(old_pte);
        flush_tlb_kernel_range(kaddr,kaddr);
        flush_tlb_kernel_range(raddr,raddr);
        return (void *)_dgs_calc_randmap(p);
}
```

of the kernel returns the start address of the page. This start address falls within the physmap area. The direct mapping makes use of 2M pages by default and can be configured to even use 1G pages. This causes problems, when moving the allocated page, because its 4096 bytes are mapped as part of a larger 2M or 1G page, thus randomizing the huge page mapping containing this page will move a whole lot of other virtual addresses as well. We tackle this problem by splitting the huge page into standard 4K pages, to move only the free page the allocator returned.

5.5 THE ROLE OF VIRTUAL MEMORY MAP

We also already mentioned, that the kernel has to distinguish between randomized and regular pages sometimes. For instance, when doing a page walk in software, traversing the lower levels is done by taking the physical address referenced from a higher level entry and calculating its virtual address. This has to be possible at runtime, and at the lowest possible cost, because a page walk is already considered an expensive operation.

For this reason, we introduce a new bit in the `flag` field of the `struct page` type defined in `include/linux/mm_types.h`. In 64 bit mode, this field is eight bytes wide and has between 20 and 32 occupied bit positions, depending on the configuration, which leaves some room for our custom flag. When a page is randomized this flag is set, when it is derandomized it is cleared again. We make use of this flag at multiple locations, such as our implementation of the `__va` macro. When it is set, we read the base address of our randomized area from the privileged register and add the physical address of the page as an offset. Otherwise we use the standard `PAGE_OFFSET` as base address into the physical mapping.

There are some challenges when implementing this approach. The virtual memory map has to be populated by the kernel during early boot, which means that randomizing pages is not possible prior to its initialization. Hence, early boot code can not make use of our modified implementation of the `__va` macro, which needs vmemmap to be populated. Fortunately, there already is an second version of this macro in the sources, namely `__boot_va`, although this macro is not yet used under AMD64. To be able to use vmemmap in `__va`, we put `__boot_va` into use and patch all early boot locations to refer to it instead of `__va`.

5.6 CHANGES TO PAGE TABLE MANAGEMENT

We incorporate our randomization facility into page table management code at the point where page table structures are allocated and freed in the kernel, i.e., `arch/x86/mm/pgtable.c`. The details of how the page tables of a process are allocated at the time its creation are depicted in Figure 5.2. User space initiates process creation by issuing the Linux kernel implementation of the `execve` system call in ❶. This triggers the system call handling code in ❷, which we already described in detail in Section 2.2. The system call handler then redirects execution to `do_execveat_common` in ❸, which implements the `execve` system call. This then calls `bprm_mm_init`, which allocates the process's `mm_struct` through a call to `mm_alloc` in ❹. After allocation, this function call `pgd_alloc`, which assigns a free page to the PGD of the new process, and sets its reference in the `mm_struct` in ❺.

Consequently, the point where the randomization of process page tables starts is `pgd_alloc` and the functions it calls. There are some differences between how the PGD level of the page table hierarchy is allocated, and how allocation works for the lower levels. For instance, the allocation functions `__pud_alloc_one` and `__pmd_alloc_one` are located in `arch/x86/include/asm/pgalloc.h`, whereas `pte_alloc_one` also resides in `arch/x86/pgtable.c`. We modify all of these allocation sites of page table structures to use our wrapper implementation for randomized pages. We introduce similar changes to kernel routines for freeing, setting, and clearing of individual page table structures.

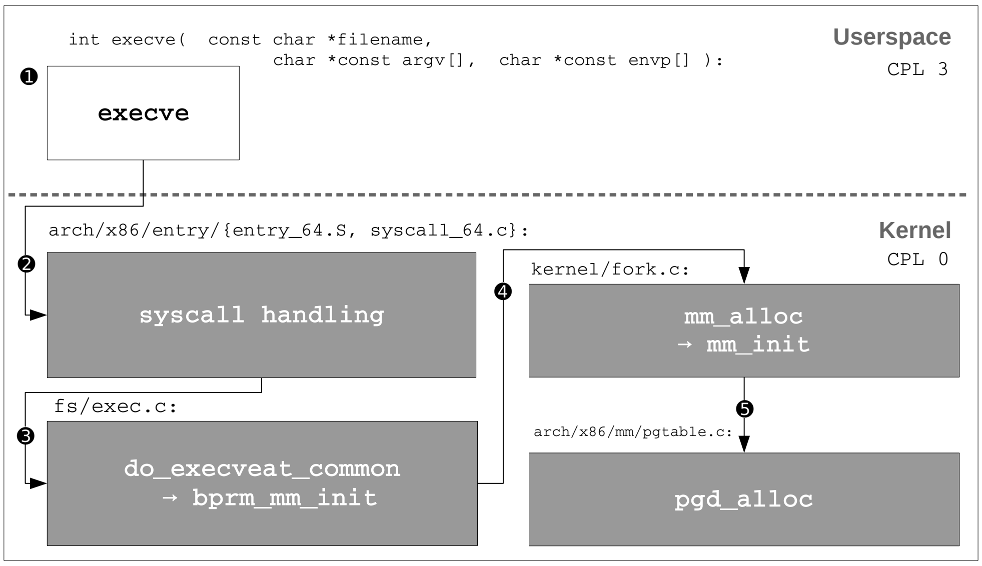

Figure 5.2: We call our randomization wrapper to page table allocation during process creation.

5.7 SUMMARY

In this Chapter, we investigated the technical details of our proof-of-concept implementation of Page Table Randomization. We explained, how it manages randomized pages in the form of a wrapper around standard kernel allocation. Additionally, we listed necessary configuration changes for our implementation. We also described, that the removal from physmap and subsequent randomization of individual pages poses some technical challenges, for instance with regard to huge pages. Moreover, we presented how randomized pages are marked and distinguished by kernel from regular pages. This enables translation from physical to randomized virtual addresses without using physmap. Finally, we laid out some of the more important modifications to page table management, that are required, such as allocation of page table structures. In the next Chapter we will evaluate our prototype in terms of performance, security, and also revisit the attack from Chapter 2.

6 EVALUATION

In this Chapter we evaluate the performance, security, and feasibility of the implemented approach on the basis of our research prototype.

6.1 SETUP

In order to test our implementation, we assembled three different configuration setups for use with a 4.2 kernel tree. The first is for development and early testing using the QEMU emulation software. It has only a minimal configuration. In particular, there is no support for modules, networking, an initram file system, or any unnecessary hardware drivers. This results in 573 configuration options being selected to be explicitly built into the kernel[1], without any additional modules.

The second setup is for more elaborate testing and also benchmarking on real hardware. Consequently, its configuration is bigger, including support for kernel modules, networking, initramfs, and some hardware drivers, resulting in 658 options selected overall. Although it is of course possible to boot into a plain single user Linux environment[2], it is usually a lot more convenient to use one of the popular Linux distributions. This second setup is explicitly designed and tested to work with Debian Jessie, version 8.2. So, in order to test and evaluate this setup a default Debian installation is used. We use `fakeroot` and `make-kpkg` to build a `.deb` package for installation of our custom kernel and initramfs via the `dpkg` package manager within the existing Debian installation.

The third setup is very similar to the second, but additionally includes support for a graphical desktop, adding further options to the configuration, resulting in 802 selected options. For comparison, a standard Ubuntu configuration contains as many as 5697 selected options. In addition to the default headless Debian installation in the second setup, we use the SLiM display manager and LXDE as desktop environment. An installation for the second setup can be changed into a graphical setup by installing the packages required for these two and installing the custom kernel with graphics support.

