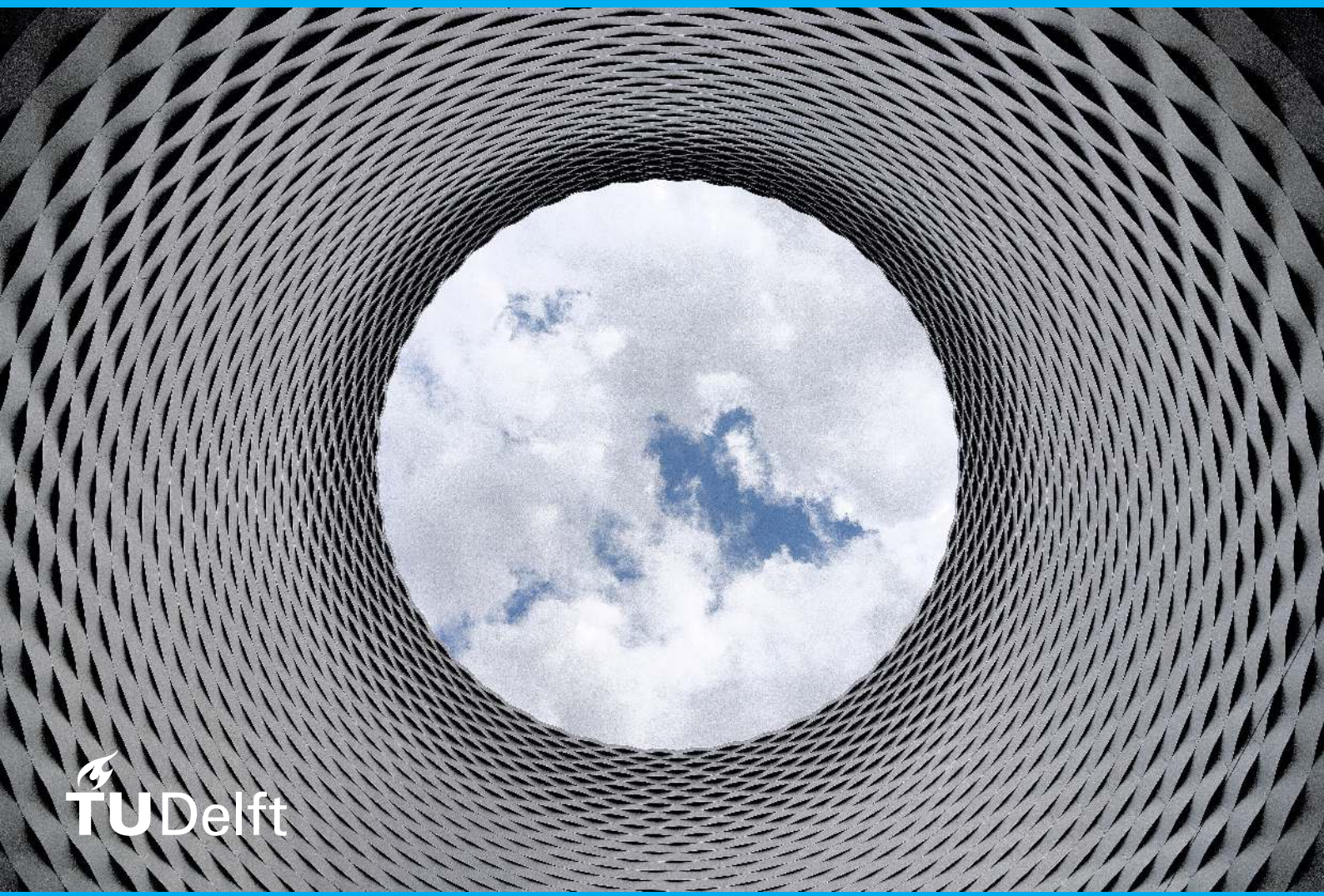
Kernel isolation of a Capability-based security Operating System

M. in ’t Hout



TU Delft

Kernel isolation of a Capability-based security Operating System

by

M. in 't Hout

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday October 29, 2018 at 10:00 AM.

Student number:	4309464	
Project duration:	November 20, 2018 – October 29, 2018	
Thesis committee:	dr. ir. S. Wong,	TU Delft, supervisor
	dr. ir. A. van Genderen,	TU Delft
	Prof. dr. ir. K. Langendoen,	TU Delft
	ir. T. van Leeuwen,	Technoltuion
	ir. J. Hofman,	Technoltuion

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Contents

1 Introduction

Computers have taken an important role in society; we use them to provide for our livelihood, critical infrastructure runs on embedded systems and we share our most important and intimate secrets using computers. This requires a large amount of trust in such systems, trust in aspects like reliability and integrity. The reality today is that many systems are eventually compromised. Widely used encryption software is compromised and security vulnerabilities are found in hardware. If computers and embedded systems are such important systems, is it right to put a huge amount of trust in them?

It is not necessarily bad when a trusted governing body has a lot of power. This means that when this body is inherently good, it can do a lot of good and prevent bad actions from its subjects. However, when this body becomes corrupt, either through own mistakes or an external factor, this power becomes a liability. The same applies to a governing body within an Operating System (OS). This body is called the kernel. Because the kernel has a lot of influence in a system, malicious parties often try to infiltrate the kernel. In common operating systems, when a kernel is corrupted and/or infiltrated, attackers have full control over all memory. Most efforts are to protect the kernel from external influences, but perhaps we should strive to strip the powers of the kernel to the bare minimum. In that way, when there is a hostile take-over, the information in the system could still be secure.

The Open Source OS Framework Genode has already started at an attempt to make the kernels influence smaller [24]. The main goal of this Operating System is to make the required trust as small as possible. Genode does this by dividing the Operating System into small components. After initialisation, the kernel only serves the purposes of messenger between components and as a scheduler of tasks. Aside from the kernel, components only need to trust other components that are truly necessary for that component. This compartmentalising and the small kernel are the reasons that we will be working with Genode in the course of this thesis when investigating how the power of the kernel can be reduced even more.

1.1. Problem definition

As mentioned above, most endeavours have been put into increasing the security instead of decreasing the impact of a breach in the outer wall. In the event of such a breach, the kernel is one of the critical systems that is exposed. This leads us to the following question:

Can we reduce the amount of privileges of the kernel in a performance efficient manner?

To answer this question we introduce three sub-questions which we will subsequently deal with.

- *"What vectors are there to attack a kernel?"* Attackers have a long-standing tradition of bypassing most defences that have been proposed. In order to know which privileges are instrumental in attacks, we should look at the state of attacks and defences that are currently known.
- *"How can we reduce the number of privileges of the kernel?"* Reducing privileges of a kernel seems an idealistic goal. We should investigate whether there is a viable option to accomplish this goal.
- *"What is a performance efficient manner to implement such a restriction?"* Most defences have an overhead in terms of computation time, memory bandwidth, area or power. That is why we should analyse the found defence, if such is found in the second research question, and map current bottlenecks.

1.2. Methodology

To investigate the aforementioned set of questions, a number of actions are established. The sub-questions were introduced in order to reach a conclusion regarding the main question. This section lays out the actions needed to answer the sub-questions.

- *"What vectors are there to attack a kernel?"* This question will be approached with a literature study into the different attack vectors that exist on software. These vectors will be grouped in terms of method, the desired end result of the attack and defence category.
- *"How can we reduce the number of privileges of the kernel?"* In order to provide this question with an answer, a defence category will be selected from the literature study. Subsequently, we will develop and implement a defence within this category, which will also be tailored to fit the existing infrastructure.
- *"What is a performance efficient manner to implement such a restriction?"* To answer this question, we will review the proposed system found and developed in the previous actions. This review will focus on performance bottlenecks and suggest improvements on the developed defence. These improvements will be verified through a combination of simulation and actual implementation.

1.3. Thesis outline

The rest of the paper is structured as follows. First, in Chapter 2, the background of this thesis is laid out. This comprises of the existing infrastructure in which this thesis takes place and existing attacks and defences. Next, in Chapter 3 a defence is proposed. In Chapter 3.5 the proposed system will be reviewed and improved upon. Thereafter, in chapter **??** the results will be presented. Finally, this thesis will be concluded in Chapter 5.

2
Background

Before answering the first sub-question by providing an overview of possible attacks, defences and goals, we will sketch the foundation of security in OS and focus on the OS used in this research. As mentioned, the reason for choosing Genode as an OS was that the developers have already created a smaller kernel than usual. Further reasons will also be mentioned in the coming section.

