CyBOK

OPERATING SYSTEMS AND VIRTUALISATION SECURITY

KNOWLEDGE AREA

(DRAFT FOR COMMENT)

AUTHOR: Herbert Bos – Vrije Universiteit Amsterdam

EDITOR: Andrew Martin – Oxford University

REVIEWERS:

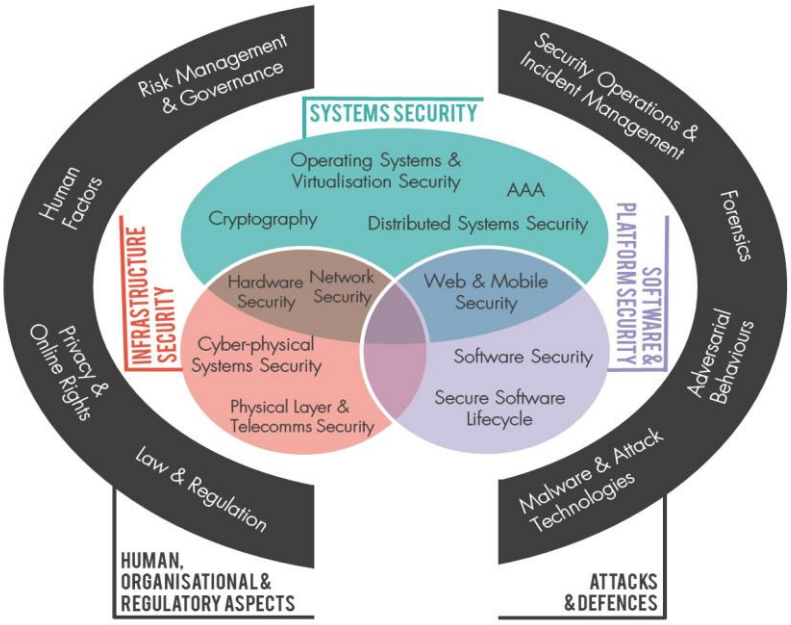
Chris Dalton – Hewlett Packard

David Lie – University of Toronto

Gernot Heiser – University of New South Wales

Mathias Payer – École Polytechnique Fédérale de Lausanne

Following wide community consultation with both academia and industry, 19 Knowledge Areas (KAs) have been identified to form the scope of the CyBOK (see diagram below). The Scope document provides an overview of these top-level KAs and the sub-topics that should be covered under each and can be found on the project website: https://www.cybok.org/.



We are seeking comments within the scope of the individual KA; readers should note that important related subjects such as risk or human factors have their own knowledge areas.

It should be noted that a fully-collated CyBOK document which includes issue 1.0 of all 19 Knowledge Areas is anticipated to be released by the end of July 2019. This will likely include updated page layout and formatting of the individual Knowledge Areas.

Operating Systems and Virtualisation Security

Herbert Bos
Vrije Universiteit Amsterdam

April 2019

INTRODUCTION

In this knowledge area, we introduce the principles, primitives and practices for ensuring security at the operating system and hypervisor levels. We shall see that the challenges related to operating system security have evolved over the past few decades, even if the principles have stayed mostly the same. For instance, when few people had their own computers and most computing was done on multiuser (often mainframe-based) computer systems with limited connectivity, security was mostly focused on isolating users or classes of users from each other[1]. Isolation is still a core principle of security today. Even the entities to isolate have remained, by and large, the same. We will refer to them as security domains. Traditional security domains for operating systems are processes and kernels, and for hypervisors virtual machines (VMs). Athough we may have added trusted execution environments and a few other security domains in recent years, we still have the kernel, user processes and virtual machines as the main security domains today. However, the threats have evolved tremendously, and in response, so have the security mechanisms.

As we shall see, some operating systems (e.g., in embedded devices) do not have any notion of security domains whatsoever, but most distinguish between multiple security domains such as the kernel, user processes and trusted execution environments. In this knowledge area, we will assume the presence of multiple, mutually non-trusting security domains. Between these security domains, operating systems manage a computer system's resources such as CPU time (through scheduling), memory (through allocations and address space mappings) and disk blocks (via file systems and permissions). However, we shall see that protecting such traditional, coarse-grained, resources is not always enough and it may be necessary to manage explicitly the more low-level resources as well. Examples include caches, TLBs (translation lookaside buffers), and a host of other shared resources. Recall that Saltzer and Schroeder's Principle of Least Common Mechanism [1] states that every mechanism shared between security domains may become a channel through which sensitive data may leak—and, indeed, all of the above shared resources have served as side channels to leak sensitive information in attack scenarios.

As the most privileged components, operating systems and hypervisors play a critical role in making systems (in)secure. For brevity, we mainly use the term operating system and processes in the remainder of this knowledge area and refer to hypervisors and VMs explicitly where the distinction is important.

While security goes beyond the operating system, the lowest levels of the software stack form the bedrock on which security is built. For instance, the operating system may be capable of executing privileged instructions not available to ordinary user programs and typically offers the means to authenticate users and to isolate the execution and files of different users. While it is up to the application to enforce security beyond this point, the operating system guarantees that non-authorised processes cannot access its files, memory, CPU time, or other resources. These security guarantees

[1] A situation, incidentally, that is not unlike that of shared clouds today.

are limited by the capabilities of the hardware. For instance, if a CPU's instruction set architecture (ISA) does not have a notion of multiple privilege levels or address space isolation to begin with, shielding the security domains from each other is difficult—although it may still be possible using language-based protection (as in the experimental Singularity operating system [2]).

The security offered *by* the operating system is also threatened by attacks that aim to evade the system's security mechanisms. For instance, if the operating system is responsible for the separation between processes and the operating system itself gets compromised, the security guarantees are void. Thus, we additionally require security *of* the operating system.

After explaining the threat model for operating system security, we proceed by classifying the different design choices for the underlying operating system structure (monolithic versus microkernel-based, multi-server versus libraryOS etc.), which we then discuss in relation to fundamental security principles and models. Next, we discuss the core primitives that operating systems use to ensure different security domains are properly isolated and access to sensitive resources is mediated. Finally, we describe important techniques that operating systems employ to harden the system against attacks.

CONTENT

1 Attacker model

We assume that attackers are interested in violating the security guarantees provided by the operating system or hypervisor: leak confidential data (e.g., crypto keys), modify data that should not be accessible (e.g., to elevate privileges) or limit the availability of the system and its services (e.g., by crashing the system or hogging its resources). In this knowledge area, we focus on the technical aspects of security, leaving aside insider threats, human behaviour, physical attacks, project management, company policies etc. Not because they are not important, but because they are beyond OS control and would require a knowledge area of their own. Table 1 lists some of the threats and attack methods that we do consider.

The simplest way to compromise the system is to inject a malicious extension into the heart of the operating system. For instance, in monolithic systems such as Linux and Windows, this could be a malicious driver or kernel module, perhaps inadvertently loaded as a Trojan, that has access to all privileged functionality [3]. To maintain their hold on the system in a stealthy manner regardless of what the operating system or hypervisor may do, the attackers may further infect the system's boot process (e.g., by overwriting the master boot record or the Unified Extensible Firmware Interface, or UEFI, firmware)—giving the malicious code control over the boot process on every reboot, even *before* the operating system runs, allowing it to bypass any and all operating system level defenses [4].

Besides using Trojans, attackers frequently violate the security properties without any help from the user, by exploiting vulnerabilities. In fact, attackers may use a wide repertoire of methods. For instance, they commonly abuse vulnerabilities in the software, such as memory errors [5] to change code pointers or data in the operating system and violate its integrity, confidentiality or availability. By corrupting a code pointer, they control where the program resumes execution after the call, jump or return instruction that uses the corrupted code pointer. Changing data or data pointers opens up other possibilities, such as elevating the privilege level of an unprivileged process to 'root' (administrator privileges) or modifying the page tables to give a process access to arbitrary memory pages. Likewise, they may use such bugs to leak information from the operating system by changing which or how much data is returned for a system call, or a network request.

Attackers may also abuse vulnerabilities in hardware, such as the Rowhammer bug present in many DRAM chips [6]. Since bits in memory chips are organised in rows and packed very closely together, accessing a bit in one row may cause the neighbouring bit in the adjacent row to leak a small amount of charge onto its capacitor—even though that bit is in a completely different page in memory. By repeatedly accessing the row at high frequency ('hammering'), the interference accumulates so that,

Attack	Description
Malicious extensions	Attacker manages to convince the system to load a malicious driver or kernel module (e.g., as a Trojan).
Bootkit	Attacker compromises the boot process to gain control even before the operating system gets to run.
Memory errors (software)	Spatial and temporal memory errors allow attackers (local or remote) to divert control flow or leak sensitive information.
Memory corruption (hardware)	Vulnerabilities such as Rowhammer in DRAM allow attackers (local or remote) to modify data that they should not be able to access.
Uninitalised data leakage	The operating system returns data to user programs that is not properly initialised and may contain sensitive data.
Concurrency bugs and double fetch	Example: the operating system uses a value from userspace twice (e.g., a size value is used once to allocate a buffer and later to copy into that buffer) and the value changes between the two uses.
Side channels (hardware)	Attackers use access times of shared resources such as caches and TLBs to detect that another security domain has used the resource, allowing them to leak sensitive data.
Side channels (speculative)	Security checks are bypassed in speculative or out-of-order execution and while results are squashed they leave a measurable trace in the microarchitectural state of the machine.
Side channels (software)	Example: when operating systems / hypervisors use features such as memory deduplication, attackers can measure that another security domain has the same content.
Resource depletion (DoS)	By hogging resources (memory, CPU, buses, etc.), attackers prevent other programs from making progress, leading to a denial of service.
Deadlocks/hangs (DoS)	The attacker brings the system to a state where no progress can be made for some part of the software, e.g., due to a deadlock (DoS).

Table 1: Known attack methods / threats to security for modern operating systems

in some cases, the neighbouring bit may flip. We do not know in advance which, if any, of the bits in a row will flip, but once a bit flips, it will flip again if we repeat the experiment. If attackers succeed in flipping bits in kernel memory, they enable attacks similar to those based on software-based memory corruption. For instance, corrupting page tables to gain access to the memory of other domains.

Another class of attacks is that of concurrency bugs and double fetch [7]. The double fetch is an important problem for an operating system and occurs when it uses a value from userspace twice (e.g., a size value is used once to allocate a buffer and later to copy into that buffer). Security issues such as memory corruption arise if there is a race between the operating system and the attacker, and the attacker changes the userspace value in between the two accesses and makes it smaller. It is similar to a TOCTOU (time-of-check-to-time-of-use) attack, except that the value modified is *used* twice.

In addition to direct attacks, adversaries may use side channels to leak information indirectly. One famous family of hardware side channels abuses speculative and out-of-order execution [8, 9]. For performance, modern CPUs may execute instructions ahead of time—before the preceding instructions have been completed. For instance, while waiting for the condition of a conditional branch to be resolved, the branch predictor may speculate that the outcome will be 'branch taken' (because that was the outcome for the last n times), and speculatively execute the instructions corresponding to the branch taken. If it turns out that it was wrong, the CPU will squash all the results of the speculatively executed instructions, so that none of the stores survive in registers or memory. However, there may still be traces of the execution in the microarchitectural state (such as the content of caches, TLBs and branch predictors that are not directly visible in the instruction set architecture). For instance, if a speculative instruction in a user program reads a sensitive and normally inaccessible byte from memory in a register and subsequently uses it as an offset in a userspace array, the array element at that offset will be in the cache, even though the value in the register is squashed as soon as the CPU discovers that it should not have allowed the access. The attacker can time the accesses to every element in the array and see if one is significantly faster (in the cache). The offset of that element will be the secret byte.