1 We report the output of `sed /^#/d .config | sed /^\s*$/d | wc -l`.
2 For instance, via booting with `init=/bin/sh`.

6.2 PERFORMANCE

In order to accurately assess the overhead our prototypical implementation of Page Table Randomization is causing, compared to a stock 4.2 kernel, we conduct a series of benchmarks using the two widespread benchmarking tools LMBench 3.0 and SPEC2006. We can additionally report, that the graphical setup is working correctly, however we only collect benchmarking numbers for the second setup. Both benchmarking kernels are built with the second configuration, but only one has Page Table Randomization enabled. The machine these benchmarks are run on has an Intel Core i7-4790 CPU running at 3.60GHz and 8GB of main memory installed.

6.2.1 *LMBench*

For running the LMBench benchmarking suite, we use the headless Debian Jessie installation and 4.2 custom kernel configuration as described in the previous section. We configure LMBench to run in single CPU mode and let the scheduler assign job placements. The most important results are summarized in Figure 6.1 and Figure 6.2. The numbers represent the arithmetic mean over three benchmark runs. In particular, we see that while the latency overhead during process creation is at around 20%, there is very low to practically no overhead involved during process execution time.

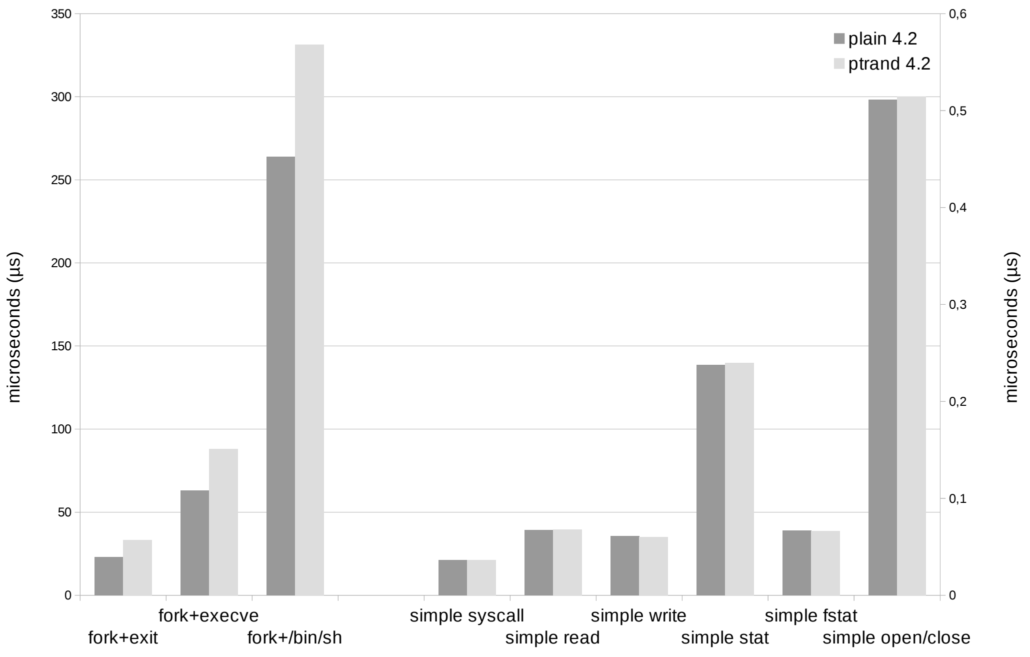

Figure 6.1: Process creation and runtime latency comparison

This makes sense, because our prototypical implementation of Page Table Randomization has to randomize a number of page table pages only in the beginning of process creation in the kernel. Note, that there is also overhead involved in destroying a process, because the randomized pages have to be derandomized again. During process execution, randomization only causes a difference in calculating virtual addresses of randomized pages, which does not have any huge impact in terms of wall clock time.

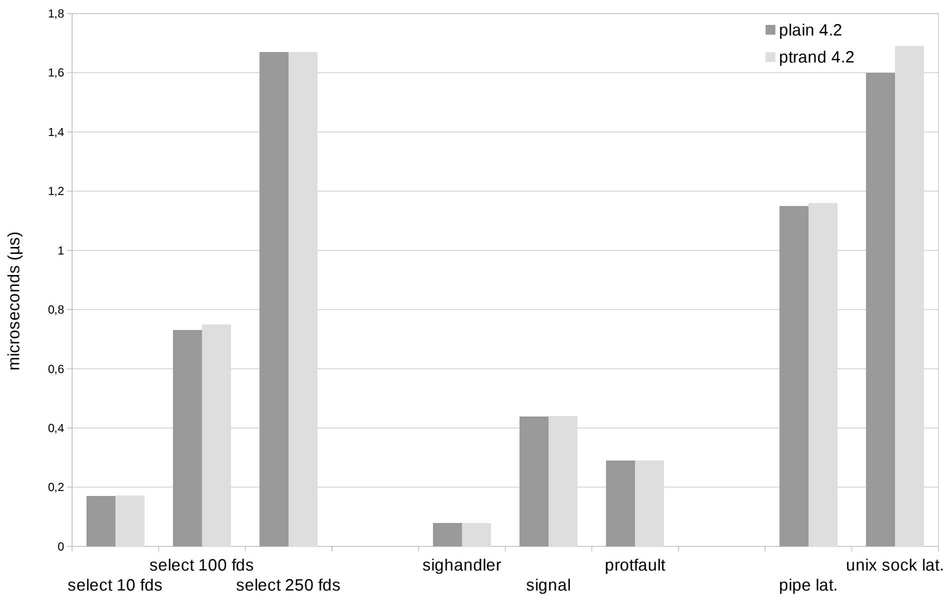

Figure 6.2: Runtime latency comparison (cont.)

Because we made modifications to the page table processing routines in the kernel, one might expect some overhead to occur upon task switching, with our modifications. In Figure 6.3, we see that the overhead for context switching is negligible, however. There is slight overhead for file system access, which we assume is caused by the additional memory pages dynamically mapped into the process space, because the additional page table entries required for this have to be randomized during process runtime.

Note, that LMBench also has a number of tests related to memory bandwidth like memcopy and piping, tests related to computational tasks like integer and floating point operations, L1 and L2 cache latency, and general main memory latency on sequential and random accesses. However, all of these numbers are not reported here, because they show no difference in comparison to an unpatched 4.2 kernel.