2.1. Foundations of Security in Operating Systems

2.1.1. Origin

One of the first OS that needed some form of security can be attributed the Compatible Time-Sharing System (CTSS) that was created by researchers at Massachusetts Institute of Technology (MIT) [14]. This system enabled the usage of one machine by multiple users and therefore needed separation between users. MIT continued this project with the Multics system, which is a heavy influence in today's operating systems. Multics was designed to be more secure than other OS at that time. Security rings, virtual memory, Access Control Lists (ACL) and other protection features that can still be found today were implemented in Multics [47]. Researchers of MIT also introduced security concepts such as *capabilities* [20], a concept of unforgeable tokens on which Section 2.1.3 will expand.

As stated above, security features of Multics can be found in current OS, for example, UNIX. This is not surprising as UNIX is a descendant of Multics. But unlike Multics, UNIX is not designed with security in mind. In a secure system, all security policies must still work when everything outside of the Trusted Code Base (TCB) is malicious. UNIX does not adhere to this specification. There are attempts to increase the security of UNIX, for example with the module Security-Enhanced Linux (SELinux).

One aspect of security design that this chapter also will show, is that whenever someone makes a secure system, finding flaws in those systems is as important as the design itself. As finding flaws enables better designs. This also can be seen in an analysis of the Multics system [34]. This analysis of flaws in Multics sparked new security measures in both software and hardware aspects. The old kernel of Multics was too large to be able to verify its correctness. That is why they adopted a simplified and smaller kernel [48].

2.1.2. Microkernels

As shown with the Multics system, the more simple and small a kernel is, the more a kernel can be better understood and is less likely to have bugs. That is the underlying philosophy with microkernels. As the name suggests, a microkernel is a smaller than average kernel. Microkernels move every non-critical traditional OS service, such as the file system, outside of the kernel.

Also, Tanenbaum [56] underlines that while monolithic kernels are typically faster, microkernels are more reliable. The reasoning for this is that the code size of a kernel such as Linux is 4000 times larger compared to a microkernel. This increases the number of bugs and exploitable objects significantly. Secondly, with such a large code size there is nobody that can comprehend such a

large system, which makes bug fixing more difficult. Third, a modern OS does not have isolation between components in the kernel. This makes that bugs (and possible malicious code) can have more impact.

The RC 4000 Multiprogramming System is the first notable attempt to create such a microkernel, although the reasoning behind this system was more on flexibility than on reliability or security [28]. For the same reason, the OS Hydra was built, but it did feature a capability-based protection [39].

The Mach microkernel is also a notable and a much-researched microkernel. The microkernel was built to be a replacement of the monolithic BSD OS. Mach is in terms of security features not much different from UNIX, except that the TCB has become much smaller with the Mach kernel. Mach also inspired a range of derivatives of Mach, the most notable derivatives for our research are so-called *security kernels*.

Security kernels are kernels with specific focus on security. The goal of these kernels is to achieve a level of integrity and implement a security policy with assurances. The assurances are provided by a combination of compile-time and run-time verification. The reference monitor is an important component of a security kernel. The reference monitor performs run-time verification of the enforcement of the security policy. As with the microkernels, in order to properly verify the TCB, the kernel and other components in the TCB needs to be as simple and small as possible.

Research in Microkernels stagnated around 2000s because of the slow performance compared to monolithic kernels. That is why some research focused on the bottlenecks of microkernels. Liedtke, for example, found that a large bottleneck is Inter Process Communication (IPC) and created the L4 microkernel to mitigate some of these bottlenecks. However, monolithic kernels such as Linux or Windows are currently more popular.

2.1.3. Capabilities

A *capability* is an unforgeable pair made up of an object identifier and a set of authorised operations on that object [20]. The object identifier points to a single unique object inside the system. The referred object can be virtual such as a file or array but also physical such as memory or a debugging port. Each process holds a set of capabilities and that set dictates the access to other processes in the system. Such a set of capabilities that a process has is called a *protection domain*.

Capabilities provide a mechanism that is suitable to make small components instead of large monolithic programs. This is because capabilities provide protection per protection domain instead of per user. This has the advantage that similar to microkernels, small components can be better understood and therefore contain fewer faults.

As shown before, capabilities are not a new concept and are embedded in the very first generations of OS's. However, in common OS security based on Access Control Lists are prevalent. Drawbacks in capability-based systems can explain the reasoning for choosing for Access Control Lists. One of the advantages of a capability-based system is the fine-grained segmentation and isolation of programs. However, there is overhead in switching and transferring information between protection domains. More segments equal more overhead.

There is still active research in capability-based systems. CHERI is a current research project from the University of Cambridge. They revisit hardware-based capability systems and combine them with a RISC Instruction Set Architecture (ISA). They focus on creating a low overhead system for fine-grained memory and pointer protection and also on reducing the overhead of compartmentalisation.

2.1.4. Genode

The Genode OS Framework strives to create a modular OS in which the kernel is reduced to only the truly necessary. Because of the combination of a microkernel with capability-based security, this results in an OS that is intrinsic more secure than current modern OS (e.g. Linux or Windows).

The most recent generation of microkernels (OLK4, SeL4) stems from the L4 microkernel family. Genode is a framework that builds on those microkernels.

Genode implements the mentioned capability-based security with so-called *capability lists*. These are lists of capabilities that are maintained inside the kernel. The components themselves get a capability reference from the kernel that is context dependent for a specific component. This fulfils the unforgeable requirement of capabilities, as only the kernel has the ability to create or grant capabilities. When a component does not have a certain capability, the component cannot create a valid reference on its own.

That the kernel keeps all capabilities has an impact on sharing capabilities, as those need to be shared with the help of the kernel. Components can not simply pass capabilities without the knowledge of the kernel. When a component has a capability, it also gets the right to delegate this capability to other processes.

As the kernel keeps all capabilities and passes all communication, the kernel has a key role in the integrity of the OS. Should the kernel be overtaken or maliciously tweaked, the whole security will be void.

2.2. Attacks and defences

It is important to know what types of attacks there exist in order to observe common pitfalls in the security of operating systems. This research will focus on attacks on low-level programming languages such as C, C++ and assembly. Those programming languages are commonly used in operating systems, Linux is mostly programmed in C, Windows is programmed in a combination of C, C++ and C#. For interfacing with the underlying hardware, some assembly is also required.

Another reason for the focus on low-level languages is that for the purposes of this literature study, Genode is also composed of low-level languages (C++, assembly). The goal of this study is to map different attack vectors on operating systems and listing possible defences.

2.2.1. Attack methods

As stated previously, the focus is on attacks on low-level programming languages such as C, C++ and assembly. According to [23] there are four types of attacks on low-level languages that are representative of all attacks on low-level languages. Those are stack-based buffer overflows, heap-based buffer overflows, jump-to-libc attacks, and data-only attacks. They discard "format-string attacks" as a viable option because they are simple to eliminate. However, [12] estimates that there are over 1,000 format string vulnerabilities in the packages of the Linux distribution Debian which makes the problem more widespread than initially thought. The last type that was dismissed was integer overflow, which exists, but as an attack which leads to the other four types. Slightly ambiguous is the distinction between the buffer overflows and the jump-to-libc / Return-Oriented Programming (ROP) attack as an overflow exploit or other pointer corruption exploit is needed to perform such an attack.

Figure 2.1 provides an overview of the types of attacks and the flow between attacks and goals. In this figure, green blocks represent bugs or entry points, orange blocks are needed permissions and blue blocks represent end goals or exploits. Attacks start with a form of user input which exploits a bug in the system. Integer overflow, string formatting, stack and heap overflow are all dependent on bugs in a system. [29] estimates that there are 3-6 bugs per thousand lines of code of well-written software. This means that bugs are likely to exist in most software. Exploits also have an average lifetime of 6.9 years ([4]) which can make the reward for finding an exploit great. After the initial exploit, several methods can be applied to achieve an attackers goal. These exploits and methods will be explained in the subsequent sections.