In fact, this last-phase cache sidechannel is also used on its own [10]. For instance, attackers can fill a cache set with their own data or code and then periodically access these addresses. If any of the accesses is significantly slower, they will know that someone else, presumably the victim, also accessed data/code that falls in the same cache set. Now assume that the victim code calls functions in a secret dependent way. For instance, an encryption routine processes the key bit by bit and calls function `foo` if the bit is 0, and `bar` if it is 1, where `foo` and `bar` are in different cache sets. By monitoring which cache sets are used by the side channel, the attackers quickly learn the key.

Besides caches, hardware side channels can use all kinds of shared resources, including TLBs, MMUs, and many other components [11]. However, side channels need not be hardware related at all. For instance, memory deduplication and page caches, both implemented in the operating system, are well-known sources for side channels. Focusing on the former for illustration purposes, consider a system that aggressively deduplicates memory pages: whenever it sees two pages with the same content, it adjusts the virtual memory layout so that both virtual pages point to the same physical page. This way, it needs to keep only one of the physical pages to store the content, which it can share in a copy-on-write fashion. In that case, a write to that page takes longer (because the operating system must copy the page again and adjust its page table mappings), which can be measured by an attacker. So, if a write to page takes significantly longer, the attacker knows that some other program also has a copy of that content—a side channel that tells the attacker something about a victim's data. Researchers have shown that attackers may use such coarse-grained side channels to leak even very fine-grained secrets [12]. In many of the side channels, the issue is a lack of isolation between security domains in software *and* in hardware (e.g., there may be no or too little isolation during hardware-implemented speculative execution). It is important to realise that domain isolation issues extend to the hardware/software interface.

For confidentiality in particular, information leaks may be subtle and seemingly innocuous, and still lead to serious security problems. For instance, the physical or even virtual addresses of objects may not look like very sensitive information, until we take into account code reuse or Rowhammer attacks that abuse knowledge of the addresses to divert control flow to specific addresses or flip specific bits.

As for the origin of the attacks, they may be launched from local code running natively on the victim's machine in user space, (malicous) operating system extensions, scripting code fetched across the network and executed locally (such as JavaScript in a browser), malicious peripherals, or even remote systems (where attackers launch their exploit across the network). Clearly, a remote attack is harder to carry out than a local one.

In some cases, we explicitly extend the attacker model to include malicious operating systems or malicious hypervisors as well. These attackers may be relevant in cloud-based systems, where the cloud provider is not trusted, or in cases where the operating system itself has been compromised. In these cases, the goal is to protect the sensitive application (or a fragment thereof), possibly running in special hardware-protected trusted execution environments or enclaves, from the kernel or hypervisor.

A useful metric for estimating the security of a system is the *attack surface* [13]—all the different points that an attacker can reach and get data to or from in order to try and compromise the system. For instance, for native code running locally, the attack surface includes all the system calls the attacker can execute as well as their arguments and return values, together with all the code implementing them that the attacker can reach. For remote attackers, the attack surface includes the network device drivers, part of the network stack, and all the application code handling the request. For malicious devices, the attack surface may include all the memory the device may access using DMA or the code and hardware functions with which the device may interact. Note, however, that the exposure of more code to attackers is only a proxy metric, as the quality of the code differs. In an extreme case, the system is formally verified so that a wide range of common vulnerabilities are no longer possible.

2 The role of operating systems and their design in security

At a high level, operating systems and hypervisors are tasked with managing the resources of a computer system to guarantee a foundation on which it is possible to build secure applications with respect to confidentiality, integrity and availability.

The main role of these lowest layers of the software stack with respect to security is to provide isolation of security domains and mediation of all operations that may violate the isolation. In the ideal case, the operating system shields any individual process from all other processes. For instance, they should not be able to access the memory allocated to it, learn anything about its activities except those which the process chooses to reveal explicitly, or prevent it from using its allocated resources, such as CPU time indefinitely. Some operating systems may even regulate the information flows such that top secret data can never leak to processes without the appropriate clearance, or classified data cannot be modified by processes without the appropriate privilege levels.

There are many ways to design an operating system. Fig. 1 illustrates four extreme design choices. In Fig. 1(a), the operating system and the application(s) run in a single security domain and there is no isolation whatsoever. Early operating systems worked this way, but so do many embedded systems today. In this case, there is little to no isolation between the different components in the system and an application can corrupt the activities of the file system (FS), the network stack, drivers, or any other component of the system.

Fig. 1(b) shows the configuration of most modern general-purpose operating systems, where most of the operating system resides in a single security domain, strictly isolated from the applications, while each application is also isolated from all other applications. For instance, this is the structure of Windows Linux, OS X and many of the descendants of the original UNIX [14]. Since every component of

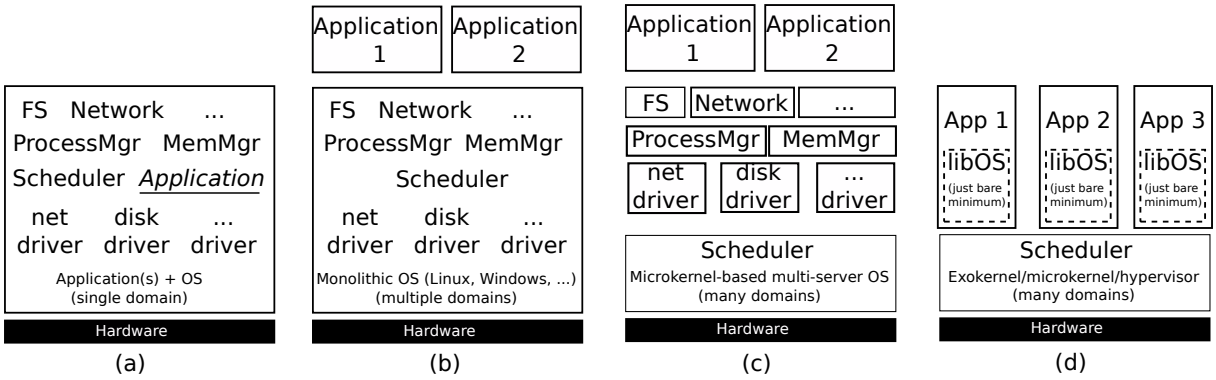


Figure 1: Extreme design choices for operating systems: (a) single domain (sometimes used in embedded systems), (b) monolithic OS (Linux, Windows, and many others), (c) microkernel-based multi-server OS (e.g., Minix-3) and (d) Unikernel / Library OS

the operating system runs in a single security domain, the model is very efficient (because the components interact simply by function calls and shared memory) and safe, as long as every component is benign. If attackers manage to compromise even a single component (say a driver), however, all security is void. In general, device drivers and other operating system *extensions* (e.g., Linux Kernel Modules) are important considerations for the security of a system. Often written by third parties and more buggy than the core operating system code, extensions running in the single security domain of the operating system may compromise the security of the system completely.

Fig. 1(c) shows the extreme breakup in separate processes of all the components that make up the operating system in a multi-server operating system [15, 16]. The configuration is potentially less efficient than the previous model, because all the interactions between different components of the operating system involve inter-process communication (IPC). In addition, the operating system functions as a distributed system and anyone who has ever built a distributed system knows how complicated the problems may get. However, the advantage of a multi-server system is that a compromised driver, say, cannot so easily compromise the rest of the system. Also, while conceptually the multi-server looks like a distributed system, a lot of the complexity of a real distributed system is due to the unreliability of communication and this does not exist in mult-iserver systems. The common view is that microkernel-based multi-server designs have security and reliability advantages over monolithic and single-domain designs, but incur somewhat higher overheads—the price of safety.

Finally, Fig. 1(d) shows a situation that, at first glance, resembles that of Fig. 1(a): on top of a minimal kernel that multiplexes the underlying resources, applications run together with a minimal 'library operating system' (libOS [17, 18]). The libOS contains code that is typically part of the operating system, but is directly included with the application. This configuration allows applications to tailor the operating system exactly according to their needs and leave out all the functionality they were not going to use anyway. Library operating systems were first proposed in the 1990s (e.g., in MIT's Exokernel and Cambridge's Nemesis projects). After spending a few years in relative obscurity, they are becoming popular again—especially in virtualised environments where they are commonly referred to as Unikernels. In terms of security, Unikernels are difficult to compare against, say, microkernel-based multi-server systems. On one hand, they do not have the extreme separation of operating system components, while on the other, they allow the (library) operating system code to be much smaller and less complex—it only has to satisfy the needs of this one application. Moreover, the library cannot compromise isolation: it may be part of this application's trusted computing base, it is not in anyone else's.

Which design is better has been the topic of considerable debate, some of which goes back to the famous *flame war* between Andrew S. Tanenbaum and Linus Torvalds in 1992. By that time, MINIX [19], a small UNIX-like operating system developed by Tanenbaum, had been around for half a decade or

so, and was gaining traction as an education operating system around the world—especially since Bell Labs' original UNIX was sold as a commercial product with a restrictive license prohibiting users from modifying it. One of MINIX's users was Torvalds, then a Finnish student who announced a new operating system kernel in a post in the `comp.os.minix` newsgroup on Usenet. In January 1992, Tanenbaum criticised the design for its lack of portability, and also took aim at Linux's monolithic design, claiming Linux was obsolete from the outset. Torvalds responded with his own criticism of MINIX. This heated exchange contained increasingly sophisticated arguments, many of which still stand today.

Who won the debate is itself a topic of active debate to this day. What is clear is that Linux has become wildly popular and few people would consider it obsolete, but also that ideas from multi-server systems such as MINIX have been incorporated into existing operating systems and hypervisor-based systems. Interestingly, at the time of writing even MINIX itself is running in hundreds of millions of Intel processors as a miniature operating system on a separate microprocessor known as the Management Engine. In addition, now that the CPUs in modern systems are increasingly elaborate systems-on-chip (SOCs), the hardware itself is starting to look like a distributed system and some researchers explicitly advocate designing the operating system accordingly, with a focus on message passing rather than memory sharing for communication [20].

The situation for virtualised environments, in general, is comparable to that of operating systems. We have already seen that in one extreme case, the entire virtual machine with the application and a (stripped-down) operating system can form a single domain. A more common case is to have a hypervisor at the lowest level supporting one or more operating systems such as Linux or Windows in a *virtual machine*. In other words, these hypervisors provide each of the operating systems with the illusion that they run on dedicated hardware. At the other end of the spectrum, we again find the entire system decomposed into separate, relatively small, virtual machines. Indeed, some operating systems, such as QubesOS completely integrate the concepts of virtualisation and operating systems by allowing individual user processes to be isolated in their own virtual machines. Finally, as we have already seen, Unikernels are popular in virtualised environments, on top of hypervisors.

Incidentally, one of the drawbacks of virtual machines is that each operating system image uses storage and adds redundancy, as every system will think that it is the king of the hardware mountain, while in reality it is sharing resources. Moreover, each operating system in a virtual machine needs separate maintenance: updates, configuration, testing etc. A popular alternative is, therefore, to virtualise at the operating system level. In this approach, multiple environments, known as *containers*, run on top of a single shared operating system. The containers are isolated from each other as much as possible (and have their own kernel name spaces, resource limits etc.), but ultimately share the underlying operating system kernel, and often binaries and libraries. Compared to virtual machines, containers are much more lightweight. However, if we ignore the management aspects for a moment, virtual machines are often perceived as more secure than containers, as they partition resources quite strictly and share only the hypervisor as a thin layer between the hardware and the software. On the other hand, some people believe that containers are more secure than virtual machines, because they are so lightweight that we can break applications into 'microservices' with well-defined interfaces in containers. Moreover, having fewer things to keep secure reduces the attack surface overall. Early work on containers (or 'operating system level virtualisation" is found in the `chroot` call that was first added to Version 7 Unix in 1979 [21]. Today, we have many container implementations. A popular one is Docker [22].