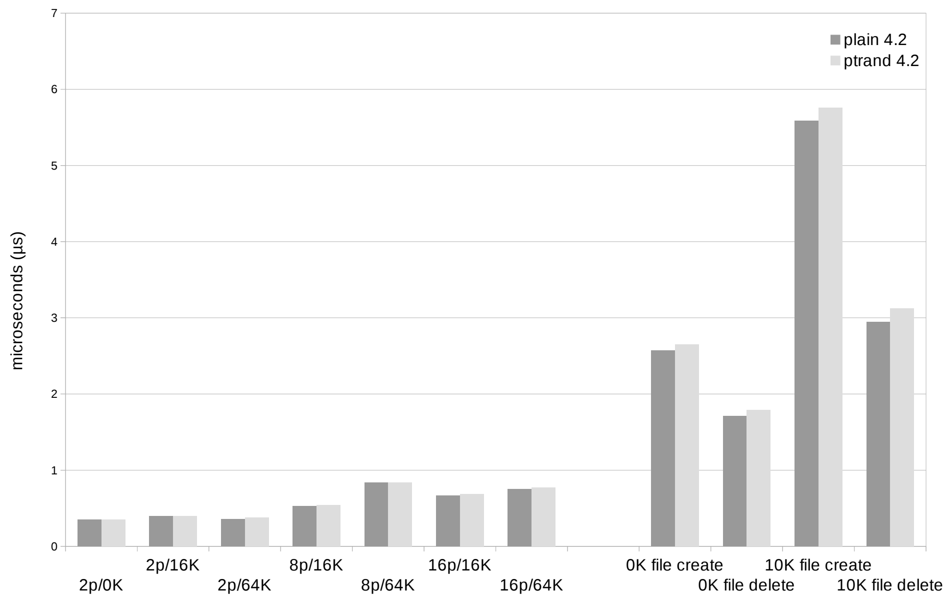

Figure 6.3: Context switching and file system access

6.2.2 *SPEC2006*

The results of the SPEC2006 benchmark tool are given in Figure 6.4. SPEC2006 reports the median of three consecutive runs over all benchmarks. The visible overhead ranges from -0.5% to 3%. Some of the tests even seem to benefit from our modifications to page table management. The average overhead of all 19 tests is 0.3806%.

Overall, this shows that the average overhead caused by our prototypical implementation of Page Table Randomization to a process depends on its execution life time. For very short-lived processes this may be around 10%, however for processes living as long as a couple of seconds the average constantly drops to quickly reach less than 1%. That is why we find the usefulness of an average overhead number in percentage to be rather limited for our prototypical implementation. This behavior is due to our wrapper implementation of randomization of memory pages.

6.3 SECURITY

We now evaluate the security properties of our approach, based on the proof-of-concept implementation presented in the last Chapter. In particular, we look into resilience to information disclosure, available entropy, and briefly revisit the attack presented in Section 2.5. It is important to note, that more advanced attack approaches like, e.g.,

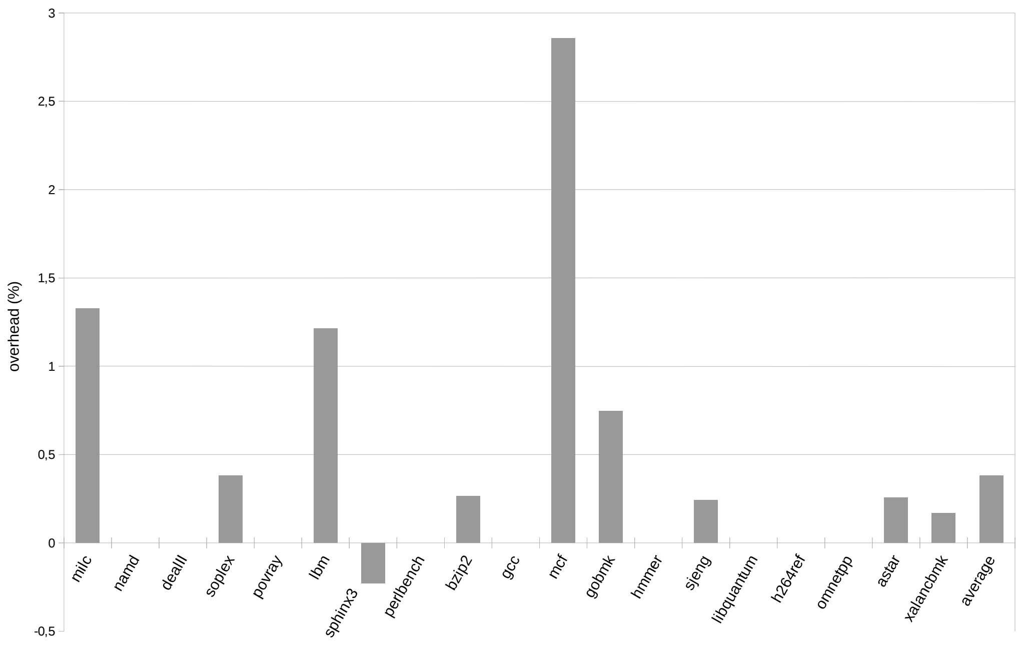

Figure 6.4: SPEC2006 benchmark results

timing-side-channels [39], represent an interesting point of research, but are out of scope for our current work. [3]

6.3.1 *Memory-Disclosure Attacks*

In Section 4.2, we explained that randomization approaches are vulnerable to memory disclosure [25, 71, 73, 74]. Thus, we outlined our basic strategies for addressing the threat of memory disclosure in Section 4.4 on a conceptual level. For our prototypical implementation, we choose to keep the base address in one of the AMD64 debug registers, although there are several alternatives for implementing this, such as model specific registers. The advantage of using debug registers is, that these registers are available and stable across a wide range of AMD64 processors. Additionally the debug registers should not be used on production systems.

We actively prevent the leakage of any page table addresses through returning the physical address of newly randomized pages within the page table allocation functions in the kernel. This also ensures, that there are no leaked references to page tables in the kernel source

3 Although Hund, Willems, and Holz present many technical details, it remains unclear, whether their attack can be applied to Page Table Randomization. For instance, they make use of the timing difference between 2M pages and 4K pages, whereas our prototypical implementation only utilizes the latter for page tables.

we forgot to check. Otherwise the kernel would crash upon using the returned physical address. Additionally, we protect local variables containing randomized addresses from leaking any information through using explicit register variables [85]. This is necessary, because an adversary with read access in kernel memory could monitor kernel stack contents [20], to expose information about the base address through local variables placed on the stack. Because we do not maintain any global variables, this can be achieved through use of inline assembly locally, within a routine, and we do not need to instrument, or reconfigure the compiler. We use these register variables for local accesses and calculations, and clear them again prior to calling another routine or returning.

An adversary is not able to disclose the base address through a data-only attack, because we keep it in a privileged register and it is never written to memory. Although accessing registers is not possible through data-only attacks, an adversary might launch a code-reuse attack to spill the register. As explained in Chapter 3, our approach is not meant to defeat code reuse, because kernel monitor implementations like CFI already aim to provide protection against this type of attack. However, our approach enables efficient implementation of such kernel monitors.

Although our defense approach is not designed to mitigate code-reuse attacks, we take great care to avoid placing a gadget in the kernel code, which would allow an adversary to trivially leak the contents of the register holding our base address. In particular, we are able to verify, that there is no sequence of at most 13 instructions, that would represent such a gadget. [4] While such a gadget with more than 13 instructions might exist, with increasing complexity of the gadgets it becomes less likely that an adversary is actually able to use it. In particular, to leak the base address the attacker has to prevent destroying the register contents or crashing the system. This represents a huge challenge for kernel-level code reuse that needs to incorporate instruction sequences of this size.

Besides using data-only, or code-reuse attacks to disclose the base address, an adversary might corrupt a device driver for peripherals capable of direct memory access. An adversary could use such a device to access the physical addresses written to main memory during our randomization of page tables. The problem is that translation of this physical address to a virtual address is no longer required for an adversary equipped with access to physical memory. However, this kind of attack is not possible within our threat model, because physical memory accesses are assumed to be restricted through hardware support like IOMMU [1]. More specifically, the kernel can restrict

4 We manually inspect the output of `objdump -dr vmlinux | grep -color=always -A 13 'mov.*\%db3' | grep -B 13 'ret'`. For gadget sizes of more than 13 instructions, these get more complex, with intermediate function calls or jumps.

DMA accesses to protect page frames containing page tables using this architectural defense.