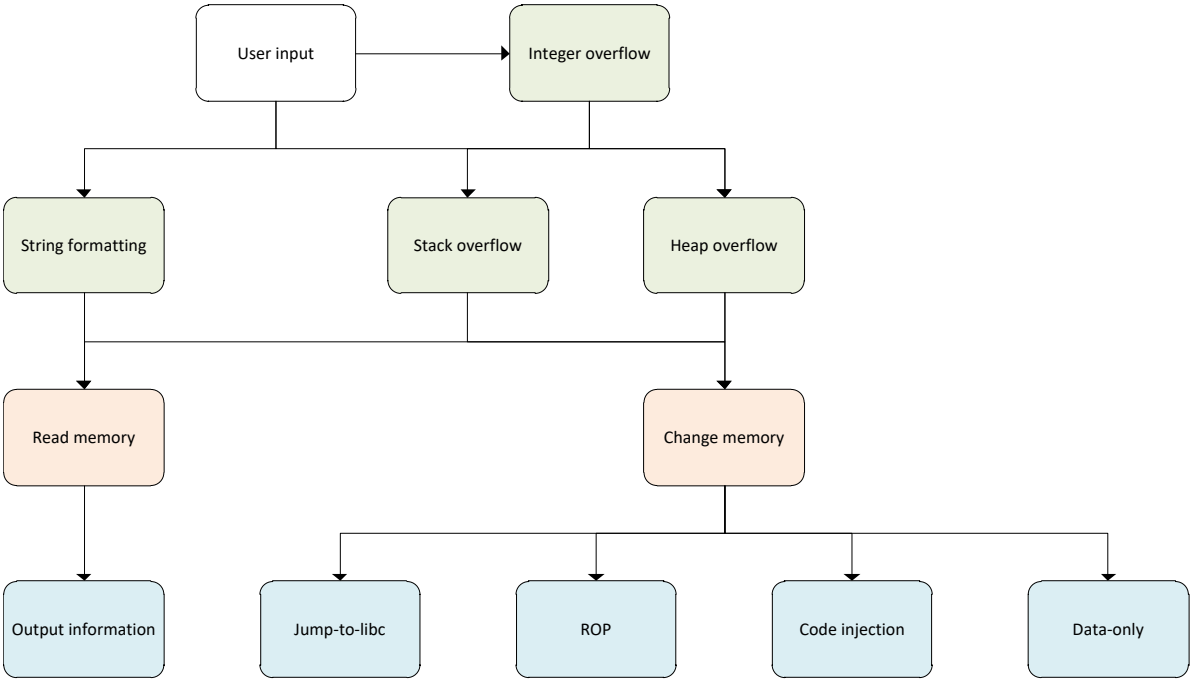


Figure 2.1: Attack structure on low-level languages

Buffer overflow

A buffer overflow happens when a program writes to an address on the current stack or heap outside the intended range. A buffer overflow usually happens by writing to an unchecked buffer. Exploits can leverage this to write to the stack and to divert the current control flow. Examples of things that can be overwritten is a local variable or a return address. Figure 2.2 illustrates a simple example. The figure on the left would be the memory map in normal operation, while on the right the memory map after a successful buffer overflow. The attacker has not only written to the buffer region but has also overwritten the return address of the current function. This enables the attacker to redirect the current control flow. If the attacker sets the return address to the address of the buffer, the attacker can inject and run arbitrary code.

Both Stack and Heap overflows are types of buffer overflows, the difference is the location of the overflow region. Because the stack and heap are used differently, the exploitation of the overflow is also different. The example described above is a stack overflow because the return address is located on the stack. The information that is stored on the heap are data structures, these structures can combine data buffers and function pointers. When a buffer in the heap overflows, the subsequent function pointer can be overwritten.

Buffer errors vulnerabilities in software is a large problem. According to the US National Vulnerability Database (NVD), buffer errors are the most found type of vulnerability [2].

Return-to-libc

As it is possible to overwrite addresses, a way to exploit a buffer overflow is to divert the control flow to existing functions. The most used target for this is libc, which is a code library that among other things contains system calls. By chaining such calls, it is possible to further exploit the system. An example of a convenient function in libc for attackers is *strcpy()* which copies a string to another location. That way you could copy the buffer to another location where code can run and execute it. The first documented exploit of a return-to-libc attack is attributed to [21]. The attack *return-to-libc* was born because simple overflow exploits are reasonably easy to defend against. Such a simple defence is to prevent the execution of code from the stack. Attackers still can overwrite addresses

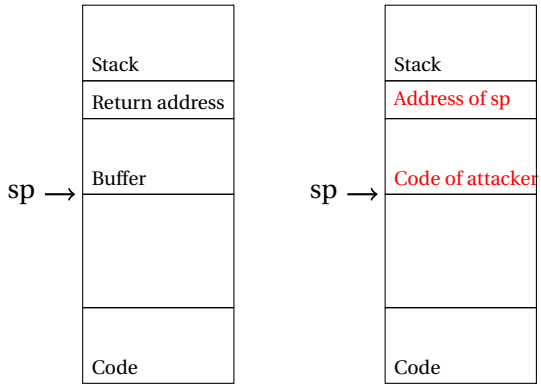


Figure 2.2: Buffer overflow attack

trough the original buffer overflow but the defence decreases the usefulness of the exploit as the attacker can not directly inject code.

ROP

ROP generalises the Return-to-libc attack by using small excerpts from existing programs and not needing libc or any other default library. Instead of using whole functions, it tries to find small pieces of code that end in a return instruction. The return instruction makes it possible to chain the small pieces of code. The small excerpts are called *gadgets* and if there are enough of them it is possible that the set is Turing complete. This makes it possible for an attacker to execute arbitrary code. The first occurrence of ROP was introduced by [50] on the x86 architecture. This attack soon expanded to other architectures [9, 25]. While most exploits depend on the *return* instruction, return-oriented programming also is possible without such instructions [11].

Integer overflow

An integer overflow exists after a computation of which the end result is outside the representable range of the result type. When this is not properly handled, this can lead to unintended behaviour. An example of such an attack is CVE-2002-0639 [15] which uses an integer overflow as a step to create a buffer overflow and therefore can write to memory. The authors of [22] remark that intentional integer overflows are more common than previously believed. This makes detecting unintentional overflows harder.

Non-Control Data Attack

Unlike most attacks that focus on subverting the control flow, non-Control-Data Attacks changes memory that is not directly related to the control flow. The authors of [13] show that such attacks are realistic and most defences are useless against such attacks. They present examples of attacks that, while not diverting control flow, can achieve higher privileges trough existing exploits.

2.2.2. Attacks on kernels

As stated before, the kernel has one of the most important roles in an OS. The kernel is hence also a wanted target in cyber attacks. Attacks that focuses on kernels to subvert the control or data flow of the kernel. An attack on a kernel that replaces some of the control flow of the kernel is called a *rootkit*. Because rootkits enable the full control of the system, the goals of rootkits are wide-ranging. The goals include privilege escalation (e.g. backdoors) and collecting sensitive information such as documents or encryption keys.

Next to rootkits, there are also exploitable bugs or design flaws in kernels that do not subvert the normal control flow. These usually revolve around privilege escalation. An example of such an

exploit is called 'dirty cow' which uses a race condition found inside the Linux kernel to achieve write access to otherwise read-only memory[1].

The confused deputy problem is also an example of the usage such a design flaw. It is often explained with an example of a program that provides compilation as a service. The clients provide names of input and output files, and the server compiles the input and writes to the output. The server stores billing information in a file named BILL, as the service is pay-per-use. When a client supplies 'BILL' as output file name, the server overwrites the billing information. This occurs even when the clients have no access to the file BILL, this takes advantage of the permissions that the server has. The problem lies that the server uses its own permissions to open the files of clients.

The attacks mentioned in section 2.2.1 are all leveraged in taking control of the kernel. [19] describes a kernel attack in the Android OS. This attack uses a heap overflow present in Android as an initial attack and exploits the bug with a ROP attack.

2.2.3. Available defences

Now that we have an understanding of the types of attacks that can be initiated, a defence must be found to combat these attacks. The research field of computer defences is large, even with the specific focus on defences that apply to low-level programming languages.

An overview of the steps an attacker makes and the defences thereof can be seen in Figure 2.3. Each container represents an defence and each inner box represents a step of an attack. As can be seen, there are a lot of defences available, each with its own advantages and caveats. We will expand on the different categories of defences in the following sections.