A final class of operating systems explicity targets small and resource constrained devices such as those found in the Internet of Things (IoT). While everybody has a different opinion on what IoT means and the devices to consider range from smartphones to smart dust, there is a common understanding that the most resource constrained devices should be part of it. For such devices, even stripped down general-purpose operating systems may be too bulky and operating systems are expected to operate in just a few kilobytes. As an extreme example, popular IoT operating systems such as RIOT can be

Principle of...
Economy of mechanism
Fail-safe defaults
Complete mediation
Open design
Separation of privilege
Least privilege / least authority
Least common mechanism
Psychological acceptability

Table 2: Saltzer & Schroeder's security principles [1]

less than 10 KB in size and run on systems ranging from 8-bit microcontrollers to general-purpose 32 CPUs, with or without advanced features such as memory management units (MMUs) etc. The abundance of features and application isolation that we demand from operating systems such as Windows and Linux may be absent in these operating systems, but instead there may be support for functionality such as real-time schedule or low-power networking. which is important in many embedded systems.

Since we are interested in the security guarantees offered by the operating system, we will assume that there are multiple security domains. In the next section, we will elaborate on the advantages and disadvantages of the different designs from the viewpoint of well-established security principles. Our focus will be on the security of the design and the way in which we can stop attacks, but not before observing that there is more to security at this level. In particular, management and maintainability of the system—with respect to updates, extensions, configuration etc.—play an important role.

3 Operating System Security Principles and Models

Since operating systems (and/or hypervisors) are the foundation on which rests the security of all higher-level components in the system, it is common to hear their designs debated in terms of security principles such as those of Saltzer and Schroeder (see Table 2), and security models such as the Bell-LaPadula [23] and Biba [24] access models—the topic of the next few subsections.

3.1 Security principles in operating systems

From a security perspective, the walls between different security domains should be as high and as thick as possible—perfect isolation. Any interaction between domains should be subject to rigorous mediation, following the Principle of Complete Mediation, and security domains should have as few mechanisms (especially those involving a shared state such as global variables) in common as possible, adhering to the Principle of Least Common Mechanism. For instance, given a choice between adding a shared procedure with global variables to the operating system kernel and making it available in a user-space library that behaves in an isolated manner for each process, we should choose the latter option (assuming it does not increase the code size too much or violate any other principles or constraints). Moreover, mediation should follow the Principle of Fail-Safe Defaults: the policy for deciding whether domains can access the resources of other domains should be: 'No, unless'. In other words, only explicitly authorised domains should have access to a resource. The principles of Least Common Mechanism and Economy of Mechanism also dictate that we should minimise the amount of code that should be trusted, the *Trusted Computing Base (TCB)*. Since studies have shown that even good programmers introduce between 1 and 6 bugs per 1000 lines of code, a small TCB translates to fewer bugs, a smaller attack surface and a better chance of automatically or manually verifying the correctness of the TCB with respect to a formal specification.

With respect to the designs in Fig. 1, we note that if there is a single domain, the TCB comprises

all the software in the system, including the applications. All mechanisma are “common” and there is virtually no concept of fail-safe defaults or rigorously enforced mediation. For the monolithic OS design, the situation is a little better, as at least the operating system is shielded from the applications and the applications from each other. However, the operating system itself is still a single security domain, inheriting the disadvantages of Fig. 1(a). The extreme decomposition of the multi-server operating system is more amenable to enforcing security: we may enforce mediation between individual operating components in a minimal-size microkernel with fail-safe defaults. Much of the code that is in the operating system’s security domain in the other designs, such driver code, is no longer part of the TCB. Unikernels are an interesting alternative approach: in principle, the operating system code and the application run in a single domain, but the libOS code is as small as possible (Economy of Mechanism) and the mechanism common to different applications is minimized. Resource partitioning can also be mediated completely at the Unikernel level. For a Unikernel application, the TCB consists only of the underlying hypervisor/Exokernel and the OS components it decides to use. Moreover, the library implementing the OS component is only in this application’s TCB, as it is not shared by others.

Another principle, that of Open Design, is perhaps more controversial. In particular, there have been endless discussions about open source (which is one way to adhere to the principle) versus closed source and their merits and demerits with respect to security. The advantage of an open design is that anybody can study it, increasing the probability of finding bugs in general and vulnerabilities in particular[2]. A similar observation was made by Auguste Kerckhoffs about crypto systems and is often translated as that one should not rely on *security by obscurity*. After all, the obscurity is unlikely to last forever and when bad people find a vulnerability before the good guys do, you may have a real problem. The counter argument is that with an open design, the probability of them finding the bug is higher.

In contrast, there is little doubt that a design with a strict decomposition is more in line with the Principle of Least Privilege and the Principle of Privilege Separation than one where most of the code runs in a single security domain. Specifically, a monolithic system has no true separation of privileges of the different operating system components and the operating system always runs with all privileges. In other words, the operating system code responsible for obtaining the process identifier of the current process runs with the power to modify page tables, create root accounts, modify any file on disk, read and write arbitrary network packets, and crash the entire system at any time it sees fit. Multi-server systems are very different and may restrict what calls individual operating system components can make, limiting their powers to just those privileges they need to complete their job, adhering to the Principle of Least Authority (POLA) with different components having different privileges (Principle of Privilege Separation). Unikernels offer a different and interesting possibility for dealing with this problem. While most of the components run in a single domain (no privilege separation or POLA), the operating system is stripped down to just the parts needed to run the application, and the Unikernel itself could run with just the privileges required for this purpose.

Of course, however important security may be, the Principle of Psychological Acceptability says that in the end the system should still be usable. Given the complexity of operating system security, this is not trivial. While security hardened operating systems such as SELinux and QubesOS offer clear security advantages over many other operating systems, few ordinary users use them and even fewer feel confident to configure the security settings themselves.

3.2 Security models in operating systems

An important question in operating systems concerns the flow of information: who can read and write what data? Traditionally, we describe system-wide policies in so-called access control models.

[2]On the other hand, researchers have encountered security bugs of years or sometimes decades old, even in security critical open source software such as OpenSSL or the Linux kernel, suggesting that the common belief that “given enough eyeballs, all bugs are shallow” (also known as Linus’ Law) does not always work flawlessly.

For instance, the Bell-LaPadula model [23] is a security access model to preserve the confidentiality of information, initially created for the US government. In the 1970s, the US military faced a situation where many users with different clearance levels would all be using the same mainframe computers—requiring a solution known as Multi-Level Security. How could they ensure that sensitive information would never leak to non-authorised personnel? If it adheres to the model designed by David Bell and Leonard LaPadula, a system can handle multiple levels of sensitive information (e.g., *unclassified*, *secret*, *top secret*) and multiple clearance levels (e.g., the clearance to access *unclassified* and *secret*, but not *top secret* data) and keep control over the flow of sensitive infomation. Bell-LaPadula is often characterised as 'read down, write up'. In other words, a subject with clearance level *secret* may create *secret* or *top secret documents*, but not *unclassified ones*, as that would risk leaking secret information. Likewise, a user can only read documents at their own security level, or below it. Declassification, or lowering of security levels (e.g., copying data from a *top secret* to a *secret* document) can only be done explicitly by special, 'trusted' subjects. Strict enforcement of this model prevents the leakage of sensitive information to non-authorised users.

Bell-LaPadula only worries about confidentiality. In contrast, the Biba model [24] arranges the access mode to ensure data *integrity*. Just like in Bell-LaPadula, objects and subjects have a number of levels of integrity and the model ensures that subjects at lower levels cannot modify data at higher levels. This is often characterised as 'read up, write down', the exact opposite of Bell-LaPadula.

Bell-LaPadula and Biba are access control models that the operating system applies when mediating access to resources such as data in memory or files on disk. Specifically, they are *mandatory access control* (MAC) models, where a system-wide policy determines which users have the clearance level to read or write which specific documents, and users are not able to make information available to other users without the appropriate clearance level, no matter how convenient it would be. A less strict access control model is known as *discretionary access control* (DAC), where users with access to an object have some say over who else has access to it. For instance, DAC may restrict access to objects based on a user or process identity or group membership, and more importantly, allow a user or process with access rights to an object to transfer those rights to other users or processes. Having only this group-based DAC makes it hard to control the flow of information in the system in a structured way. However, it is possible to combine DAC and MAC, by giving users and programs the freedom to transfer access rights to others, within the constraints imposed by MAC policies.

For completeness, we also mention role-based access control (RBAC [25]), which restricts access to objects on the basis of roles which may be based on job functions. While intuitively simple, RBAC allows one to implement both DAC and MAC access control policies.

4 Primitives for Isolation and Mediation

In the 1960s, Multics [26] became the first major operating system designed from the ground up with security in mind. While it never became very popular, many of its security innovations can still be found in the most popular operating systems today. Even if some features were not invented directly by the Multics team, their integration in a single, working, security-oriented OS design was still novel. Multics offered rings of protection, virtual memory, segment-based protection, a hierarchical file system with support for discretionary accress control (DAC) *and* mandatory access control (MAC). Indeed, in many ways, the mandatory access control in Multics, added at the request of the military, is a direct software implementation of the Bell-LaPadula security model. Finally, Multics made sure that its many small software components were strongly encapsulated, accessible only via their published interfaces where mediation took place.

If any of this sounds familiar, this is not surprising, as Jerome Saltzer was one of the Multics team leaders. If you read the 'Trusted Computer System Evaluation Criteria' (TCSEC), better known as the famous *Orange Book* [27], which describes requirements for evaluating the security of a computer system, you will find that it was strongly based on Multics. There is no doubt that Multics was

very advanced and perhaps ahead even of some modern operating systems, but this was also its downfall—the system became so big and so complex that you might say it violated the Principle of Psychological Acceptability for at least some of its developers. Frustrated, Ken Thomson and Dennis Ritchie decided to write a new and much simpler operating system. As a pun and to contrast it with Multics, they called it 'Unics', later spelt UNIX. Like all major general purpose operating systems in use today, it relied on a small number of core primitives to isolate its different security domains.

So what are these major isolation primitives? First, the operating system has to have some way of authenticating users and security domains so it can decide whether or not they may access certain resources. To isolate the different security domains, the operating system also needs support for access control to objects (such as files). In addition, it needs memory protection to ensure that a security domain cannot simply read data from another domain's memory. Finally, it needs a way to distinguish between privileged code and non-privileged code, so that (only) the privileged code can configure the desired isolation at the lowest level and guarantee mediation for all operations.

4.1 Authentication and identification

Since authentication is the topic of the AAA knowledge area, we will just observe that to determine access rights, an operating system needs to authenticate its users and that there are many ways to do so. Traditionally, only usernames and passwords were used for this purpose, but more and more systems nowadays also use other methods (such as smartcards, fingerprints, iris scans, or face recognition)—either instead of passwords or as an additional factor. Multi-factor authentication makes it harder for attackers to masquerade as a legitimate user, especially if the factors are of a different nature, e.g., something you know (like a password), something you own (like a smartcard), and something you 'are' (biometric data such as fingerprints).

For every user thus authenticated, the operating system maintains a unique user id. Moreover, it may also keep other information about users such as in which groups they reside (e.g., student, faculty, and/or administrator). Similarly, most operating systems attach some identity to each of the processes running on behalf of the user and track the ownership and access rights of the files they use. For instance, it gives every running process a unique process id and also registers the id of the users on whose behalf it runs (and thus the groups in which the user resides). Finally, it tracks which user owns the executing binary. Note that the user *owning* the binary and the user *running* the binary need not be the same. For instance, the administrator can create and own a collection of system programs, that other users may execute but not modify.