6.3.2 *Brute-Force Attacks*

Although an adversary is not able to disclose information about randomized addresses, we additionally have to prevent feasible brute-force search for our randomized base address. These attacks represent another major threat for randomization approaches [30, 71]. Page Table Randomization does not set an upper limit on the amount of entropy, apart from the size of the relocated memory area used for randomization. For our proof-of-concept implementation the size of this area is 40 bits.

In practice, further restrictions apply to this upper limit. For instance, we make use of the `get_random_bytes` function, which represents the interface to the `/dev/random` character device for kernel use. All restrictions of this interface also apply to our random offset. However, this is also the case for key generation, seeding of TCP sequence numbers, file system encryption, and many more call sites in the kernel. Our calculations for the available amount of entropy are based on the assumption, that the random numbers returned by this interface during boot are distributed uniformly.

Recall, that the kernel is assumed to be built with configuration options, which cause the machine to shutdown upon a page fault in kernel memory. [5] Hence, an adversary is limited to a one-shot guess and cannot scan our virtual memory area for page table entries. We limit the random offset to `0x1000000`, or 2^{24}, to avoid wrapping around the top of our randomized area. This is not a fundamental limitation, but rather an arbitrary choice, which facilitates our prototypical implementation. Taking into account, that randomization always happens per page, this leaves 4096 possible addresses, or 12 bits, that an adversary has to guess from. The chances of brute force discovery of the random offset therefore are at 0.02442%.

6.3.3 *Real-world Benefits*

In Section 2.5 we presented a data-only attack on kernel memory, which allows an adversary to subvert process page tables. Recall, that there were two main steps involved. The page table entry point of the process had to be located, and then a page walk had to be performed. With Page Table Randomization enabled, both steps are not possible anymore. The `pgd` field of the `mm_struct`, where an adversary would usually have to look for the entry point to the page table hierarchy of the target process now holds a physical address. As before, this

5 This means we have `PANIC_ON_OOPS` enabled, or booted with `oops=panic`.

is also the case for internal references between different page table levels. However, because of the modifications to physmap and the new, randomized area for page table pages in virtual memory, these physical addresses are no use to an adversary. The attacker would have to know the secret offset into the randomized area, but this secret is kept in a privileged register and not present in memory. As a result, data-only attacks that aim to subvert page table mappings in the kernel are no longer possible.

6.4 SUMMARY

Evaluating our proof-of-concept implementation, we find it to satisfy the goals and requirements stated in Chapter 3. In particular, it offers enhanced security guarantees, that can be used to implement efficient kernel monitors. Additionally, it demonstrates feasibility and stability through a whole range of different tests and setups, including a fully-fledged graphical desktop configuration and running real-world applications. Although the measured runtime overhead is practically negilible, there is still some room for improvement. An interesting observation is that larger performance penalties are visible to process creation and destruction. However, this overhead to starting and exiting processes seems to be an artifact of our wrapper implementation, and not inherent to the conceptual design of Page Table Randomization. In the next Chapter we will discuss these findings.

7 DISCUSSION

We will now look into applications, some lessons learned, and limitations of our work. We also discuss interesting ideas for ports of our approach and future research.

7.1 APPLICATION TO KERNEL MONITORS

In the beginning, we introduced two key motivational goals for designing and implementing a protection mechanism for the page tables within the memory domain of the kernel. First, an adversary with read and write access in kernel space is able to completely bypass the security guarantees through data-only attacks. Second, a memory protection mechanism in the kernel withstanding malicious read and write operations enables the implementation of efficient kernel monitors.

In Section 6.3.3 we described, how our defense approach offers real-world benefits to kernel memory protection. Additionally, it can be used as a building block for kernel monitors, by utilizing the strengthened memory protection guarantees in the kernel. In particular, Page Table Randomization allows to create an immutable memory domain for policies, by setting the page table mappings of the kernel monitor as read only after setup.[1] To bypass the kernel monitor, an adversary would have to modify the page tables. But because Page Table Randomization randomizes the location of the page tables in virtual memory, these cannot be located by the attacker. Therefore, kernel monitors can be implemented natively, using the memory protection guarantees that the kernel provides.

This represents a substantial gain, compared to other kernel monitor implementations. Currently, these have to resort to additional hardware capabilities such as virtualization, trusted execution environments, or resource-intensive runtime checks, to realize a secure memory domain for storing policies. Page Table Randomization does not have such requirements. Additionally, it provides significant gain in terms of performance, without increasing TCB complexity, because the implementation introduces only small code changes to kernel virtual memory management. However, verifying the runtime speedup for kernel monitor implementations is beyond the scope of this work, and we have to leave this for future research.

1 In the case of CFI the policies may, e.g., be embedded in kernel code.

7.2 LIMITATIONS AND LESSONS LEARNED

In Section 5.3 we described the required changes in the kernel configuration setup for our implementation to work properly, and offer reasonable security guarantees. As a result, our implemented patch is not compatible with any configuration setups that provide support for paravirtualization, sleep mode (suspend-to-RAM), hibernation (suspend-to-disk), or various debug options, such as kmemcheck. Nevertheless, we feel that these are not major drawbacks, as serious disk encryption setups have essentially the same implications. The largest drawback is probably the incompatibility with any kind of virtualized setup, although we see no particular reason, why a port of our implementation to Xen should not be possible. However, this is outside the scope of this work and we find this to be an interesting question for further research.

Instead of having `STRICT_DEVMEM` enabled in the configuration, disabling `DEVMEM` altogether represents a cleaner solution. Yet, we already explained this may not be possible in certain cases, such as using X11 with a graphical desktop. In our headless setup configuration file we do not expose this character device. We did not look into the implications this has for the graphical setup in terms of security in greater detail, thus this question also remains for future research.

One limitation of our prototypical implementation is that randomizing pages is not possible prior to the initialization of the virtual memory map subsystem (`vmemmap`). Nonetheless, this has no impact on the security guarantees, because we do not consider the attacker to have access to the system at this point in time. We also mentioned, that the random numbers used for initializing Page Table Randomization are assumed to be distributed uniformly. In reality, there might be several reasons why this is not the case. However, as our proposed defense approach requires a benign boot environment anyway, we do not investigate the security risks arising from adversaries manipulating random number generation during boot time.

Although we find our implementation to offer appropriate performance, this could still be improved further. In particular, designing the randomization of individual pages in the form of a wrapper around the standard kernel allocator is non-optimal. Ideally, the allocator would have the ability to return randomized pages directly, for instance, through managing a pool of randomized pages, saving the overhead of randomization and derandomization upon allocation and freeing. This should eliminate the current penalties during process creation completely, and bring our implementation even closer to the runtime properties of the stock kernel.

7.3 FUTURE WORK

Looking ahead, there are some interesting questions, which we had to leave to further research, because of limited time and resources. In particular, implementing a kernel monitor based on Page Table Randomization would allow for even better performance and security evaluation, and demonstrate further applicability. Moreover, porting our presented defense approach to a hypervisor, e.g., Xen, should as already mentioned be possible in principal, and thus seems like an interesting project of its own. It would of course be highly encouraging, to incorporate the results of our research in a real world implementation of Page Table Randomization for Linux. However, considering the more conservative approach some of the kernel developers exhibit — which is beneficial to the overall development — this seems unlikely. Finally, throughout this work, we had to strongly focus on the AMD64 architecture, which is historically one of the main architectures deployed. Yet, the conceptual idea we presented in this work is not fundamentally tied to that platform, and developing an implementation for, e.g., the ARM architecture should be possible, and would allow us to offer this solution to the world of embedded and mobile devices as well.