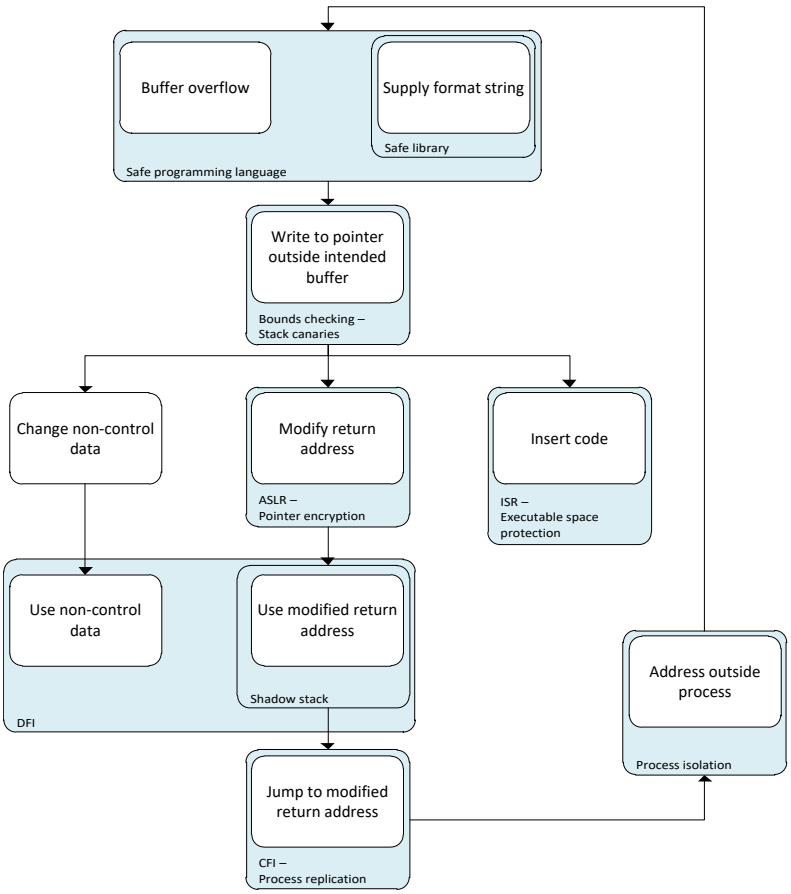


Figure 2.3: Attack steps and defence structure on low-level languages

Address Space Layout Randomization (ASLR)

ASLR The authors of [57] strife to make the finding of functions and gadgets needed by ROP improbable by randomising the address space of processes. Each time a program is loaded, various sections will be placed somewhere different trough the use of virtual memory. The pages of virtual memory are usually 4Kb sized, and therefore a maximum of 20bit that can be used for ASLR. The authors of [51] show that ASLR is ineffective on 32 bit systems and can be beaten in an average of 216 seconds. There are also more fine-grained types of randomisation that not only randomises the memory address, but also the relative distance between objects and therefore making it harder to find objects and functions [6]. But also 64 bit systems and more fine-grained ASLR are not safe [53]. Attackers can recover gadgets dynamically and on-the-fly; Attackers are therefore not bound to the static locations of gadgets or functions.

Stack canaries

Stack canaries are values that are written on the stack and are validated before any jump in the code. If such a value is overwritten an exception is raised and is handled by the OS. This is a method to detect an overflow and prevent that the address in the stack is followed. If an attacker can read the canary value this method is broken. Stackguard[16] is an example of such a defence. Aside from the stack, canaries are also used to protect other locations such as the heap [46].

Instruction Set Randomization

The randomisation or encryption of the used instruction set is called Instruction Set Randomization (ISR). This defence prevents attackers from inserting valid code in the memory. [35] uses a simple XOR with a key to encrypt instructions. While cheap on hardware, XOR provides no diffusion. This means that specific bit can be targeted, such as the opcode, which makes retrieving the key more feasible [55]. More secure encryption processes can be used such as AES, but also increases the overhead. A variant that is similar to ISR is to encrypt pointers [18]. This should prevent the attacker to select a viable address to jump to. Because pointers are widely used, the overhead is still large but can be less then ISR. These schemes also have the drawback that it depends on a secret value (key) that may leak at some point. While ISR only prevents code injection attacks, encrypting pointers can also prevent ROP and return-to-libc attacks.

Executable space protection

Executable space protection aims to prevent the execution of code from the stack. Write XOR Execute (W⊕X) is an implementation thereof [58]. It divides memory pages into either executable data or writeable data but not both. But because code first needs to be loaded into those pages, this option can be turned off at any time by the kernel. This is a simple protection that has made its way into almost every architecture (Intel, AMD, RISC-V, ARM, Transmeta, VIA, SPARC). While it can provide a protection against the code injection attack, it does not provide any protection against a Return-to-libc or ROP attack. This is because a ROP attack uses existing code to execute the payload.

Control Flow Integrity

Control Flow Integrity (CFI) makes sure that the control flow of the programs that are running are within a valid path. This method is first suggested by [3]. The idea is that a compromised program can be detected because it results in an abnormal control flow. CFI usually has two components, one component analyses the program on valid execution traces and composes a Control Flow Graph (CFG). This analysis can be static, dynamic or a combination thereof. The other component inspects that the current control flow is valid. While an ideal implementation of CFI would stop all known ROP attacks, creating such an ideal CFI is infeasible. This is because creating a complete and precise CFG is difficult and when having a complete CFG there is a performance overhead that can not be ignored. When strafing from the ideal implementation towards the more practical CFI defences, the security that such a practical method delivers is less and some ROP attacks are still feasible. [26]

Data Flow Integrity

Similarly to CFI that focuses on the control flow, Data Flow Integrity (DFI) focuses on the data flow of a program. This is a valid mitigation technique because buffer overflows and ROP depends on data being changed by the attacker. This method can also defend against Non-Control data attacks e.g. attacks that do not divert from the normal control flow. A basic form of DFI is also called dynamic taint analysis. Dynamic taint analysis taints incoming input and prevents the usage of input in control-flow instructions. [10] was the first to fully implement DFI, the overhead of their method in terms of executing time is 2.5 times.

Library replacements

As mentioned in Section 2.2.1, string format vulnerabilities occur with certain functions that use a format string. A defence for such vulnerabilities would be to add certain checks to prevent the misuse of such functions. Some methods change the semantics in order to make a distinction between safe and unsafe data [42]. Functions can then require that the format strings must be safe data. This gives the programmer tools to make string vulnerabilities less likely. The downside of this approach is that the code needs to be rewritten or converted because of the change in semantics. Also, bugs are less likely but will depend on the skill of the programmer and therefore the bugs can still persist. Other methods keep the default semantic but make the exploitation harder. [17] counts the arguments of the function at compile-time, which makes it harder because most exploits depend on supplying more format specifiers than arguments. FormatShield [38] works by identifying exploitable function calls and restricting those calls by forbidding format specifiers.

Language-based security

Safe programming languages impose security by providing memory, type and thread safety. This should make a buffer overflow exploit impossible as programs are guaranteed to only access memory locations that they are permitted. Language-based security builds on proven constructs which prevent overflows and other bugs. Cyclone [32] is an example of such a safe programming language. Cyclone combines static analysis during compile-time and run-time checks to ensure memory bounds. They estimate that about 10% of the code of common C programs needs to be changed to conform to cyclone. The average performance overhead is 1.4 [62]. CCured is also a so-called safe dialect of C. Unlike Cyclone, CCured is backwards compatible with C. They use annotations and type interference to separate pointers into several safety levels. The average performance overhead is 4.7 [62]. While language-based security is a good solution to combat the mentioned vulnerabilities, it is not always feasible to write all code of applications, dependencies and OS in a safe language [30]. If only the user programs are written in a safe language, the used libraries and the OS are still unsafe.[45]

Process isolation

Most OS's apply some sort of process isolation. This defence aims to prevent processes from reading and interfering with each other. The defence is usually implemented through hardware mechanisms that are supported by the Memory Management Unit (MMU). These mechanisms are memory segmentation, page mapping, and differentiated user and kernel instructions. Virtual addresses, aside from their practical use, is such a mechanism. Because switching between address spaces incur a non-negligible overhead, fine-grained isolation can lead to a too expensive protection [31]. The authors of [30] combine traditional process isolation and Language-based security to reduce the overhead of address switching. They created a safe programming language, added verification tools to aid the programmer to create safe programs and created an OS to make the traditional mechanisms more flexible.