Given these identities, the operating system is equipped to reason about which user and which process is allowed to perform which operations on a specific object: access control.

It is important to observe that access restrictions at the level of the operating system do not necessarily translate to the same restrictions at the physical level. For instance, the operating system may 'delete' a file simply by removing the corresponding metadata that makes it appear as a file (e.g., when listing the directory), without really removing the *content* of the file on disk. Thus, an attacker that reads raw disk blocks with no regard for the file system may still be able to access the data. It turns out that securely deleting data on disk is not trivial. Naive deletion, for instance, by overwriting the original content with zeros, is not always sufficient. For instance, on some magnetic disk data on the disk's tracks leaves (magnetic) traces in areas close to the tracks and a clever attack with sufficient time and resources may use these to recover the content. Moreover, the operating system may have made copies of the file that are not immediately visible to the user, for instance, as a backup or in a cache. All these copies need to be securely deleted. The situation for solid-state drives (SSDs) is no better, as SSDs have their own firmware that decides what to (over)write and when, beyond the control of the OS. For most operating systems, truly secure deletion, in general, is beyond the operating system's capabilities and we will not discuss it further in this knowledge area, except to say that full disk encryption, a common feature of modern operating systems, helps a lot to prevent file

recovery after deletion.

4.2 Access control lists

Multics introduced an access control list (ACL) for every block of data in the system [26]. Conceptually, an ACL is a table containing users and data blocks that specifies for each data block which users have which kind of access rights. Most modern operating systems have adopted some variant of ACLs, typically for the file system. Let us look at an example. On UNIX-based systems [14], the default access control is very simple. Every file is owned by a user and a group. Moreover, every user can be in one or more groups. For instance, on a Linux system, user `herbertb` is in nine different groups:

```
herbertb@nordkapp:~$ groups herbertb
herbertb : herbertb adm cdrom sudo dip plugdev lpadmin sambashare cybok
herbertb@nordkapp:~$
```

Per file, a small number of permission bits indicates the access rights for the owning user, the owning group, and everyone else. For instance, let us look at the ACL for a file called `myscript` on Linux:

```
herbertb@nordkapp:~/tmp$ getfacl myscript
# file: home/herbertb/tmp/myscript
# owner: herbertb
# group: cybok
user::rwx
group::rwx
other::r-x
```

We see that `myscript` is owned by user `herbertb` and group `cybok`. The owning user and all users in group `cybok` have permissions to **r**ead, **w**rite, and e**x**ecute the file, while all other users can **r**ead and e**x**ecute (but not write) it.

These basic UNIX file permissions are quite simple, but modern systems (such as Linux and Windows) also allow for more extensive ACLs (e.g., with explicit access rights for multiple users or groups). Whenever someone attempts to read, write or access a file, the operating system verifies whether the apropriate access rights are in the ACL. Moreover, the access control policy in UNIX is typically discretionary, because the owning user is allowed to set these rights for others. For instance, on the above Linux system, user `herbertb` can himself decide to make the file `myscript` writable by all the users (‘`chmod o+w myscript`’).

Besides DAC, Multics also implemented MAC and, while it took a long time to reach this stage, this is now also true for many of the operating systems that took their inspiration from Multics (namely most popular operating systems today). Linux even offers a framework to allow all sorts of access control solutions to be plugged in, by means of so-called ‘reference monitors’ that vet each attempt to execute a security sensitive operation and, indeed, several MAC solutions exist. The best known one is probably Security-Enhanced Linux (SELinux [28]), a set of Linux patches for security originally developed by the National Security Agency (NSA) in the US and derived from the Flux Advanced Security Kernel (FLASK) [29].

SELinux gives users and processes a context of three strings: `(username, role, domain)`. While the tuple already (correctly) suggests that SELinux also supports RBAC, there is significant flexibility in what and what not to use. For instance, many deployed systems use only the `domain` string for MAC and set `username` and `role` to the same value for all users. Besides processes, resources such as files, network ports, and hardware resources also have such SELinux contexts associated with them. Given this configuration, system administrators may define system-wide policies for access control on their systems. For instance, they may define simply which domains a process have to perform specific operations (read, write, execute, connect, ...) on a resource, but policies may also be much more complicated, with multiple levels of security and strict enforcement of information flow *a la* Bell-LaPadula, Biba, or some custom access-control model.

Mandatory access control in systems such as SELinux revolves around a single system-wide policy that is set by a central administrator and does not change. They do not allow untrusted processes to define and update their own information control policy. In contrast, research operating systems such as Asbestos [30], HiStar [31] and Flume [32] provide exactly that: distributed information flow control. In other words, any process can create security labels and classify and declassify data.

4.3 Capabilities

So far, we have assumed that access control is implemented by means of an ACL or access matrix, where all information is kept to decide whether a process P may access a resource R. After authenticating the users and/or looking up their roles or clearance levels, the reference monitor decides whether or not to grant access. However, this is not the only way. A popular alternative is known as *capability* and, stemming from 1966, is almost as old.

In 1966, Jack Dennis and Earl Van Horn, researchers at MIT, proposed the use of capabilities for access control in an operating system [33]. Unlike ACLs, capabilities do not require a per-object administration with the exact details of who is allowed to perform what operation. Instead, the users present a capability that in itself proves that the requested access is permitted. According to Dennis and Van Horn, a capability should have a unique identifier of the object to which it pertains, as well as the set of access rights provided by the capability. Most textbooks use the intuitive definition by Henry Levy that a capability is a 'token, ticket, or key that gives the possessor permission to access an entity or object in a computer system' [34]. Possession of the capability should give you all the rights specified in it and. whenever a process wants to perform an operation on an object, it should present the appropriate capability. Conversely, users do not have access to any resources other than the ones for which they have capabilities.

Of course, it should be impossible to forge capabilities, lest users give themselves arbitrary access to any object they want. Thus, an operating system should either store the capability in a safe place (for instance, the kernel), or protect it from forgery using dedicated hardware or software-based encryption. For instance, in the former case, the operating system may store a process' capabilities in a table in protected memory, and whenever the process wants to perform an operation on a file, say, it will provide a reference to the capability (e.g., a file descriptor) and never touch the capability directly. In the latter case, the capability may be handled by the process itself, but any attempt to modify it in an inappropriate manner will be detected [35].

Capabilities are very flexible and allow for convenient delegation policies. For instance, given full ownership of a capability, a process may pass it on to another process, to give that process either the same access rights, or, alternatively, a subset of those rights. Thus, discretionary access control is easy. On the other hand, in some situations you may not want capabilities to be copied and spread arbitarily. For this reason, most capability-based systems add a few bits to the capabilities to indicate such restrictions: whether copying is permitted, whether the capability's lifetime should be limited to a procedure invocation, etc.

Comparing ACLs and capabilities, we further observe that ACLs are typically based on users ('the user with id x is allowed to read and write'), while capabilities can be extremely fine-grained. For instance, we may use different capabilities for sending and receiving data. Following the Principle of Least Authority, running every process with the full power of the user, compared to running a process with just the power of the capabilities it acquires, is less secure. Moreover, capabilities do not even allow a process to name an object unless it has the appropriate capability, while ACLs should permit the naming of any object by everyone, as the access check only occurs when the process attempts the operation. Finally, ACLs may become very large with growing numbers of users, access rights, and objects.

On the other hand, revoking a particular access right for a particular user in an ACL is easy: just remove a permission in the appropriate table entry. With capabilities, the process is more involved.

After all, we may not even know which users/processes have the capability. Adding a level of indirection may help somewhat. For instance, we could make the capabilities point to an indirect object, which in turn points to the real object. To invalidate the capability (for *all* users/processes) the operating system could then invalidate that indirect object. But what to do if we only want to revoke the capability in a subset of processes? While there are solutions, revocation of capabilities remains the most difficult part.

Since the 1960s, many capability-based systems have appeared—initially all supported in hardware [34] and typically more rigid than the more general capabilities discussed so far. The first was the MIT PDP-1 Timesharing System [36], followed shortly after by the Chicago Magic Number Machine at the University of Chicago [37], a very ambitious project with hardware-supported capabilities, which, as is not uncommon for ambitious projects, was never completed. However, it did have a great impact on subsequent work, as Maurice Wilkes of the University of Cambridge learned about capabilities during several visits to Chicago and wrote about it in his book on time-sharing systems. Back in the UK, this book was picked up by an engineer at Plessey which built the fully functional Plessey System 250 (with explicit hardware support for capability-based addressing). Maurice Wilkes himself went on to build the Cambridge CAP computer together with Roger Needham and David Wheeler [38]. CAP was the first computer to demonstrate the use of secure capabilities. This machine ensured that a process could only access a memory segment or other hardware if it possessed the required capabilities. Another noteworthy capability-based system of that time was CMU's Hydra [39]—which added explicit support for restricting the use of capabilities or operations on objects (allowing one to specify, for instance, that capabilities must not survive a procedure invocation). Finally, in the 1980s, the Amoeba distributed operating systems explored the use of cryptographically protected capabilities that could be stored and handled by user processes [35].

Nowadays, many major operating systems also have at least some support for capabilities. For instance, the L4 microkernel, which is present in many mobile devices today, embraced capability-based security in the version by Gernot Heiser's group at NICTA in 2008[3]. A formally verified kernel called seL4 [40] from the same group similarly relies on capabilities for access control to resources. In 1997, Linux adopted very limited capability support (sometimes referred to as 'POSIX capabilities'), but this was different from the capabilities defined by Dennis and Van Horn (with less support for copying and transferring capabilities). For instance, Linux capabilities refer only to operations, not objects. An interesting effort to merge capabilities and UNIX APIs is the Capsicum project [41] by the University of Cambridge and Google, where the capabilities are extensions of UNIX file descriptors. FreeBSD adopted Capsicum in version 9.0 in 2012.

4.4 Memory protection and address spaces

Access control is only meaningful if security domains are otherwise isolated from each other. For this, we need separation of the security domains' data according to access rights and a privileged entity that is able to grant or revoke such access rights. We will look at the isolation first and talk about the privileges later, when we introduce protection rings.

A process should not normally be able to read another process' data without going through the appropriate access control check. Multics and nearly all of the operating systems that followed (such as UNIX and Windows) isolate information in processes by giving each process (a) its own processor state (registers, program counter etc.) and (b) its own subset of memory. Whenever the operating system decides to execute process P_2 at the expense of the currently running process P_1 (a so-called context switch), it first stops P_1 and saves all of its processor state in memory in an area inaccessible to other processes. Next, it loads P_2's processor states from memory into the CPU, adjusts the bookkeeping that determines which parts of the physical memory are accessible, and starts executing P_2 at the address indicated by the program counter that it just loaded as part of the

[3] Incidentally, other L4 variants, such as the L4 Fiasco kernel from Dresden, also supported capabilities.