8 RELATED WORK

In this chapter, we give an overview of system security research, as well as current concepts and topics, focusing on enhanced memory protection guarantees for the kernel.

8.1 GENERAL SYSTEM SECURITY AND TRUSTED COMPUTING

Just like today's internet protocols, security of computer systems was first regarded and researched in the context of military organizations using this technology within mission-critical components. This is why one of the early standards in this field was publicized by the United States Department of Defense [83]. Over the years, many other standards and best practices have been published, most notably the Common Criteria for Information Technology Security Evaluation [19], which represents an international standard for classification of IT systems into different assurance levels, based on formal methods.

There have also been industry-driven efforts of standardizing platform security, such as the Trusted Platform Module (TPM) [84]. Additionally, there are vendor specific technologies promising to enable secure software and computation through Trusted Execution Environments [40], or related techniques, such as Intel's Trusted Execution Technology (TXT) [42], Intel's Software Guard Extensions (SGX) [41], AMD's Platform Security Processor [6], or ARM's TrustZone [82]. Most of these proprietary technologies have been heavily criticized for being non-auditable, non-verifiable mechanisms, that are used for digital rights management, platform lockdown, and "lawful interception", rather than providing enhanced user and platform security [11, 12, 66, 69].

Furthermore, these technologies are being incorporated into modern platform architecture and chip designs, such as Intel's Management Engine (ME) [67], thereby introducing novel capabilities for firmware. Intel ME is a management and security processor placed within the CPU package, that runs a proprietary real-time operating system, features its own internal SRAM, internal ROM, cryptographic engine, access to the platform's SPI flash, direct memory access to host memory, the ability to shutdown the platform, and drive execution on the host CPU. It is used to implement security features of current Intel processors, such as TXT, AMT, SGX, and recent versions of Intel's TPM. [1] It is therefore incorporated into all major core pro-

1 We would like to refer the interested reader to [22] for a slightly more elaborate discussion of Intel's current security technology and especially of SGX.

cessors in this form since the Haswell generation [67]. It continues to run, when the host CPU is in sleep mode.

These architectural modifications invalidate the traditional ways of achieving isolation and integrity, such as the ring model, or virtual memory, which we introduced in Chapter 2, because of their demanding capabilities. Unlike these hardware-based mechanisms, firmware-based solutions are implemented through vulnerable software. In the case of Intel ME, leaking the engine's key essentially creates a plausibly deniable backdoor with unrestricted access to the platform [67, 69].[2]

8.2 OS KERNELS FROM A SECURITY PERSPECTIVE

Trust, in the sense of computing technology, is a liability. A trusted component has the ability to compromise the computing system, thus the goal is to minimize the amount and size of components that must be trusted. The operating system kernel typically represents part of the Trusted Computing Base (TCB) of a computer system in terms of [83]. To establish some foundation for trust in the kernel, two hypotheses are commonly stated:

A) Kernel integrity can be described by a set of invariants, i.e., properties that always hold at runtime.

B) These invariants can be enforced.

Regardless of how the set described in A) is obtained, different approaches can be taken to achieve B). For instance, an implementation can be statically checked for *correctness* against a set of conditions, called the *specification*. If the implementation can be shown to be correct, the conditions of the specification indeed always hold, which means that they are enforced. Another approach is to enforce — at least partially — intended behavior directly using a *reference monitor*. Reference Monitors facilitate runtime checks of an implementation against a set of rules called *policies*.

It is important to note, that both approaches make no further statement about the completeness or quality of the set of invariants described in A). Regardless of the approach taken, the hardware and build toolchain still have to be trusted. While malicious hardware is a huge issue that currently remains largely unsolved [78], building a trustworthy toolchain is indeed possible, alas this is not at all a trivial task [88]. We will now look at some examples of the two approaches noted above.

2 This key is subject to US legislation.

8.2.1 *Correctness*

Proof of correctness may include fully automatic procedures, such as leveraging theorem provers [28], for statically showing correctness of an implementation according to a formal specification. The correctness approach was taken, e.g., by Klein et al. for verifying seL4 [49], and Baumann and Bormer for verifying PikeOS [9]. However this is often considered to be infeasible for large code bases.[3] Also, the limited scope and hardware support of these micro kernels currently makes them unfit for the broader market of general purpose platforms.

Nonetheless, the idea to statically prove certain properties of the code can be generalized to a larger class of approaches and tools. These tools utilize static analyses, to show strict physical type safety [15], a selected set of program states [7], temporal relations and restrictions [56], or call-graph, points-to, alias, and taint information [90]. Using these properties, the next step is to prove further statements, such as the absence of memory-corruption vulnerabilities in a program [23]. There are specialized tools for static analyses of Linux kernel trees, like checking for type inconsistencies [80], so-called semantic patches [52], or variability issues [47]. Currently, free analysis frameworks which also support less specialized and more complex static analyses of large code bases, such as Soot [91] or Clang [64], focus on application-level code.

8.2.2 *Reference Monitors*

In the case of using a reference monitor, its implementation has to be trusted in turn. Reference monitors designed towards safeguarding operating system kernels are called *kernel monitors*. This approach is only sensible, if the reference monitor implementation is smaller than the kernel being monitored, otherwise the size of the TCB is increased. Examples of kernel monitors include Data-Flow Integrity (DFI) [14], Memory Safety [10, 21], Control-Flow Integrity (CFI) [2], and Code-Pointer Integrity (CPI) [50]. More general approaches, such as Qubes OS [65], might be categorized as an implementation of a kernel monitor as well. Note, that Linux Security Modules like SELinux [61] and AppArmor [55] are monitor implementations enforcing policies — mostly for Mandatory Access Control — in user space, and are thus by design not able to monitor the kernel.

In contrast to correctness approaches, which try to assess the vulnerability, reference monitors aim to prevent its exploitation at runtime. There are several limitations to this approach. The development of defense techniques like CFI and related monitor approaches raised

3 Where *large* means more than 10.000 lines of code.

the question, if code-based integrity checking offers sufficient protection in general [13, 14, 18], and further research will be necessary to assess the overall effectiveness of code-based reference monitors.

8.3 KERNEL MONITOR IMPLEMENTATIONS

There are several implications for the implementation of kernel monitors, following the assumption of our adversarial model. Recall, that the goal of a kernel reference monitor is to enforce policies, which have to be stored somewhere in memory. Therefore, the challenge in implementing a kernel reference monitor is to enforce these policies against an adversarial model, which includes malicious memory accesses within the memory domain of the kernel. There are additional challenges, when also considering remote attackers, or external devices, because as already mentioned in that case the network stack, and devices capable of direct memory access also represent system boundary.

SECVISOR One possibility is to protect the policies using a hypervisor, because a hypervisor maintains its own memory domain, and virtualized guests should not have access to it. An early example of a kernel monitor implementation using a thin hypervisor is SecVisor [70], which implements a weaker form of code-based protection, that aims to defeat code injection, but not code reuse. Moreover, it shows significant runtime overhead, when compared to other hypervisors. The threat model for SecVisor includes malicious DMA accesses.

KCOFI An example of a full CFI implementation for an operating system kernel is KCoFI [26], which stores the policies for safeguarding its virtualized guests securely inside a memory region that is only accessible to the hypervisor. Alas, this solution also comes with significant overhead, added to virtualization costs of a novel hypervisor implementation, called the Secure Virtual Architecture (SVA). SVA builds on the LLVM intermediate representation for providing enhanced memory safety guarantees. Currently, Linux is not available on SVA, because it cannot be fully compiled using LLVM. The adversary model assumed by KCoFI is very similar to ours, with a strong focus on code-reuse attacks.