Bounds checking

Bound checkers protect against buffer overflows by checking all pointers to ensure that they do not write or read outside their allotted memory. This method uses the information about the bounds of pointers during compile time and stores this metadata. This can either happen by appending pointers with metadata, so-called fat-pointers, or by creating a table of the metadata and perform a look-up during the usage of the pointer. Some methods use a table to store object information in a separate table and infer the bounds information from that table. The first method to implement such a defence is [37]. This is later refined by [33] by storing object information in a separate table and thereby creating a more inter-operability with non-safe libraries. Because of the slowdown for [37] and [33] is around 5-6 times, this defence is also implemented in hardware by [52]. By implementing a write-only checking policy and compiler optimizations they achieve a slowdown of around 5%. These defences are applicable to the stack and heap overflows because these attacks rely on pointers that go out of their intended bounds.

Information replication

Information replication is focused on replicating the information that is vital for the control flow. A basic example is to copy all data related to control-flow to a different memory location (preferable outside user-accessible memory). When control data is used, the copies are compared, when they are different an unauthorised change is made. This is called a shadow stack [59]. Some methods make changes during the compilation [59], others make changes in hardware to reduce the overhead in terms of execution time [60]. Replication can also be applied to processes. This works by introducing a diversity between two (or more) duplicates and comparing the output of the duplicates [8]. While the former can only prevent stack overflows, process replication can provide a defence to all mentioned attacks. The defence of process replication does depend on the amount and type of diversification.

2.2.4. Defences on kernels

Like the attacks, most defences can also be applied to kernels. The authors of [44] for example argues that most rootkits subvert the control flow and therefore a form of CFI needs to be implemented. This proposed state-based CFI periodically verifies if the kernel is in a valid state, instead of real-time CFI, to decrease the performance strain.

Non-control data attacks must be defended according to [61]. They first determine that the ability to change these non-control data can lead to serious security problems. To combat this they opted for an *information replication* method by running multiple virtual machines and cross-checking anomalous changes.

As data from user-space must be considered unsafe data, hardware techniques are developed to disable execution from user-space, which is called Supervisor Mode Execution Prevention (SMEP). The variant in which access is prevented is called Supervisor Mode Access Prevention (SMAP). Also, most modern kernels employ some form of kernel ASLR. The authors of [27] show that those systems can be subverted and proposes a stronger kernel isolation to combat side channel attacks.

2.3. Discussion and Conclusion

In order to choose a defence to be implemented, we need to take a look at the privileges it can restrict and if it is viable to implement such a defence in the current context. *Safe libraries* mostly defend against format strings, and therefore, will not amount to an effective defence. *Safe programming* is promising, but costly in terms of time needed to convert an OS to a safe language. *Stack canaries*, *ASLR*, *ISR*, *pointer encryption* all implement a probabilistic defence and therefore not limit any privileges. *Process replication* depends on diversification, a microkernel is per definition small in terms of code, therefore also the diversification will be small. *Bounds checking, shadow stacks, CFI, DFI,*

Process isolation and *executable space protection* all limit the kernel in some sense and are viable to implement. Our focus will continue on *Process isolation,* as that is most compatible with the extensive compartmentalising that Genode does.

When looking ahead to the next sub-question *"How can we reduce the number of privileges of the kernel?"* we can conclude that one the fitting category defences for this question is *process isolation*. This is because the defence limits the freedom of processes and the kernel.

3
Kernel isolation

3.1. Privileges of a kernel

A kernel without privileges seems paradoxical. The reason for a kernel to exist is to have a section of software that has access to the inner most resources and from there can orchestrate all user processes and resources. So when you strip a kernel of privileges, what would remain of the kernel? And moreover, what privileges can be stripped from the kernel, while it still has its tasks to perform? To answer these questions we first need to know the inner workings of a kernel, what privileges it has and what privileges it really needs to perform its tasks. Only when these questions are answered, the problem of stripping the kernel to a bare minimum can be dealt with. So what does a kernel do?

A kernel consists of everything except user applications in a monolithic based OS. Examples of responsibilities that fall under such a monolithic kernel are File Systems, Device Drivers and Virtual Memory Management. So this monolithic kernel generally carries most responsibilities. Besides the monolithic kernel, there is also a version of the kernel that is called the microkernel. This is a stripped version of the kernel, in the sense that a microkernel-based in an OS is smaller and performs responsibilities by separating processes in user-space. It is therefore sensible to describe the microkernel and its responsibilities further when looking for the basic privileges a kernel might need.

The three responsibilities that these kernels have, are Initialisation, IPC and Scheduling. Dealing with all three of these tasks would be preferable, but would also make it too extensive for this research. Consequently, we had to select a responsibility of the kernel to focus our efforts on when minimising the privileges of a kernel. In making this selection we kept in mind that memory access is imperative for all attacks, as discussed in Chapter 2. That already narrows the choice down to Initialisation and IPC. Because Initialisation is only performed once during start-up and IPC has a critical role in a microkernel, we decided to put our focus on the IPC. Therefore we will leave a detailed description of Initialisation and Scheduling behind and expand on the purpose of IPC in the microkernel.

3.1.1. IPC

IPC is a fundamental mechanism in the contemporary OS. It is used for a wide range of functions. These functions include, but are not limited to, exchanging data between processes, synchronisation, access to timers and exceptions. The more an OS is split into components, the more data needs to be exchanged through IPC. This is a reason why monolithic kernels are generally faster as explained in Chapter 2. It also means that a lot of effort needs to go to the performance of IPC when creating a microkernel.

Liedtke investigated the bottlenecks of a second-generation microkernel and confirms this. He came to the conclusion that the most important points when designing a microkernel is IPC performance [40] and platform dependent optimisations [41]. Improving IPC speed can be done by preventing superfluous copy actions and minimising memory usage, for example by using register

to pass small amounts of data. The platform dependent optimisations mostly focus on reducing cache misses and TLB misses.

3.2. Implementation specific details

In order to understand the upcoming design for memory protection that will be presented further in this chapter, the details of the implementation of IPC and other important details of Genode will be discussed.

3.2.1. Tasks

The task structure is a key aspect of Genode. All tasks can create other tasks, but the relationship between these tasks is a parent-child relation. This means two things; first, the parent's resources must be shared with all children. This includes during start-up but also during the whole lifetime of the child. Second, the parent has the ability to stop its children. When a parent creates a child, it will allocate a part of its own memory to the child. This means effectively that no memory will be used without explicit allocation.

3.2.2. Capabilities

As mentioned in Section 2.1.4 one of the fundamental architecture design elements of Genode is security based on capabilities. In Genode, a capability is an unambiguous reference to a Remote Procedure call (RPC) object. Calling a RPC object enables the communication between components.

Each component of Genode has its own *protection domain*. Each capability has significance in only one protection domain. As Section 2.1.4 explained, capabilities are kept by the kernel in capability lists. The kernel 'translates' the capabilities or references between protection domains. The usage of capability lists inside the kernel also gives the kernel the task of creating and deleting the capabilities.

Capabilities can be delegated to other protection domains. A capability thus can be seen as a right to access the RPC object. This delegation of a capability can also only be performed by the kernel. Figure 3.1 [24] shows the relation between capabilities, kernel, protection domains and the RPC object. The circle represents a capability in hands of a protection domain. As can be seen, the kernel translates the capability and maps it to an object identity.

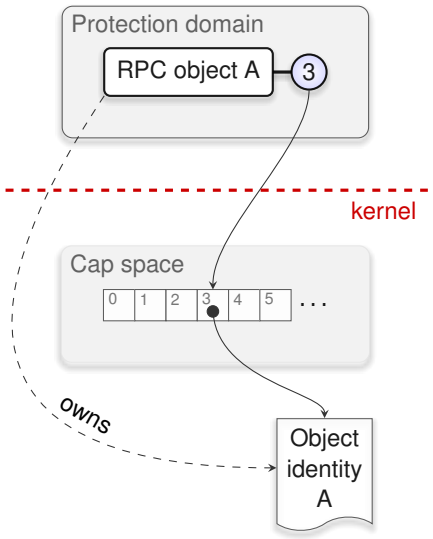


Figure 3.1: Relation between capabilities and the kernel

3.2.3. UTCB

The communication between the processes is done with User-Level Thread-Control Blocks (UTCB). A UTCB is a small (preferably the size of one virtual page) memory region which is always mapped. All information pertaining to the IPC is stored inside this block. The structure of a UTCB can be seen in Figure 3.2. It is the task of the kernel to translate the given capabilities to the side of the receiver.