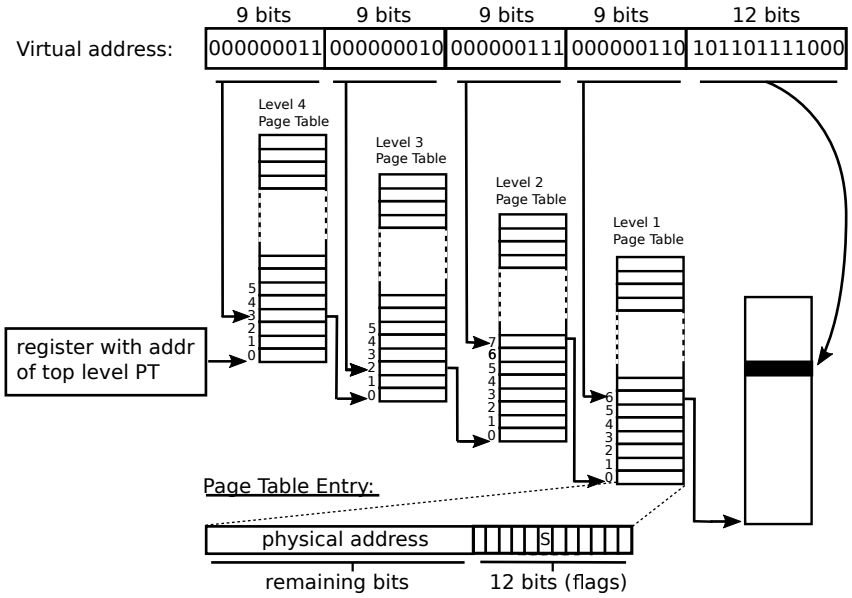


Figure 2: Address translation in modern processors. The MMU 'walks' the page tables to find the physical address of the page. Only if a page is 'mapped' on a process' page tables can the process address it, assuming it is present and the process has the appropriate access rights. Specifically, a user process cannot access the page for which the supervisor (*S*) bit is set in the page table entry.

processor state. Since user processes cannot directly manipulate the bookkeeping themselves, $P2$ cannot access any of P_1's data in a non-mediated form.

Most modern operating systems keep track of the memory bookkeeping by means of *page tables*, as illustrated in Fig. 2. For each process, they maintain a set of page tables (often containing multiple levels organised as a directed acyclic graph[4]), and store a pointer to the top level page table in a register that is part of the processor state and that must be saved and restored on a context switch.

The main use for the page table structure is to give every process its own *virtual* address space, ranging from address 0 to some maximum address (e.g., 2^{48}), even though the amount of physical memory may be much less [42, 43, 44]. Since two processes may both store data at address 0x10000, say, but should not be allowed to access each others' data, there has to be a mapping from the virtual addresses each process uses to the physical addresses used by the hardware. It is like a game of basketball, where each side may have a player with the number 23, but that number is mapped onto a different physical player for each team.

This is where the page tables comes in. We divide each of the virtual address spaces into fixed size pages and use the page table structure to map the address of the first byte of a virtual page onto a physical address. The processor often uses multiple levels of translation. In the example in Fig. 2, it uses the first nine bits of the virtual address as an index in the top level page table (indicated by a control register that is part of the processor state) to find an entry containing the physical address of the next level page table, which is indexed by the next nine bits, and so on, until we reach the last level page table, which contains the physical address of the physical page that contains the virtual address. The last 12 bits of the virtual address are simply the offset in this page and point to the data.

Paging allows the (total) size of the virtual address spaces of the processes to be much larger than the physical memory available in the system. First, a process typically does not use all of its possibly gigantic address space and only virtual pages that are in actual use need backing by physical pages. Second, if a process needs more memory to store some data and no physical pages are free at that moment (for instance, because they are already in use by other processes, or they are backing some

[4]While it is often helpful to think of page table structures as trees, different branches may point to the same leave nodes.

other virtual pages of this process), the operating system may swap the content of these pages to disk and then re-use the physical page to store the new data.

A key consequence of this organisation is that a process can only access data in memory if there is a mapping for it in its page tables. Whether this is the case, is controlled by the operating system, which is, therefore, able to decide exactly what memory should be private and what memory should be shared and with whom. The protection itself is enforced by specialised hardware known as the memory management unit (MMU). If the mapping of virtual to physical for a specific address is not in the small but very fast cache known as the translation lookaside buffer (TLB), the MMU will look for it by walking the page tables and then triggering an interrupt if the page containing the address is not mapped.

The MMU will also trigger interrupts if the page is currently not in memory (swapped to disk), or, more relevant to security, if the user does not have the required privilege to access this memory. Specifically, the last 12 bits of the page table entry (PTE) contain a set of flags and one of these flags, the *S* bit in Fig. 2, indicates whether this is a page for supervisor code (say, the operating system running at the highest privilege) or for ordinary user processes. We will have more to say about privileges later.

Page tables are the main way modern operating systems control access to memory. However, some (mostly older) operating systems additionally use another trick: *segmentation*. Not surprisingly, one of the earliest operating systems using both segmentation and paging was Multics [26, 44]. Unlike pages, segments have an arbitrary length and start at an arbitrary address. However, both depend on hardware support: an MMU. For instance, processors such as Intel's 32 bits x86 have a set of dedicated registers known as segment selectors: one for code, one for data etc. Each segment has specific permissions, such as read, write, or execute. Given a virtual address, the MMU uses the current value in the corresponding segment selector as an index in a so-called descriptor table. The entry in the descriptor table contains the start address and length of the segment, as well as protection bits to prevent code without the required privilege level to access it. In case there is only segmentation (and no paging), the resulting address is the original virtual address added to the start of the segment and that will be the physical address, and we are done. However, both the GE-645 mainframe computer used for Multics and the more modern x86-32 allow one to combine segmentation and paging. In that case, the virtual address is first translated into a so-called *linear address* using the segment descriptor table and that linear address is then translated into a physical address using the page table structure.

If you think this is complicated, you are right; none of the popular modern operating systems still uses segmentation. The best known examples of operating systems using segmentation were OS/2 (an ill-fated collaboration between Microsoft and IBM that started in the mid-1980s and that never caught on) and IBM's AS/400 (also launched in the 1980s[5] and still running happily today on a mainframe near you). In fact, the 64-bit version of the Intel x86 no longer even supports full segmentation (although some vestiges of its functionality remain). On the other hand, complicated multi-level address translation is still quite common in virtualised environments. Here, the hypervisor tries to give virtual machines the illusion that they are running all by themselves on real hardware, so the MMU translates a virtual address first to what is known as a *guest physical address* (using page tables). However, this is not a real physical address yet, as many virtual machines may have the same idea of using, say, physical address 0x10000. So, instead, the MMU uses a second translation stage (using what Intel refers to as extended page tables, maintained by the hypervisor) to translate the guest physical address to a host physical address ('machine address').

[5] Or even the 1970s, if you want to count the System/38.

4.5 Modern hardware extensions for memory protection

Also, while segmentation is mostly dead, there are many other forms of hardware support for memory protection beyond paging. For instance, many machines have had support for buffer bounds checking and some date back a quarter of a century or more. To illustrate the corresponding primitives, however, we will look at what is available in modern general purpose processors, focusing mostly on the Intel x86 family. The point here is not whether we think this processor is more important or even that feature X or Y will be very important in the future (which is debatable and hard to predict), but rather to illustrate that this is still a very active area for hardware development today.

As a first example, consider the Intel Memory Protection Extensions (MPX) that enhance Intel's workhorse processors with functionality to ensure that array pointers cannot stray beyond the array boundaries (stopping vulnerabilities such as buffer overflows from being exploited). For this purpose, a small set of new registers can store the lower and upper bounds of a small number of arrays, while prior to dereferencing the pointer, new MPX instructions check the value of the array pointer for boundary violations. Even in systems that use MPX only in userspace, the operating system plays a role, for instance, to handle the exception that the hardware throws when it encounters a buffer boundary violation.

Similarly, Intel added Memory Protection Keys (MPKs) in more recent processors[6]. Intel MPK allows one to set four previously unused bits in the PTE (Fig. 2) to one of 16 'key' values. In addition, it adds a new 32-bit register containing 2 bits for each key to indicate whether reading and writing is allowed for pages tagged with that key. MPK allows developers to partition the memory in a small number (in this case 16) protection domain and, for instance, allow only a specific crypto library to access cryptographic keys. While unprivileged user processes may update the value of the register, only privileged operating system code can tag the memory pages with keys.

Meanwhile, some processors, especially in low-power devices, do not have a full-fledged MMU at all. Instead, they have a much simpler memory protection unit (MPU) which serves only to protect memory, in a way that resembles the MPK functionality discussed above. In MPU designs, the operating systems define a number of memory regions with specific memory access permissions and memory attributes. For instance, the MPU on ARMv8-M processors supports up to 16 regions. Meanwhile, the MPU monitors all the processor's memory accesses (including instruction fetches and data accesses) and triggers an exception on detecting an access violation.

Note that in the above, we have assumed that the operating system needs protection from untrusted user applications. A special situation arises when the operating itself is not trusted. Perhaps you are running a security-sensitive application on a compromised operating system, or in the cloud, where you are not sure you want to trust the cloud provider. For this purpose, processors may offer hardware support for running extremely sensitive code in a secure, isolated environment, known as a trusted execution environment in ARM's 'TrustZone' or an enclave in Intel's 'Software Guard Extension' (SGX). They offer slightly different primitives. For instance, the code running in an SGX enclave is intended to be a part of a normal user process. The memory it uses is always encrypted as soon as it leaves the processor. Moreover, SGX offers hardware support to perform attestation, so that a (possibly remote) party can verify that the code is running in an enclave and that it is the right code. ARM TrustZone, on the other hand, isolates the 'normal world' that runs the normal operating system and user applications, from a 'secure world' that typically runs its own (smaller) operating system as a well as a small number of security sensitive applications. Code in the normal world can call code in the secure world in a way that resembles the way applications call into an operating system. One interesting application of special environments such as ARM TrustZone (or Intel's SMM mode, discussed later) is to use it for runtime monitoring of the integrity of a regular operating system—hopefully detecting whatever stealthy malware or rootkit compromised it before it

[6] Again, Intel was actually late to the party, as similar features existed in a variety of processors since the 1960s.

can do some serious damage. Although aspects of these trusted environments clearly overlap with operating system security, we consider them mostly beyond the scope of this knowledge area.

Switching gears again, it may be the case that the operating system is fine, but the hardware is not. Malicious (or faulty) hardware may use the system's direct memory accress (DMA) to read or overwrite sensitive data in memory that should be inacessible to them. Moreover, with some standards (such as Thunderbolt over USB-C), a computer's PCIe links may be directly exposed to devices that a user plugs into a computer. Unfortunately for the user, it is hard to be sure that what looks like, say, a display cable or power adapter, does not also contain some malicious circuitry designed to compromose the computer. As a partial remedy, most architectures nowadays come with a special MMU for data transferred to and from devices. This hardware, called an IOMMU, serves to map device virtual addresses to physical addresses, mimicking exactly the page-based protection illustrated in Fig. 2, but now for DMA devices. In other words, devices may access a virtual memory address, which the IOMMU translates to an actual physical address, checks for permissions and stops if the page is not mapped in for the device, or the protection bits do not match the requested access. While doing so provides some measure of protection against malicious devices (or indeed drivers), it is important to realise that the IOMMU was designed to facilitate virtualisation and really should not be seen as a proper security solution. There are many things that may go wrong. For instance, perhaps the administrator wants to revoke a device's access rights to a memory page. Since updating the IOMMU page tables is a slow operation, it is not uncommon for operating systems to delay this operation and batch it with other operations. The result is that there may be a small window of time during which the device still has access to the memory page even though it appears that these rights have already been revoked.

Finally, we can observe that the increasing number of transistors per surface area enables a CPU vendor to place more and more hardware extensions onto their chips, and the ones discussed above are by no means the only security-related ones in modern processors. Additional examples include cryptograhic units, memory encryption, instructions to switch extended page tables efficiently, and pointer authentication (where the hardware detects modification of pointer values). There is no doubt that more features will emerge in future generations and operating systems will have to adapt in order to use them in a meaningful way. A broader view of these issues is found in the Knowledge Area on *Hardware Security*.

4.6 Protection rings

Among the most revolutionary ideas introduced by Multics was the notion of *protection rings*—a hierarchical layering of privilege where the inner ring (ring 0) is the most privileged and the outer ring is the least privileged [26]. Accordingly, untrusted user processes execute in the outer ring, while the trusted and privileged kernel that interacts directly with the hardware executes in ring 0, and the other rings could be used for more or less privileged system processes.