SKEE Another possibility for creating a secure memory domain for policies, is to pass all memory management through a set of runtime checks. One example of this approach is SKEE [5], which instruments the kernel's memory management operations for the ARM architecture in order to add security checks, which are performed inside a secure environment, to prevent malicious modifications. The secure environment is running time-shared with the protected kernel and also relies on a fixed set of rules for its runtime checks. The overhead of running this environment time-shared with the main CPU for the purpose of guarding memory management varies between 3% and 15%. The authors designed SKEE explicitly for the ARM platform, and it remains unclear, whether their solution is applicable to other platforms as well.

hypersafe A conceptually similar approach was presented in [86] for the AMD64 platform, reporting an average overhead of around 5%. Although HyperSafe represents a CFI monitor for a hypervisor, we simply refer to the hypervisor implementation as a kernel implementation in this case. More specifically, it implements a table-based variant of CFI for the BitVisor hypervisor. Because Wang and Jiang change the page table mappings of the hypervisor to be write protected, they also have to modify page table management code to include additional runtime checks, a technique they call *non-bypassable memory lockdown*. As the authors of KCoFI already noted, HyperSafe does not protect the return address against malicious modifications in interrupted program state, and it remains unclear how the authors protect the system against malicious concurrent memory accesses during a page table update.

8.4 probabilistic defense approaches

Historically, probabilistic approaches have been developed as defense techniques that try to lessen the reach of an effective vulnerability among identical systems by randomizing parts of the memory,[4] while aiming to induce only minimal overhead. Although initially designed to counter code injection, this was proposed to counter code-reuse attacks later as well. Randomization is used in user space to randomize code, data, or overall process memory layout. This aims to prevent widespread exploitation of a common vulnerability on systems deploying, for instance, the same version of some popular library like libc. These techniques leverage the virtual memory capabilities of the system to provide a probabilistic defense mechanism, that relies on available entropy and resilience to information disclosure, to provide enhanced memory protection guarantees. The various approaches are categorized into user- and kernel-level randomization [48].

8.4.1 *Randomization in User space*

The idea of address space layout randomization (ASLR) dates back to 1997 [32] and was implemented in 2000 [77, 89] as a means to randomize process memory layout on Linux-based systems. In particular,

4 Thus, these approaches are sometimes also referred to as *software diversification*.

PaX ASLR randomizes stack, heap, and shared libraries prior to program loading. Different configurations of PaX ASLR are possible to randomize different parts of a binary on a platform per compilation, or per execution requiring different amounts of entropy, and introducing varying overhead. There are several issues, that were discovered even if ASLR is applied correctly to the system, the most pressing being information-disclosure attacks [25]. An info leak discloses information about the memory layout or individual addresses of a process giving an adversary the opportunity to adjust their exploit.

In particular, leaking a single code pointer is sufficient to completely derandomize an application secured by ASLR [73]. There are multiple examples of memory-disclosure vulnerabilities exploited in the real world, e.g. in Adobe's Flash [35], Adobe's PDF Reader [51] or Microsoft's Internet Explorer [29]. Because of this, more fine-grained randomization techniques were subsequently proposed. Building the program as an position-independent executable (PIE) allows for randomization of the base address of the text segment, where code is placed. Address space layout permutation [48] was introduced in 2006 as an extension to also randomize static code and data by randomly permuting procedures, and data objects within loaded binaries.

Other fine-grained randomization defense mechanisms proposed more recently, like Instruction Location Randomization [37], try to randomize program binaries on the instruction level thereby introducing runtime penalties that render those schemes infeasible in practice. A more efficient scheme called Self-Transforming Instruction Relocation [87] was introduced in 2012 for the x86 platform. Readactor [25] presents a fine-grained randomization scheme, including resilience to direct and indirect memory disclosure, to also defeat advanced code-reuse attacks, such as JIT-ROP [73].

8.4.2 *Randomization in the Kernel*

The kernel faces exploit techniques that are conceptually similar to that used in user space, like kernel-ROP [38], return-to-user [46], and kernel-level code injection. Thus, the conceptual idea of randomizing parts of the kernel address space is essentially also similar to randomization in user space. Yet, there are major differences in terms of details. The general concept of randomizing different parts of the kernel is commonly referred to as *kASLR* [34]. To our knowledge the built-in configuration option for randomizing kernel code is the only kASLR implementation available for Linux. Other major operating system kernels also implement some form of kernel space randomization [62, 68].

The built-in randomization feature of the Linux kernel was proposed during development of version 2.6 [27], but only implemented late into version 3.9 [44] after being implemented for Google's Chrome-OS. As of version 4.2 it is not selected by default. It aims at diversifying kernel code addresses by randomizing the physical memory destination address for unpacking of the kernel image at boot time. This means that — when enabled — the start address of the kernel in physical memory is different on every boot. Because this happens during early boot time, there are certain alignment and range restrictions to this kind of randomization, i.e., the amount of entropy available for kASLR is limited.

This concept of kernel code randomization has been criticized by some as being trivially vulnerable to information disclosure attacks, because of the way it is currently implemented, and the multitude of info leaks present in the kernel [74].

9 CONCLUSION

In the beginning of this work we explained, that an operating system kernel faces unique challenges and adversaries, compared to application level software. It has to maintain integrity, access restrictions, and isolation for user processes. Additionally, it must enforce strong separation between kernel and userland execution and memory. Runtime attacks, e.g., through exploitation of memory-corruption vulnerabilities, threaten to corrupt the integrity of the kernel.

In Chapter 2, we learned that in order to maintain integrity, the kernel needs hardware support in the form of privilege levels and virtual memory. We also saw, why corrupting page tables is a profitable goal for an adversary and how this violates memory-protection guarantees of the system.

In Chapter 3, we analyzed the technical root causes for this in detail and specified open challenges and requirements. We then formulated three principal ideas for solving these and selected the randomization approach for further research because of its unobtrusive runtime properties.

In Chapter 4, we explained our detailed design of Page Table Randomization and also presented solutions to the challenges related to it. Chapter 5 gave a deep, technical description of our research prototype implementing the key concepts of this design. In Chapter 6, we tested, benchmarked, and evaluated our ideas based on this proof-of-concept implementation. We discussed these results and our work in Chapter 7, briefly presenting related concepts and research in Chapter 8.

Although we identified some limitations to our solution, we find these to be due to our prototypical implementation and not of conceptual nature. In addition to providing enhanced kernel security through a novel defense technique, we find this work to be an essential building block for the implementation of kernel monitors that do not need to rely on external memory management enforcement like hypervisors, trusted execution environments, or resource heavy runtime checks.

APPENDIX A

Listing A.1: Configuration requirements of Page Table Randomization.

```
CONFIG_RELOCATABLE=y
CONFIG_RANDOMIZE_BASE=y
CONFIG_X86_NEED_RELOCS=y
CONFIG_RANDOMIZE_BASE_MAX_OFFSET=0x40000000
CONFIG_PANIC_ON_OOPS=y
# CONFIG_DEVKMEM is not set
# CONFIG_DEVMEM is not set
# CONFIG_HUGETLBFS is not set
# CONFIG_WANT_PAGE_VIRTUAL is not set
```

Listing A.2: KConfig requirements of Page Table Randomization.