Until now in this chapter, we have taken a more detailed look into the (micro)kernel, the IPC amongst its other responsibilities and the implementation of the IPC within Genode. Now that some background knowledge is sketched about the inner workings of the IPC within Genode, we can focus on the main goal of this chapter and this thesis; our design for the memory protection. In the next section our design for the kernel separation will be discussed.

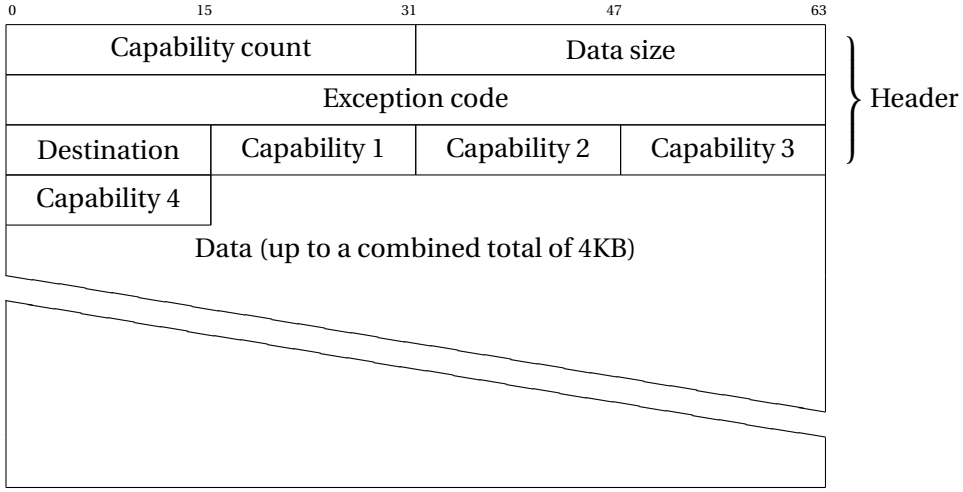


Figure 3.2: Structure of a UTCB

3.3. Memory Isolation

Preventing access to parts of memory is critical in creating a safe system. The goal of most attacks on the kernel is to execute arbitrary code in the privileged mode of the kernel, as stated in Chapter 2. To examine how we can achieve a large degree of memory isolation while keeping reasonable performance we will design a kernel isolation mechanism.

The goal of the design is to prevent the supervisor mode a.k.a. the kernel from accessing and executing user memory. The design is twofold; the first section separates the memory that the kernel uses from the memory that the user uses. The purpose of the second part is to make sure that the kernel does not need user memory access (hardware IPC). The two parts will be discussed and explained separately because the approach to the designs is different.

3.3.1. Memory Tagging

To separate user and kernel memory and making sure the kernel cannot read or execute from user memory is the first objective of the design. We will presume that the kernel can function with only system calls as a means of communication. This means that all RAM memory only is accessed by either a user process or a kernel process and not both.

Some current kernel and user memory separation are Intel's SMAP/SMEP, ARM's PXN and the RISCV PUM (Protect user Memory) bit. These protections all work essentially the same and focus on virtual memory pages that are marked as user or kernel. A current flaw in this design is that kernel pages can be mapped to the same physical address. An example of this is the ret2dir attacks [36] which takes advantage of a direct mapping of the user space inside the kernel. This suggests that a

more rigid approach is needed in order to prevent such attacks.

Another separation technique is called *Memory Tagging* (or tainting/colouring), which works as follows:

- With every n bits of memory, a tag of m bits is kept.
- Every load and or store this tag is adjusted if necessary.
- The tag cannot be viewed by software, only checked.

Memory tagging is a versatile technique and can be used to implement a wide range of defence schemes. CFI, DFI and process isolation can all be implemented with memory tagging.

Memory tagging can be implemented in software by a so-called 'shadow memory', this is a scheme used for example by [49]. With shadow memory, load and store instructions are augmented by extra load instructions that load a separate part of memory which contains the tags. This scheme, however, is not fast. The author of [49] talks about an average slowdown of 75%. This is in contrast to hardware, an example of such a scheme is implemented by [5]. They saw an average slowdown of 8.4%. While these implementations cannot be directly compared, the contrast of performance results is large enough to justify doing memory tagging in hardware.

3.3.2. Implementation details

One challenge that arises is in order to track physical memory to virtual entities (user and kernel processes), some relation must exist. Another challenge is to handle the creation and destruction of virtual memory and to translate that to physical memory with accompanying tags.

The physical difference between memory accesses of the kernel and that of an user process is the privilege level in which the CPU is currently in. The used ISA RISC-V CPU supports 3 privilege levels, User, Supervisor and Machine. The bootloader runs in Machine mode, the kernel runs in supervisor mode and all other code runs in User mode.

Tagging mechanisms

Virtual memory must correspond with the related tags in physical memory, even during the allocation and destruction of memory blocks. This relation must be kept, even when the system is compromised. This can be achieved by implementing the mechanism in a self-regulating manner in hardware.

The mechanism of creating tags is as follows, when accessing a memory location the request is combined with the privilege level. The tag corresponding with the memory location is retrieved. When the tag is empty or uninitialised, the privilege level is inserted into the tag. This tag will persist until the memory location is cleared.

Clearing memory locations can be detected through a string of incoming memory accesses. When a memory location is being written with zero's, the tag will be set to uninitialised. This has an added advantage that no undiscarded data can be read by both parties. Checking tags can be performed when the tag is retrieved, by comparing this with the privilege level.

Overhead

As Genode allocates memory with a granularity of 4KB, we will store a tag for every 4KB. The size of the tag depends on the number of states a tag can have. As we want to make a split between kernel and user space, there are at least two states. In the beginning, the memory is uninitialised and has no state. To be able to allocate memory dynamically, there is a third state added, uninitialised. This means we will keep a tag of 2 bits per every 4KB, this gives a small storage overhead of 1/2048 (0.0005%).

We will split the main memory into a section for tags and a section for common usage. An alternative is to use a separate memory, but this will decrease the flexibility of the tagging mechanism.

3.3.3. Memory tagging architecture

The proposed memory tagging architecture can be seen in Figure 3.5. The implementation is centred around the Memory Management Unit. The orange blocks are hardware entities that are added to the existing grey hardware entities. The MMU is responsible for checking the tag and sending a trap to the processor if necessary.

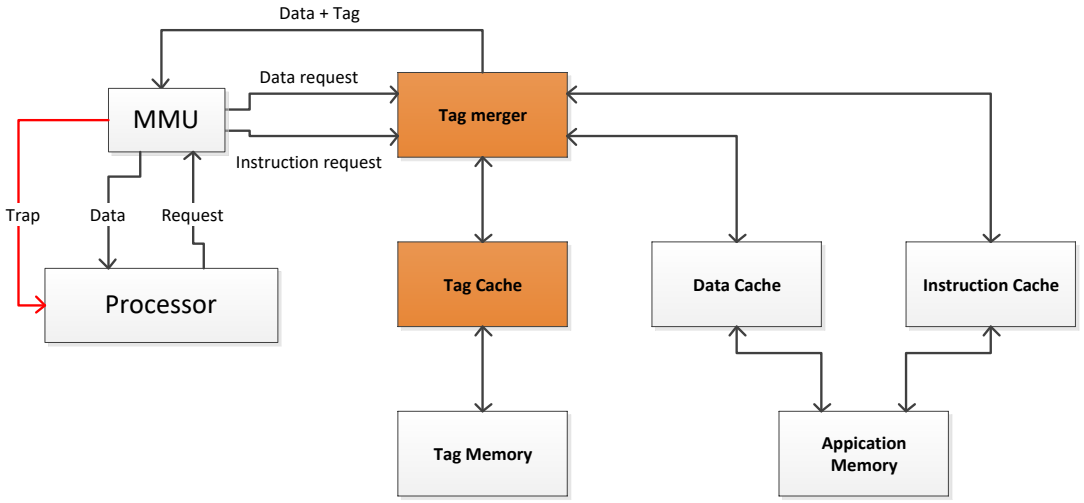


Figure 3.3: Proposed memory tagging architecture

3.4. Hardware IPC

Many operating systems that have a focus on security provide a way for processes to communicate in a secure way. TyTAN [7] for example uses a separate user process that translates and copies IPC between processes.