Protection rings typically assume hardware support, something most general purpose processors offer today, although the number of rings may differ. For instance, the Honeywell 6180 supported as many as eight rings, Intel's x86 four, ARM v7 three and PowerPC two. However, as we shall see, the story becomes slightly confusing, because some modern processors have also introduced more and different processor modes. For now, we simply observe that most regular operating systems use only two rings: one for the operating system and one for the user processes.

Whenever less privileged code needs a function that requires more privileges, it 'calls into' the lower ring to request the execution of this function as a service. Thus, only trusted, privileged code may execute the most sensitive instructions or manipulate the most sensitive data. Unless less privileged process tricks more privileged code into doing something that it should not be doing (as a confused deputy), the rings provide powerful protection. The original idea in Multics was that transitioning between rings would occur via special *call gates* that enforce strict control and mediation. For instance,

the code in the outer ring cannot make a call to just any instruction in the inner ring, but only to predefined entry points where the call is first vetted to see if it and its arguments do not violate any security policy.

While processors such as the x86 still support call gates, few operating systems use them, as they are relatively slow. Instead, user processes transition into the operating system kernel (a 'system call') by executing a software interrupt (a 'trap') which the operating system handles, or more commonly, by means of a special, highly efficient system call instruction (with names such as SYSCALL, SYSENTER, SVC, SCALL etc., depending on the architecture). Many operating systems place the arguments to the system call in a predefined set of registers. Like the call gates, the traps and system call instructions also ensure that the execution continues at a predefined address in the operating system, where the code inspects the arguments and then calls the appropriate system call function.

Besides the user process calling into the operating system, most operating systems also allow the kernel to call into the user process. For instance, UNIX-based systems support *signals* which the operating system uses to notify the user program about 'something interesting': an error, an expired timer, an interrupt, a message from another process etc. If the user process registered a handler for the signal, the operating system will stop the current execution of the process, storing all its processor states on the process' stack in a so-called signal frame, and continue execution at the signal handler. When the signal handler returns, the process executes a `sigreturn` system call that makes the operating system take over, restore the processor state that is on the stack and continue executing the process.

The boundary between security domains, such as the operating system kernel and user space processes is a good place to check both the system calls themselves and their arguments for security violations. For instance, in capability-based operating systems, the kernel will validate the capabilities [40], and in operating systems such as MINIX 3 [16], specific processes are only allowed to make specific calls, so that any attempt to make a call that is not on the pre-approved list is marked as a violation. Likewise, Windows and UNIX-based operating systems have to check the arguments of many system calls. Consider, for instance, the common `read` and `write` system calls, by which a user requests the reading of data from a file or socket into a buffer, or the writing of data from a buffer into a file or socket, respectively. Before doing so, the operating system should check if the memory to write from or read into is actually owned by the process.

After executing the system call, the operating system returns control to the process. Here also, the operating system must take care not to return results that jeopordise the system's security. For instance, if a process uses the `mmap` system call to request the operating system to map more memory into its address space, the operating system should ensure that the memory pages it returns no longer contain sensitive data from another process (e.g., by initialising every byte to zero first [45]).

Zero intialisation problems can be very subtle. For instance, compilers often introduce padding bytes for alignment purposes in data structures. Since these padding bytes are not visible at the programming language level at all, the compiler may see no reason to zero initialise them. However, a security violation occurs when the operating system returns such a data structure in response to a system call and the unitialised padding contains sensitive data from the kernel or another process.

Incidentally, even the signalling subsystem in UNIX systems that we mentioned earlier is an interesting case for security. Recall that the `sigreturn` takes whatever processor state is on the stack and restores that. Now assume that attackers are able to corrupt the stack of the process and store a fake signal frame on the stack. If the attackers are then also able to trigger a `sigreturn`, they can set the entire processor state (with all the register values) in one fell swoop. Doing so provides a powerful primitive in the hands of a skilled attacker and is known as sigreturn-oriented programming (SROP [46]).

4.7 One ring to rule them all. And another. And another.

As also mentioned earlier, the situation regarding the protection rings is slightly more confusing these days, as recent CPUs offer virtualisation instructions for a hypervisor, allowing them to control the hardware accesses at ring 0. To do so, they have added what, at first sight, looks like an extra ring at the bottom. Since on x86 processors, the term 'ring 0' has become synomymous with 'operating system kernel' (and 'ring " with 'user processes'), this new hypervisor ring is commonly referred to as *'ring –1'*. It also indicates that operating systems in their respective virtual machines can keep executing ring 0 instructions natively. However, strictly speaking, it serves a very different purpose from the original rings, and while the name ring 1 has stuck, it is perhaps a bit of a misnomer.

For the sake of completeness, we should mention that things may get even more complex, as some modern processors still have other modes. For instance, x86 offers what is known as *system management mode (SMM)*. When a system boots, the firmware is in control of the hardware and prepares the system for the operating system to take over. However, when SMM is enabled, the firmware regains control when a specific interrupt is sent to the CPU. For instance, the firmware can indicate that it wants to receive an interrupt whenever the power button is pressed. In that case, the regular execution stops, and the firmware takes over. It may, for instance, save the processor state, do whatever it needs to do and then resume the operating system for an orderly shutdown. In a way, SMM is sometimes seen as a level lower than the other rings (*ring 2*).

Finally, Intel even added a *ring 3* in the form of the Intel Management Engine (ME). ME is a completely autonomous system that is now in almost all of Intel's chipsets; it runs a secret and completely independent firmware on a separate microprocessor and is *always* active: during the booting process, while the machine is running, while it is asleep, and even when it is powered off. As long as the computer is *connected* to power, it is possible to communicate with the ME over the network and, say, install updates. While very powerful, its functionality is largely unknown except that it runs its own small operating system.[7] The additional processors that accompany the main CPU (be it the ME or related ones such as Apple's T2 and Google's Titan chips) raise an interesting point: is the operating system running on the main CPU even capable of meeting today's security requirements? At least, the trend appears to augment it with special-purpose systems (hardware and software) for security.

4.8 Low-end devices and the IoT

Many of the features described above are found, one way or another, in most general-purpose processor architectures. However, this is not necessarily true in the IoT, or embedded systems in general, and tailored operating systems are commonly used [47]. Simple microcontrollers typically have no MMUs, and sometimes not even MPUs, protection rings, or any of the advanced features we rely on in common operating systems. the systems are generally small (reducing attack surface) and the applications trusted (and possibly verified). Nevertheless, the embedded nature of the devices makes it hard to check or even test their security and, wherever they play a role in security sensitive activities, security by means of isolation/containment and mediation should be enforced externally, by the environment. Wider IoT issues are addressed in the Knowledge Area on *Cyber-Physical Systems*.

5 Operating System Hardening

The best way to secure operating systems and virtual machines is to have no vulnerabilities at all: security by design. For instance, we can use formal verification to ensure that certain classes of bugs cannot be present in the software or hardware, and that the system is functionally correct [40]. Scaling the verification to very large systems is still challenging, but the field is advancing rapidly and we have now reached the stage that important components (such as a microkernel, file systems

[7] Version 11 of the ME, at the time of writing, is based on MINIX-3.

and compilers) have been verified against a formal specification. Moreover, it is not necessary to verify all the components of a system: you only need a verified microkernel/hypervisor and a few more verified components to be able to guarantee isolation. Verification of other components may be desirable, but is not essential for isolation. Of course, the verification itself is only as good as the underlying specification. If you get that wrong, it does not matter if you have verified it, you may still be vulnerable.

Also, despite our best efforts, however, we have not been able to eradicate all security bugs from large, real-world systems. To guard themselves against the types of attacks described in the threats model, modern operating systems employ a variety of solutions to complement the above isolation and mediation primitives. We distinguish between five different classes of protection: information hiding, control flow restrictions, partitioning, code and data integrity checks, and anomaly detection.

5.1 Information hiding

One of the main lines of defense in most current operating systems consists of hiding whatever the attackers may be interested in. Specifically, by randomising the location of all relevant memory areas (in code, heap, global data and stack), attackers will not know where to divert the control flow, which addresses contain sensitive data etc. The term Address Space Layout Randomisation (ASLR) was coined around the release of the PaX security patch, which implemented this randomisation for the Linux kernel in 2001 [48] — see also the discussion in the Knowledge Area on *Software Security* (Section 4.2). Soon, similar efforts appeared in other operating systems and the first mainstream operating systems to have ASLR enabled by default were OpenBSD in 2003 and Linux in 2005. Windows and MacOS followed in 2007. However, these early implementations only randomised the address space in user programs and randomisation did not reach the kernel of major operating systems, under the name of KASLR (Kernel ASLR), until approximately a decade after it was enabled by default in user programs.

The idea of KASLR is simple, but there are many non-trivial design decisons to make. For instance, how random is random? In particular, what portion of the address do we randomise? Say your Linux kernel has an address range of 1GB (=2^{30}) for the code, and the code should be aligned to 2MB (=2^{21}) boundaries. The number of bits available for randomisation (the *entropy*) is $30 - 21 = 9$ bits. In other words, we need at most 512 guesses to find the kernel code. If attackers find a vulnerability to divert the kernel's control flow to a guessed address from a userspace program and each wrong guess leads to a system crash, it would suffice to have userspace access to a few hundred machines to get it right at least once with high probability (although many machines will crash in the process).

Another important decision is what to randomise. Most implementations today employ coarse-grained randomisation: they randomise the *base* location of the code, heap or stack, but within each of these areas, each element is at a fixed offset from the base. This is simple and very fast. However, once attackers manage to get hold of even a single code pointer via an information leak, they know the addresses for every instruction. The same is true, *mutatis mutandis*, for the heap, stack etc. It is no surprise that these information leaks are highly valued targets for attackers today.

Finer-grained randomisation is also possible. For instance, it is possible to randomise at the page level or the function level. If we shuffle the order of functions in a memory area, even knowing the base of the kernel code is not sufficient for an attacker. Indeed, we can go more fine-grained still, and shuffle basic blocks, instructions (possibly with junk instructions that never execute or have no effect) or even the register allocations. Many fine-grained randomisation techniques come at the cost of space and time overheads, for instance, due to reduced locality and fragmentation.

Besides the code, fine-grained randomisation is also possible for data. For instance, research has shown that heap allocations, globals and even variables on the stack can be scattered around memory. Of course, doing so will incur a cost in terms of performance and memory.

Considering KASLR, and especially coarse-grained KASLR, as our first line of defense against mem-

ory error exploits would not be far off the mark. Unfortunately, it is also a very weak defense. Numerous publications have shown that KASLR can be broken fairly easily, by leaking data and/or code pointers from memory, side channels, etc.

5.2 Control-flow restrictions

An orthogonal line of defense is to regulate the operating system's control flow. By ensuring that attackers cannot divert control to code of their choosing, we make it much harder to exploit memory errors, even if we do not remove them. The best example is known as control-flow integrity (CFI [49]), which is now supported by many compiler toolchains (such as LLVM and Microsoft's Visual Studio) and incorporated in the Windows kernel under the name of Control Flow Guard as of 2017 — see also the Knowledge Area on *Software Security*.

Conceptually, CFI is really simple: we ensure that the control flow in the code always follows the static control flow graph. For instance, a function's return instruction should only be allowed to return to its callsite, and an indirect call using a function pointer in C, or a virtual function in C++, should only be able to target the entry point of the legitimate functions that it should be able to call. To implement this protection, we can label all the legitimate targets for an indirect control transfer instruction (returns, indirect calls and indirect jumps) and add these labels to a set that is specific for this instruction. At runtime, we check whether the control transfer the instruction is about to make is to a target that is in the set. If not, CFI raises an alarm (and/or crashes the program).