```
# HAVE_ARCH_TRANSPARENT_HUGEPAGE is not selected

config PHYSICAL_ALIGN
       default "0x200000"

config X86_DIRECT_GBPAGES
       def_bool n

config ARCH_WANT_HUGE_PMD_SHARE
       def_bool n

config ARCH_WANT_GENERAL_HUGETLB
       def_bool n

config ARCH_ENABLE_HUGEPAGE_MIGRATION
       def_bool n
```

LIST OF FIGURES

LISTINGS

ACRONYMS

CFI	Control-Flow Integrity
CPL	Current Privilege Level
MMU	Memory Management Unit
TLB	Translation Lookaside Buffer
VMA	Virtual Memory Area
PTE	Page Table Entry
PMD	Page Middle Directory
PUD	Page Upper Directory
PGD	Page Global Directory
ROP	Return Oriented Programming
TSS	Task State Segment

BIBLIOGRAPHY

[1] I AMD. “I/O Virtualization Technology (IOMMU) Specification.” In: *AMD Pub* 34434 (2007).

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. “Control-flow integrity.” In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2005.

[3] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, George C. Necula, and Michael Vrable. “XFI: Software Guards for System Address Spaces.” In: *7th USENIX Symposium on Operating Systems Design and Implementation*. OSDI. 2006.

[4] Andi Kleen. *AMD64 Linux Virtual Memory Map*. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt. 2004.

[5] Ahmed Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. “SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM.” In: *"23rd" ndss*. A preprinted version of this paper was kindly provided to us by the authors. NDSS. 2016.

[6] *BIOS and Kernel Developer’s Guide (BKDG)*. http://support.amd.com/TechDocs/52740_16h_Models_30h-3Fh_BKDG.pdf. AMD, 2015.

[7] Thomas Ball and Sriram K Rajamani. “The SLAM toolkit.” In: *Computer aided verification*. Springer. 2001, pp. 260–264.

[8] Arash Baratloo, Navjot Singh, Timothy K Tsai, et al. “Transparent Run-Time Defense Against Stack-Smashing Attacks.” In: *USENIX Annual Technical Conference, General Track*. 2000, pp. 251–262.

[9] Christoph Baumann and Thorsten Bormer. “Verifying the PikeOS microkernel: first results in the verisoft XT avionics project.” In: *Doctoral Symposium on Systems Software Verification (DS SSV’09) Real Software, Real Problems, Real Solutions*, p. 20.

[10] Josh Berdine, Byron Cook, and Samin Ishtiaq. “SLAyer: Memory safety for systems-level code.” In: *Computer Aided Verification*. Springer. 2011, pp. 178–183.

[11] Caspar Bowden. *Reflections on Mistrusting Trust*. http://qconlondon.com/london-2014/dl/qcon-london-2014/slides/CasparBowden_ReflectionsOnMistrustingTrustHowPolicyTechnicalPeopleUseTheTWordInOppositeSenses.pdf. 2014.

[12] Bundeministerium des Innern (Federal Ministry of the Interior — Federal Republic of Germany). *Eckpunktepapier der Bundesregierung zu "Trusted Computing" und "Secure Boot" (White Paper of the German Government concerning "Trusted Computing" and "Secure Boot" — in german only).* http://www.bmi.bund.de/SharedDocs/Downloads/DE/Themen/OED_Verwaltung/Informationsgesellschaft/trusted_computing.pdf. 2012.

[13] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. "Control-flow bending: On the effectiveness of control-flow integrity." In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 161–176.

[14] Miguel Castro, Manuel Costa, and Tim Harris. "Securing Software by Enforcing Data-flow Integrity." In: *7th USENIX Symposium on Operating Systems Design and Implementation*. OSDI. 2006.

[15] Satish Chandra and Thomas Reps. "Physical type checking for C." In: *ACM SIGSOFT Software Engineering Notes*. Vol. 24. 5. ACM. 1999, pp. 66–75.

[16] Kaiqu Chen. *Linux Kernel - libfutex - Local Root for RHEL/CentOS 7.0.1406*. https://www.exploit-db.com/exploits/35370. 2014.

[17] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. "Non-Control-Data Attacks Are Realistic Threats." In: *14th USENIX Security Symposium*. USENIX Sec. 2005.

[18] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. "Non-Control-Data Attacks Are Realistic Threats." In: *Usenix Security*. Vol. 5. 2005.

[19] *Common Criteria for Information Technology Security Evaluation*. http://www.commoncriteriaportal.org/cc/. International Organization for Standardization, 1996.

[20] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. "Losing control: On the effectiveness of control-flow integrity under stack attacks." In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 952–963.

[21] Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr. "Efficient memory safety for TinyOS." In: *Proceedings of the 5th international conference on Embedded networked sensor systems*. ACM. 2007, pp. 205–218.

[22] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. https://eprint.iacr.org/2016/086.pdf. Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2015.

[23] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "The ASTRÉE analyzer." In: *Programming Languages and Systems*. Springer, 2005, pp. 21–30.

[24] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks." In: *8th USENIX Security Symposium*. USENIX Sec. 1998.

[25] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. "Readactor: Practical Code Randomization Resilient to Memory Disclosure." In: *36th IEEE Symposium on Security and Privacy*. S&P. 2015.

[26] John Criswell, Nathan Dautenhahn, and Vikram Adve. "KCoFI: Complete control-flow integrity for commodity operating system kernels." In: *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE. 2014, pp. 292–307.

[27] Dan Rosenberg. *Randomize kernel base address on boot*. https://lwn.net/Articles/444556/. 2011.

[28] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[29] Paul Ducklin. *Anatomy of an exploit – inside the CVE-2013-3893 Internet Explorer zero-day*. https://nakedsecurity.sophos.com/2013/10/11/anatomy-of-an-exploit-ie-zero-day-part-1. 2013.

[30] Isaac Evans, Samuel Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. "Missing the Point(er): On the Effectiveness of Code Pointer Integrity." In: *36th IEEE Symposium on Security and Privacy*. S&P. 2015.

[31] Jérémy Fetiveau. *Windows 8 Kernel Memory Protections Bypass*. https://labs.mwrinfosecurity.com/blog/2014/08/15/windows-8-kernel-memory-protections-bypass. MWR Labs, 2014.

[32] Stephanie Forrest, Anil Somayaji, and David H Ackley. "Building diverse computer systems." In: *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*. IEEE. 1997, pp. 67–72.

[33] Varghese George and Tom Piazza. *Technology Insight: Intel Next Generation Microarchitecture Codename Ivy Bridge*. http://www.intel.com/idf/library/pdf/sf_2011/SF11_SPCS005_101F.pdf. 2011.

[34] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. "Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization." In: *USENIX Security Symposium*. 2012, pp. 475–490.

[35] Suhas Gupta, Pranay Pratap, Huzur Saran, and S Arun-Kumar. "Dynamic code instrumentation to detect and recover from return address corruption." In: *Proceedings of the 2006 international workshop on Dynamic systems analysis*. ACM. 2006, pp. 65–72.

[36] Martin Hilbert and Priscila López. "The world's technological capacity to store, communicate, and compute information." In: *science* 332.6025 (2011), pp. 60–65.

[37] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Mathew Hall, and Jack W Davidson. "ILR: Where'd My Gadgets Go?" In: *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE. 2012, pp. 571–585.

[38] Ralf Hund, Thorsten Holz, and Felix C Freiling. "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms." In: *USENIX Security Symposium*. 2009, pp. 383–398.