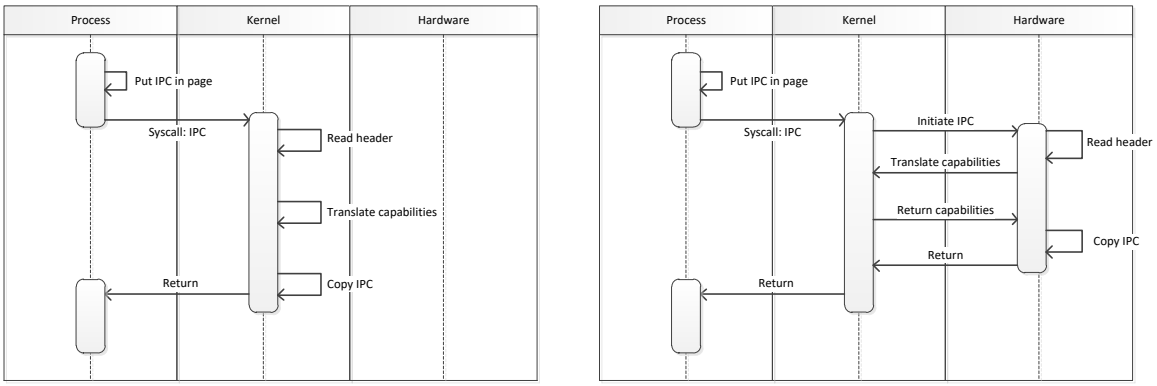
The kernel of Genode needs, like most L4 microkernels, a low amount of access to user memory. With Genode this is limited to one page that is used for IPC. They have chosen this to keep the cost of an IPC low as one page fits in one Translation Look-aside Buffer (TLB) entry. For IPC Genode translates the capabilities on that memory page and copies it to the corresponding user process. This process can be seen in Figure 3.4a.

3.4.1. Hardware IPC architecture

In order to remove user memory access, the IPC needs to happen on another communication channel. There are at least two solutions that work, one is a user process that is responsible for copying the data of the IPC. This would result in an extra context switch and would not be compatible with the current structure of Genode. The second option is to create an hardware based IPC channel. This creates the advantage that the IPC happens within a controlled but static environment.

The new algorithm that is implemented can be seen in Figure 3.4b. As seen, the translation of capabilities still happens in the Kernel, but the kernel can only read the corresponding capabilities of an UTCB.

The total proposed architecture can be seen in Figure 3.5. The implementation is centred around the Memory Management Unit. The green blocks are hardware entities that are added to the existing gray hardware entities. The IPC is implemented as a io device which is only addressable by the kernel.



(a) Old IPC algorithm (b) New IPC algorithm

Figure 3.4: Comparison between the old and new IPC algorithms

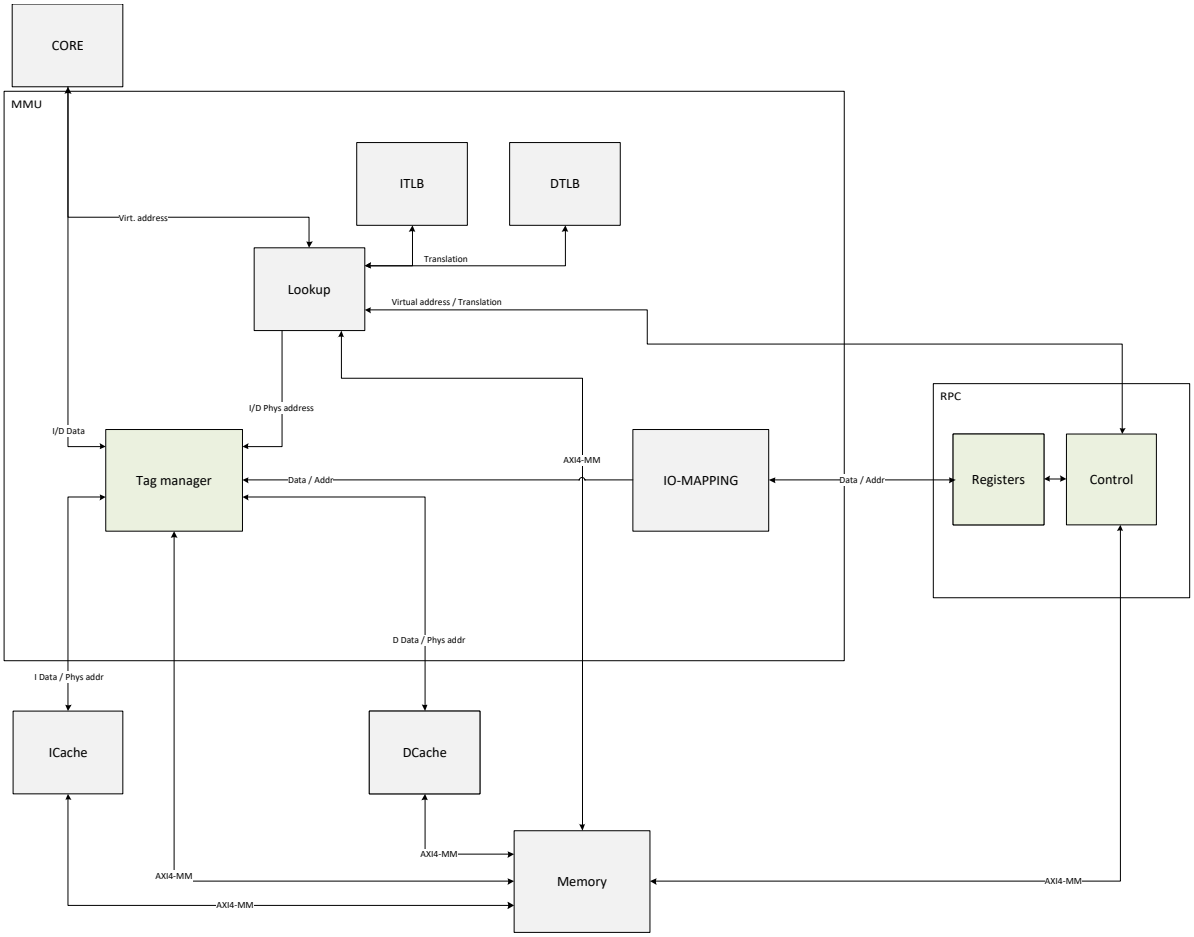


Figure 3.5: Proposed architecture

3.5. Performance

As mentioned before, one of Liedtke's first design rules for his new microkernel was 'IPC performance is the Master'. This is why we dedicated this last section to the increase in performance of the IPC. There are several ways to increase the performance of an OS. This chapter will mostly focus on several caches such as the TLB. The sections will each describe a possible change to the design and describe how this can affect the performance. We will propose improvements and the results will be discussed in the following chapter.

3.5.1. TLB

The Translation Look-aside Buffer (TLB) is a cache that buffers part of the page-table entries. This reduces the workload of the MMU considerably. An average miss rate of a TLB is around the 0.01 – 1% [43]. When looking at section 4, 76.3% of the static time in an IPC is when translating addresses.

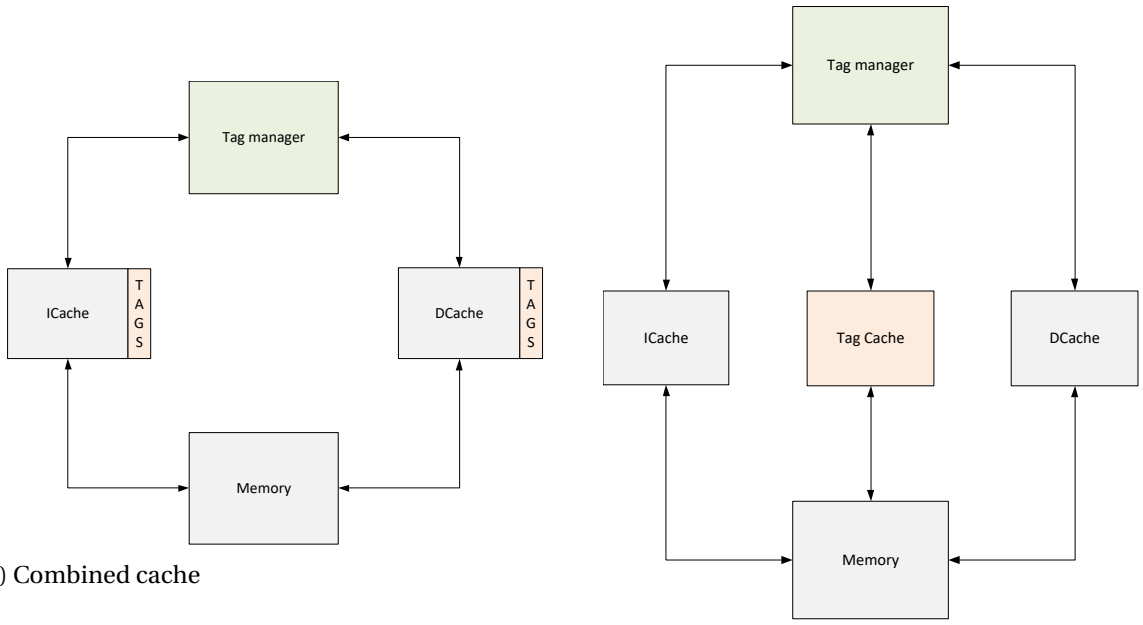
After most IPC, a context switch happens, and the new processes can properly revive the communication. When a context switch happens, the page table is changed and all TLB caches are flushed. This is important for choosing between a separate TLB or combining the entries with an existing TLB. As the entries of an IPC are mostly made just before a context switch, it makes sense to combine the new TLB with an existing TLB.