Like ASLR, CFI comes in many flavours, from coarse-grained to fine-grained, and from context sensitive to context insensitive. And just like in ASLR, most implementations today employ only the simplest, most coarse-grained protection. Coarse-grained CFI means relaxing the rules a little, in the interest of performance. For instance, rather than restricting a function's return instruction to target-only legitimate call sites that could have called this function, it may target *any* call site. While less secure than fine-grained CFI [50], it still restricts the attackers' wiggle room tremendously, and it is a much faster runtime check.

On modern machines, some forms of CFI are (or will be) even supported by hardware. For instance, Intel Control-flow Enforcement Technology (CET) supports shadow stacks and indirect branch tracking to help enforce the integrity of returns and forward-edge control transfers (in a very coarse-grained way), respectively. Not to be outdone, ARM provides pointer authentication to prevent illegitimate modification of pointer values—essentially by using the upper bits of a pointer to store a pointer authentication code (PAC), which functions like a cryptographic signature on the pointer value (and unless you get the PAC right, your pointer is not valid).

Unfortunately, CFI only helps against attacks that change the control flow—by corrupting control data such as return addresses, function pointers and jump targets—but is powerless against non-control data attacks. For instance, it cannot stop a memory corruption that overwrites the privilege level of the current process and sets it to 'root' (e.g., by setting the effective user id to that of the root user). However, if restrictions on the control flow are such a success in practice, you may wonder if similar restrictions are also possible on data flow. Indeed they are, which is called data flow integrity (DFI [51]). In DFI, we determine statically for each load instruction (i.e., an instruction that reads from memory) which store instructions may legitimately have produced the data, and we label these instructions and save these labels in a set. At runtime we remember, for each byte in memory, the label of the last store to that location. When we encounter a load instruction, we check if the last store to that address is in the set of legitimate stores, and if not, we raise an alarm. Unlike CFI, DFI has not been widely adopted in practice, presumably because of the significant performance overheads.

5.3 Partitioning.

Besides the structural decomposition of a system in different security domains (e.g, into processes and the kernel) protected by isolation primitives with or without hardware support, there are many

additional techniques that operating systems employ to make it harder for attackers to compromise the TCB. In this section, we discuss the most prominent ones.

W⊕X memory. To prevent code injection attacks, whereby the attackers transfer control to a sequence of instructions they have stored in memory areas that are not meant to contain code such as the stack or the heap, operating systems today draw a hard line between code and data [52]. Every page of memory is either executable (code pages) or writable, but not both at the same time. The policy, frequently referred to as W⊕X ('write xor execute'), prevents the execution of instructions in the data area, but also the modification of existing code. In the absence of code injection, attackers interested in diverting the control flow of the program are forced to reuse code that is already present. Similar mechanisms are used to make sensitive data in the kernel (such as the system call table, the interrupt vector table, etc.) read-only after initialisation. All major operating systems support this mechanism, typically relying on hardware support (the NX bit in modern processors[8])—even if the details differ slightly, and the name may vary from operating system to operating system. For instance, Microsoft refers to its implementation by the name Data Execution Prevention (DEP).

Preventing the kernel from accessing userspace. We have already seen that operating systems use the CPU's protection rings to ensure that user processes cannot access arbitrary data or execute code in the operating system, in accordance with the security principles by Saltzer & Schroeder, which prescribe that all such accesses be mediated. However, sometimes we also need to protect the other direction and prevent the kernel from blindly accessing (or worse, executing) things in userspace.

To see why this may be bad, consider an operating system where the kernel is mapped into every process' address space and whenever it executes a system call, it executes the kernel code using the process' page tables. This is how Linux worked from its inception in 1991 until December 2017. The reason is that doing so is efficient, as there is no need to switch page tables when executing a system call, while the kernel can efficiently access all the memory. Also since the kernel pages have the supervisor (*S*) bit set, there is no risk that the user process will access the kernel memory. However, suppose the kernel has a bug that causes it to dereference a function pointer that under specific circumstances happens to be NULL. What will happen? The most likely thing to happen is that the kernel crashes. After all, the kernel is trying to execute code on a page that is not valid. But what if a malicious process deliberately maps a page at address 0, and fills it with code that changes the privileges of the current process to that of root? In that case, the kernel will execute the code, with kernel privileges. This is bad.

It should now be clear that the kernel should probably not blindly execute process code. Nor should it read blindly from user data. After all, an attacker could use it to feed malicious data to the kernel instructions. To prevent such accesses, we need even more isolation than that provided by the default rings. For this reason, many CPUs today provide Supervisor Mode Execution Protection (SMEP) and Supervisor Mode Access Protection (SMAP)[9]. SMEP and SMAP are enabled by setting the appropriate bits in a control register. As soon as they are on, any attempt to access or transfer control to user memory will result in a page fault. Of course, this also means that SMAP should be turned off explicitly whenever the kernel *needs* to access user memory.

Some operating systems, including Linux, got SMEP-like restrictions 'for free' on systems vulnerable to the Meltdown vulnerability in 2017 [8], which forced them to adopt an alternative design (which was definitely not for free!). In particular, they were forced to abandon the single address space (where the kernel executes in the address space of the process), because of the Meltdown out-of-order execution side channel from Table 1. To recap, the Meltdown (and related Spectre) attacks consist of attackers abusing the CPU's (over-)optimism about what happens in the instructions it executes out-of-order or speculatively. For instance, it (wrongfully) assumes that load instructions have the

[8] NX (no execute) is how AMD originally called the feature in its x86 compatible CPUs. Intel calls it XD (execute disable) and ARM XN (execute never).

[9] Again, this is x86 terminology. On ARM similar features are called PAN (Privileged Access Never) and PXN (Privileged Execute Never).

privilege to read the data they access, the outcome of a branch is the same as the previous time a branch at a similar address was executed, or the data needed for a load instruction is probably the data in this temporary CPU buffer that was just written. However, even if any of these assumptions are wrong, the CPU can recover by squashing the results of the code that was executed out-of-order or speculatively.

In a Meltdown-like attack, the attackers' process executes an out-of-order instruction to read a byte at a (supposedly inaccessible) kernel address, and the CPU optimistically assumes all is well and simply accesses the byte. Before the CPU realises things are *not* well after all and this byte should not be accessible, the attackers have already used the byte to read a particular element in a large array in their own process' address space . Although the CPU will eventually squash all the results, the damage is already done: even though the byte cannot be read directly, the index of the array element that is in the cache (and is, therefore, measurably faster to access than the other elements) must be the kernel byte.

To remedy this problem on somewhat older processors (that do not have a hardware fix for this vulnerability), operating systems such as Linux use a design that completely separates the page tables of the kernel from those of the processes. In other words, the kernel also runs in its own address space, and any attempt by an out-of-order instruction to read a kernel address will fail. The kernel can still map in the pages of the user process and thus access them if needed, but the permissions can be different. Specifically, if they are mapped in as non-executable, we basically get SMEP functionality for free.

For other vulnerabilities based on speculative execution (such as the Spectre vulnerability), the fix is more problematic. Often, multiple different spot solutions are used to patch the most serious issues. For instance, after a bounds check that could be influenced by untrusted users, we may want to insert special instructions to stop speculation completely. However, it is unclear how complete the set of patches will be, until the hardware itself is fixed.

Partitioning microarchitectural states Sophisticated side channel attacks build on the aggressive resource sharing in modern computer systems. Multiple security domains share the same cache, the same TLB, the same branch predictor state, the same arithmetic units etc. Sharing is good for efficiency, but, as indicated by the Principle of Least Common Mechanism, they also give rise to side channels. To prevent such attacks, operating systems may need to sacrifice some of the efficiency and partition resources even at fine granularity. For instance, by means of *page colouring* in software or hardware-based cache allocation technology, an operating system may give different processes access to wholly disjointed portions of the cache (e.g., separating the cache sets or separating the ways within a cache set). Unfortunately, partitioning is not always straightforward and currently not supported for many low-level resources.

5.4 Code and data integrity checks

One way to reduce the exploitability of code in an operating system, is to ensure that the code and/or data is unmodified and provided by a trusted vendor. For instance, for many years already, Windows has embraced *driver signing*. Some newer versions have taken this a step further and use a combination of hardware and software security features to lock a machine down, ensuring that it runs only trusted code/apps—a process referred to by Microsoft as 'Device Guard'. Even privileged malware cannot easily get non-authorized apps to run, as the machinery to check whether to allow an app to run sits in a hardware-assisted virtualised environment. Most code signing solutions associate digital signatures associated with the operating system extensions allow the operating system to check whether the code's integrity is intact and the vendor is legitimate. A similar process is popularly used for updates.

However, what about the code that checks the signature and, indeed, the operating system itself—are we sure that this has not been tampered with by a malicious bootkit? To verify the integrity of the

booting process, the unified extensible firmware interface offers the possibility to enable Secure Boot. It verifies whether the boot loaders were signed with the appropriate key, i.e., using keys that agree with the key information in the firmware. It will prevent loaders and drivers without the appropriate signatures from gaining control of the system. Securely booting devices start with an initial 'root of trust' code, which initiates the booting process and is typically based in hardware: a microcontroller that starts executing software from internal, immutable memory, or from internal flash memory that cannot be reprogrammed at all, or only with strict authentication and authorisation checks. As an example, modern Apple computers use a separate prcocessor, the T2 Security Chip, to provide the hardware root of trust for secure boot (among other things), while Google has also developed a custom processor (called the Titan) for this.

A related topic is that of attestation, whereby a (remote) party can detect any changes that have been made to our system. It typically uses special hardware that serves as root of trust and consists of verifying, in steps, whether the system was loaded with the 'right' kind of software.

Code and data integrity may also happen at runtime. For instance, the hypervisor may provide functionality to perform introspection of its virtual machines: is the code still the same, do the data structures still make sense? This techique is known as virtual machine introspection (VMI). The VMI functionality may reside in the hypervisor itself, although it could be in a separate application. Besides the code, common things to check in VMI solutions include the process list (is any rootkit trying to hide?), the system call table (is anybody hijacking specific system calls?), the interrupt vector table, etc.

5.5 Anomaly detection

A monitor, be it in the hypervisor or in the operating system, can also be used to monitor the system for unusual events—anomaly detection [53]. For instance, a system that crashes hundreds of times in a row could be under attack by someone who is trying to break the system's address space layout randomisation. Of course, there is no hard evidence and just because an anomaly occurred does not mean there is an attack. Anomaly detection systems must strike a balance between raising too many false alarms (which are costly to process) and raising too few (which means it missed an actual attack).

6 Operating Systems, Hypervisors—what about Dababases?

Security in database systems follows similar principles as those in operating systems with authentication, privileges, access control and so on as prime concerns. The same is true for access control, where many databases offer discretionary access control by default, and role-based and mandatory access control for stricter control to more sensitive data. Representing each user as a security domain, the questions we need to answer concern, for instance, the user's privileges, the operations that should be logged for auditing, and the resource limits such as disk quota, CPU processing time, etc. A user's privileges consist of the right to connect to the database, create tables, insert rows in tables, or retrieve information from other users' tables, and so on. Note that sometimes users who do not have access to a database except by means of a specific SQL query may craft malicious inputs to elevate their privileges in so-called SQL injection attacks [54].

While database-level access control limits who gets access to which elements of a database, it does not prevent accesses at the operating system level to the data on disk. For this reason, many databases support transparent data encryption of sensitive table columns on disk—often storing the encryption keys in a module outside the database. In an extreme case, the data in the database may be encrypted while only the clients hold the keys.

Querying such encrypted data is not trivial [55]. While sophisticated cryptographic solutions (such as homomorphic encryption) exists, they are quite expensive and simpler solutions are commonly used. For instance, sometimes it is sufficient to store (also) the hash of a credit card number, say, instead

of the actual number and then query the database for the hash. Of course, in that case, only exact matches are possible—as we cannot query to see if the value in the database is greater than, smaller than, or similar to some other value (nor are aggregated values such as averages or sums possible). The problem of querying encrypted databases is an active field of research and beyond the scope of this knowledge area.