[39] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical timing side channel attacks against kernel space ASLR." In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE. 2013, pp. 191–205.

[40] GlobalPlatform Inc. *Trusted Execution Environments*. http://www.globalplatform.org/documents/GlobalPlatform_TEE_White_Paper_Feb2011.pdf. 2011.

[41] *Intel Software Guard Extensions*. https://software.intel.com/sites/default/files/article/413939/hasp-2013-innovative-technology-for-attestation-and-sealing.pdf. Intel, 2013.

[42] *Intel Trusted Execution Technology*. http://www.intel.de/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf. Intel, 2012.

[43] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf. 2012.

[44] Kees Cook. *kernel ASLR*.
https://lwn.net/Articles/546035/. 2013.

[45] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. "ret2dir: Rethinking kernel isolation." In: *Proceedings of the 23rd USENIX Conference on Security Symposium, SEC*. Vol. 14. 2014, pp. 957–972.

[46] Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. "kGuard: Lightweight Kernel Protection against Return-to-User Attacks." In: *USENIX Security Symposium*. 2012, pp. 459–474.

[47] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. *TypeChef*.
http://ckaestne.github.io/TypeChef/. 2010.

[48] Chongkyung Kil, Jinsuk Jim, Christopher Bookholt, Jun Xu, and Peng Ning. "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software." In: *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE. 2006, pp. 339–348.

[49] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. "seL4: Formal verification of an OS kernel." In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 207–220.

[50] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. "Code-Pointer Integrity." In: *11th USENIX Symposium on Operating Systems Design and Implementation*. OSDI. 2014.

[51] McAfee Labs. *Analyzing the First ROP-Only, Sandbox-Escaping PDF Exploit*.
https://blogs.mcafee.com/mcafee-labs/analyzing-the-first-rop-only-sandbox-escaping-pdf-exploit. 2013.

[52] Julia Lawall, Gilles Muller, and Nicolas Palix. *Coccinelle*.
http://coccinelle.lip6.fr/. 2010.

[53] JungSeung Lee, HyoungMin Ham, InHwan Kim, and JooSeok Song. "POSTER: Page Table Manipulation Attack." In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 1644–1646.

[54] Adam Litke. *AMD64 Page Table Structure*.
http://linux-mm.org/PageTableStructure. 2007.

[55] Canonical Ltd. *AppArmor*.
http://apparmor.net. 2007.

[56] Zohar Manna, Nikolaj Bjørner, Anca Browne, Edward Chang, Michael Colón, Luca de Alfaro, Harish Devarajan, Arjun Kapur, Jaejin Lee, Henny Sipma, et al. "STeP: The Stanford Temporal Prover." In: *TAPSOFT'95: Theory and Practice of Software Development*. Springer, 1995, pp. 793–794.

[57] Catalin Marinas. *Linux on AArch64 ARM 64-bit Architecture*. https://events.linuxfoundation.org/images/stories/pdf/lcna_co2012_marinas.pdf. 2012.

[58] Microsoft. *Data Execution Prevention (DEP)*. http://support.microsoft.com/kb/875352/EN-US/. 2006.

[59] Tilo Müller, Felix C Freiling, and Andreas Dewald. "TRESOR Runs Encryption Securely Outside RAM." In: *USENIX Security Symposium*. 2011, pp. 17–17.

[60] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C." In: *30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. 2009.

[61] National Security Agency. *Security-Enhanced Linux (SELinux)*.

[62] *OS X Yosemite: Core Technologies Overview*. https://www.apple.com/nz/osx/pdf/OSXYosemite_TO_FF1.pdf. Applie Inc., 2014.

[63] Elliott I Organick. *The multics system: an examination of its structure*. MIT press, 1972.

[64] The LLVM Project. *Clang Static Analyzer*. http://clang-analyzer.llvm.org/.

[65] *QubesOS*. https://www.qubes-os.org.

[66] *Reflections on Trusting TrustZone*. https://www.blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-on-Trusting-TrustZone.pdf. Azimuth Security, 2014.

[67] Xiaoyu Ruan. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, 2014.

[68] Mark Russinovich. *Inside the Windows Vista Kernel: Part 3*. https://technet.microsoft.com/en-us/magazine/2007.04.vistakernel.aspx. 2007.

[69] Joanna Rutkovska. *x86 considered harmful*. http://blog.invisiblethings.org/papers/2015/x86_harmful.pdf. 2015.

[70] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes." In: *ACM SIGOPS Operating Systems Review* 41.6 (2007), pp. 335–350.

[71] 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.

[72] Hovav 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.

[73] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza 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.

[74] Brad Spengler. *KASLR: An Exercise in Cargo Cult Security*. http://forums.grsecurity.net/viewtopic.php?f=7&t=3367. 2013.

[75] Andrew S Tanenbaum. *Modern operating systems*. 3rd. Pearson Education, 2009.

[76] Hacking Team. *Android SELinux Kernel Waiter Exploit*. https://github.com/hackedteam/core-android-native/blob/master/selinux_native/jni/kernel_waiter_exploit/selinux4_exploit.c. 2014.

[77] The PaX Team. *Address space layout randomization*. http://pax.grsecurity.net/docs/aslr.txt. 2000.

[78] Mohammad Tehranipoor and Farinaz Koushanfar. "A survey of hardware Trojan taxonomy and detection." In: (2010).

[79] *The Premium Exploit Acquisition Platform*. https://www.zerodium.com. Zerodium, 2015.

[80] Linus Torvalds. *A semantic parser for C*. https://sparse.wiki.kernel.org/index.php/Main_Page. 2003.

[81] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. W. Freeh, and P. Ning. "On the Expressiveness of Return-into-libc Attacks." In: *14th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID. 2011.

[82] *TrustZone*. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. ARM, 2009.

[83] *Trusted Computer System Evaluation Criteria*. http://csrc.nist.gov/publications/secpubs/rainbow/std001.txt. Department of Defense, 1985.

[84] *Trusted Platform Module*. http://www.trustedcomputinggroup.org/community/2015/12/tcg_tpm_20_library_specification_now_available_from_iso_and_the_iec. Trusted Computing Group, 2003.

[85] *Using the GNU Compiler Collection - Variables in Specified Registers*. https://gcc.gnu.org/onlinedocs/gcc/Explicit-Register-Variables.html. Free Software Foundation, Inc., 2016.

[86] Zhi Wang and Xuxian Jiang. "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity." In: *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE. 2010, pp. 380–395.

[87] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code." In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 157–168.

[88] David A Wheeler. "Fully countering Trusting Trust through diverse double-compiling." In: *arXiv preprint arXiv:1004.5534* (2010).

[89] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K Iyer. "Transparent runtime randomization for security." In: *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*. IEEE. 2003, pp. 260–269.

[90] Jeffrey S. Foster et. al. *CQUAL*. https://www.cs.umd.edu/~jfoster/cqual/. 2004.

[91] Laurie Hendren et. al. *Soot*. http://sable.github.io/soot/. 1999.

[92] Serkan Özkan. *Linux Kernel Security Vulnerabilities Published In 2014*. http://www.cvedetails.com/vulnerability-list.php?vendor_id=33&product_id=47&year=2014&opec=1&opgpriv=1. 2015.

This document was produced using free software, such as the typesetting system LaTeX, the extensible text editor Emacs, and the operating system GNU/Linux. The typographical style was inspired by Robert Bringhurst's seminal book *"The Elements of Typographic Style"* and realized through André Miede's useful template.