While security and access control in regular databases is non-trivial already, things get even more complex in the case of Outsourced Databases (ODBs), where organisations outsource their data management to external service providers [56]. Specifically, the data owner creates and updates the data at an external database provider, which then deals with the clients's queries. In addition to our earlier concerns about confidentiality and encryption, questions that arise concern the amount of trust to place in the provider. Can the data owner or the querying client trust the provider to provide data that was created by the original data owner (authenticity), unmodified (integrity), and fresh results to the queries? Conceptually, it is possible to guarantee integrity and authenticity by means of signatures. For instance, the data owner may sign entire tables, rows/records in a table, or even individual attributes in a row, depending on the desired granularity and overhead. More advanced solutions based on authenticated data structures are also commonly advocated, such as Merkle hash trees. In Merkle hash trees, originally used to distribute authenticated public keys, leaf nodes in the tree contain a hash of their data value (the database record), each non-leaf node contains a hash of the hashes of its children, and the root node's hash is signed and published. All that is needed to verify if a value in a leaf node is indeed part of the orignal signed hash tree is the hashes of the intermediate nodes, which the client can quickly verify with a number of hashes proportional to the logarithm of the size of the tree. Of course, range queries and aggregation are more involved and researchers have proposed much more complex schemes than Merkle hash trees, but these are beyond the scope of this knowledge area. The take-away message is that with some effort we can guarantee authenticity, integrity and freshness, even in ODBs.

7 Embracing Security

Increasingly advanced attacks are leading to increasingly advanced defenses. Interestingly, many of these innovations in security do not originally come from the operating system vendors or large open source kernel teams, but rather 'from the outside'—sometimes academic researchers, but in the case of operating system security, also often from independent groups such as *GRSecurity* and *the PaX Team*. For instance, the PaX Team introduced ASLR as early as 2001, played a pioneering role in making data areas non-executable and executable sections non-writable, as well as in ensuring the kernel cannot access/execute user memory. Surprisingly, where you might think that the major operating systems would embrace these innovations enthusiastically, the opposite is often true and security measures are adopted inconsistently.

The main reason is that nothing is free and a slow-down or increase in power consumption because of a security measure is not very popular. The Linux kernel developers in particular have been accused of being obsessed with performance and having too little regard for security. However, when the situation is sufficiently pressing, there is no other way than to deal with the problem, even if it is costly. In operating systems, this performance versus security tradeoff has become increasingly important. Research often focuses on methods that significantly raise the bar for attackers, at an 'acceptable' overhead.

CONCLUSION

Operating System Security has focussed to a large extent on isolation of users, processes, domains, or classes of these things, from each other. Threats of increasing sophistication have prompted improved design of operating system software, and increasingly of hardware support for the operating system's primitives. Not least among the former are hypervisors and virtual machines — and many

of the same principles are applied in the design of database management systems.

CROSS-REFERENCE OF TOPICS VS REFERENCE MATERIAL

	kopka2003 [**?**]	gratzer2016 [**?**]	...	...
?? ??				
?? ??	c4			
?? ??	c11,c12			
?? ??	c7	II.5		
...				

REFERENCES

[1] J. H. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. DOI: 10.1109/PROC.1975.9939, pp. 1278–1308, September 1975.

[2] G. C. Hunt and J. R. Larus, "Singularity: Rethinking the software stack," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 37–49, Apr. 2007. [Online]. Available: http://doi.acm.org/10.1145/1243418.1243424

[3] S. Boyd-Wickizer and N. Zeldovich, "Tolerating malicious device drivers in linux," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855849

[4] B. Grill, A. Bacs, C. Platzer, and H. Bos, "Nice Boots - A Large-Scale Analysis of Bootkits and New Ways to Stop Them," in *DIMVA*, Sep. 2015. [Online]. Available: http://www.cs.vu.nl/%7Eherbertb/papers/bootkits_dimva2015.pdf

[5] V. van der Veen, N. Dutt-Sharma, L. Cavallaro, and H. Bos, "Memory Errors: The Past, the Present, and the Future," in *RAID*, Oct. 2012. [Online]. Available: http://www.few.vu.nl/~herbertb/papers/memerrors_raid12.pdf

[6] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 361–372. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665726

[7] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. Berkeley, CA, USA: USENIX Association, 2017, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=3241189.3241191

[8] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. Berkeley, CA, USA: USENIX Association, 2018, pp. 973–990. [Online]. Available: http://dl.acm.org/citation.cfm?id=3277203.3277276

[9] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[10] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 605–622.

[11] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security*, Aug. 2018. [Online]. Available: https://www.vusec.net/download/?t=papers/tlbleed_sec18.pdf

[12] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *S&P*, May 2016. [Online]. Available: https://www.vusec.net/download/?t=papers/dedup-est-machina_sp16.pdf

[13] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza, "Attack surface metrics and automated compile-time os kernel tailoring." in *NDSS*. The Internet Society, 2013. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2013.html#KurmusTDHRRSLK13

[14] D. M. Ritchie and K. Thompson, "The unix time-sharing system," *Commun. ACM*, vol. 17, no. 7, pp. 365–375, Jul. 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361061

[15] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for unix development," 1986, pp. 93–112.

[16] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Minix 3: A highly reliable, self-repairing operating system," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 3, pp. 80–89, Jul. 2006. [Online]. Available: http://doi.acm.org/10.1145/1151374.1151391

[17] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 251–266. [Online]. Available: http://doi.acm.org/10.1145/224056.224076

[18] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The design and implementation of an operating system to support distributed multimedia applications," *IEEE J.Sel. A. Commun.*, vol. 14, no. 7, pp. 1280–1297, Sep. 96. [Online]. Available: http://dx.doi.org/10.1109/49.536480

[19] A. S. Tanenbaum, *Operating Systems: Design and Implementation*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987.

[20] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 29–44. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629579

[21] Version 7 Unix, "chroot," Modern chroot man page http://man7.org/linux/man-pages/man2/chroot.2.html, 1979.

[22] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[23] D. E. Bell and L. J. LaPadula, “Secure computer systems: Mathematical foundations,” MITRE CORP BEDFORD MA, Tech. Rep., November 1973.

[24] K. J. Biba, “Integrity considerations for secure computer systems,” MITRE CORP BEDFORD MA, Tech. Rep., April 1977.

[25] D. Ferraiolo and D. Kuhn, “Role-based access controls,” in *Proceedings of the 15th Annual National Computer Security Conference*. NSA/NIST, 1992, pp. 554–563.

[26] J. H. Saltzer, “Protection and the control of information sharing in multics,” *Commun. ACM*, vol. 17, no. 7, pp. 388–402, Jul. 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361067

[27] U. S. D. of Defense, *Department of Defense, Trusted Computer System Evaluation Criteria*, ser. The ’Orange Book’ series. Dept. of Defense, 1985. [Online]. Available: https://books.google.nl/books?id=-KBPAAAAMAAJ

[28] S. Smalley, C. Vance, and W. Salamon, “Implementing selinux as a linux security module,” *NAI Labs Report*, vol. 1, no. 43, p. 139, 2001.

[29] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, “The flask security architecture: System support for diverse security policies,” in *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, ser. SSYM’99. Berkeley, CA, USA: USENIX Association, 1999, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251421.1251432

[30] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris, “Labels and event processes in the asbestos operating system,” in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 17–30.

[31] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, “Making information flow explicit in histar,” in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 263–278.

[32] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, “Information flow control for standard os abstractions,” in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 321–334.

[33] J. B. Dennis and E. C. Van Horn, “Programming semantics for multiprogrammed computations,” *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[34] H. M. Levy, *Capability-based computer systems*. Digital Press, 1984.

[35] S. J. Mullender and A. S. Tanenbaum, “The design of a capability-based distributed operating system,” *The Computer Journal*, vol. 29, no. 4, pp. 289–299, 1986.

[36] W. B. Ackerman and W. W. Plummer, “An implementation of a multiprocessing computer system,” in *Proceedings of the First ACM Symposium on Operating System Principles*, ser. SOSP ’67. New York, NY, USA: ACM, 1967, pp. 5.1–5.10. [Online]. Available: http://doi.acm.org/10.1145/800001.811666

[37] R. Fabry, “A user’s view of capabilities,” ICR Quarterly Report. U. of Chicago Institute for Computer Research, pp. pages C1–C8, November 1867.

[38] R. M. Needham and R. D. Walker, “The cambridge cap computer and its protection system,” in *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, ser. SOSP ’77. New York, NY, USA: ACM, 1977, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/800214.806541

[39] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "Hydra: The kernel of a multiprocessor operating system," *Commun. ACM*, vol. 17, no. 6, pp. 337–345, Jun. 1974. [Online]. Available: http://doi.acm.org/10.1145/355616.364017

[40] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 207–220. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629596

[41] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for unix," in *Proceedings of the 19th USENIX Conference on Security*, ser. USENIX Security'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929820.1929824

[42] F. Güntsch, "Logischer entwurf eines digitalen rechnergerätes mit mehreren asynchron laufenden trommeln und automatischem schnellspeicherbetrieb Logical Design of a Digital Computer with Multiple Asynchronous Rotating Drums and Automatic High Speed Memory Operation," Ph.D. dissertation, Technische Universität Berlin, 1957.

[43] T. Killburn, "One-level storage system," *IRE Transactions on Electronic Computers*, vol. EC-ll, no. 2, April 1962.

[44] R. C. Daley and J. B. Dennis, "Virtual memory, processes, and sharing in multics," *Commun. ACM*, vol. 11, no. 5, pp. 306–312, May 1968. [Online]. Available: http://doi.acm.org/10.1145/363095.363139

[45] A. Milburn, H. Bos, and C. Giuffrida, "SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities," in *NDSS*, Feb. 2017. [Online]. Available: https://www.vusec.net/download/?t=papers/safeinit_ndss17.pdf

[46] E. Bosman and H. Bos, "Framing signals-a return to portable shellcode," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 243–258.

[47] E. Baccelli, O. Hahm, M. Gux0308nes, M. Wax0308hlisch, and T. C. Schmidt, "Riot os: Towards an os for the internet of things," *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 79–80, 2013.

[48] P. Team, "Address space layout randomization," https://pax.grsecurity.net/docs/aslr.txt (Patch originally released in 2001), 2001.

[49] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: ACM, 2005, pp. 340–353. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102165

[50] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out Of Control: Overcoming Control-Flow Integrity," in *S&P*, Dec. 2014. [Online]. Available: http://www.cs.vu.nl/%7Eherbertb/papers/outofcontrol_sp14.pdf

[51] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 147–160. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298470

[52] P. Team, "Design & implementation of PAGEEXEC," 2000.

[53] E. Bauman, G. Ayoade, and Z. Lin, “A survey on hypervisor-based monitoring: Approaches, applications, and evolutions,” *ACM Comput. Surv.*, vol. 48, no. 1, pp. 10:1–10:33, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2775111

[54] J. F. (signing as rain.forest.puppy), “Nt web technology vulnerabilities,” *Phrack Magazine*, vol. 8, no. 4, December 1998.

[55] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, “Cryptdb: Protecting confidentiality with encrypted query processing,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP ’11. New York, NY, USA: ACM, 2011, pp. 85–100. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043566

[56] E. Mykletun, M. Narasimha, and G. Tsudik, “Authentication and integrity in outsourced databases,” *Trans. Storage*, vol. 2, no. 2, pp. 107–138, May 2006. [Online]. Available: http://doi.acm.org/10.1145/1149976.1149977