3.5.2. Tag cache

As instruction and data information are buffered inside a cache, the tags can also be cached. The current architecture has a cache of one level. The tag cache can be added in two ways.

- A separate cache that buffers the tags independently from the other caches (Figure 3.6a).
- A combined cache that appends the tags to the data and instruction caches (Figure 3.6b).

Many proposed memory tagging system employ one of these structures [5, 63], sometimes even both [54]. A memory tagging system without a tag cache will most likely be slow. This is because for all memory requests the tagging system will create an extra separate memory request.



(a) Combined cache

(b) Separate cache

Figure 3.6: Architecture of combined and separate tag caches

3.6. Conclusion

We found an answer on the sub-question *"How can we reduce the number of privileges of the kernel?"*. We showed that it is possible to give the Kernel fewer privileges through a combination of memory tagging and changes to the kernel. If it is needed to increase performance, caching will be a suitable option.

4
Results

In this chapter, the results of the design described in Chapter 3 are presented. Section 4.2 will describe the results of the hardware IPC that is made. Next, Section 4.3 will outline the results of the memory tagging implementation. Finally, Section 4.4 will present the results of implementing a cache to the memory tagging design.

4.1. Experimental set-up

The focus of all measurements that were done is on performance in time. Another aspect that will be measured is the area of the design. Other aspects, such as power consumption have not been considered.

In order to verify the correctness and perform timing measurements tests included in Genode are being used, more specific, the *test_log* and *test_init* are used. The accompanying processor is a classic RISC 5 stage pipeline processor based on the RISC-V ISA. The processor does not support stage forwarding and has a Harvard architecture that splits instruction and data memory.

The experiment is done on a Cyclone V GX Starter Kit, this development board has a Cyclone V GX 5CGXFC5C6F27C7N FPGA and contains 4Gb of LPDDR2 RAM. The design is written in VHDL and synthesised with Intel Quartus Prime 17.1.

The TLB and Tag Cache will be simulated with a cache model written in Python. All memory requests are captured from the original system and replicated in Python. Also, to determine a maximum bound on the improvement that can be reached with caches, a fully mapped cache is implemented in the FPGA.

4.2. Hardware IPC

The processor and all peripherals run on a frequency of *50 Mhz*. The valid working of the IPC (and more aspects of Genode) can be verified through the supplied tests of the OS. The execution times of the old and new IPC can be seen in Table 4.1. The average speedup is 1.03, a small increase. When looking at the average time of an IPC (see Table 4.2), the decrease is almost fourfold.

Test	old IPC	new IPC
test_log	3.17s	3.08s
test_init	2.4s	2.32s

Table 4.1: Execution time comparison

In order to see where the speedup comes from, we will split the execution time of the new IPC into the actions that are performed. Also, the time of an IPC is influenced by the amount of data that needs to be copied, thus we will also group the execution times by data copied. Table 4.3 shows the number of cycles needed to perform the sub-actions. Remarkable is the number of cycles needed for translating the source and destination address, this sums to a large amount of time (a total of

Test	old IPC	new IPC
test_log	21.4μs	5.01μs
test_init	19.9μs	4.93μs

Table 4.2: IPC time comparison

76.3%). Figure 4.1 shows some insight into the time it takes to copy memory. The software copy is most likely faster because of the data cache that is included in the system. There is also a limited amount of coping data in the *test_log* and *test_init* programs.

Next to execution times, area is also an important metric for physical designs. The fabric of the FPGA consists of adaptive logic modules (ALM's) which is a 8-input fracturable look-up table (LUT) with two adders and two registers. An ALM can implement a 6-input LUT but also two 4-input LUT's and other combinations. The area of the IPC module consist of 870 ALM's, this is 4.9% of the total 17956 ALM's that are in use.

Action	cycles	percentage
Write to registers	2	0.8
Translate source address	87	37.6
Translate destination address	89	38.5
Read first header line	18	7.0
Read second header line	18	7.0
Read register	3	1.3
Write to registers	2	0.8
Write headers	9	3.9
Read register	3	1.3

Table 4.3: Cycles that separate IPC actions take

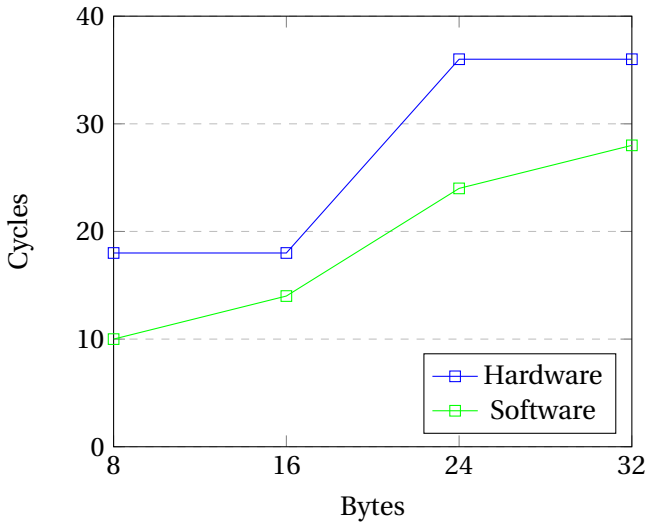


Figure 4.1: Cycles it takes to copy data from IPC

4.3. Memory tagging

Memory tagging can not be tested without the new IPC in combination with Genode because the new IPC is needed for the valid execution of Genode. That is why the memory tagging is tested

in conjunction with the new IPC. The execution times can be seen in Figure 4.2. As stated before, area is also an important metric for physical designs. The area of the tagging module consist of 234 ALM's, this is 1.3% of the total 17956 ALM's that are in use. This excludes the area needed for the caching of tags.

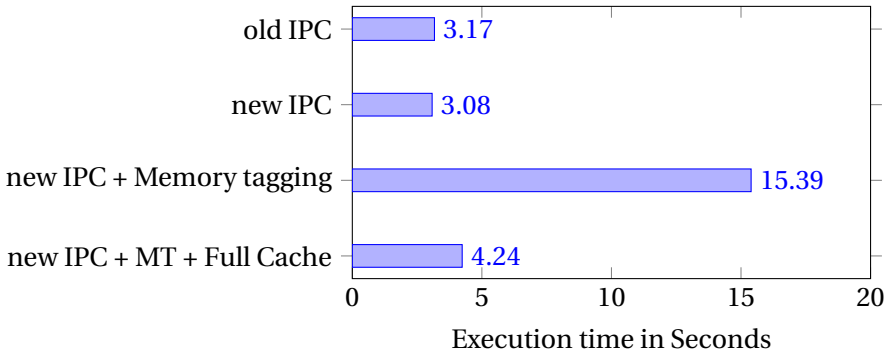


Figure 4.2: Comparison of execution times

4.4. Tag cache and TLB

The Tag cache is simulated with both the combined and separate version as described in Section 4.1. As can be seen, it already makes sense to make a cache of 1KiB as this has a hit-rate of 62.2 %. In order to view the topmost speed that can be achieved, a fully-mapped cache was implemented in the design. The results are shown in Figure 4.2. A significant speed-up compared to memory tagging without cache (3.6x), and a overhead of 26% compared to the original version.

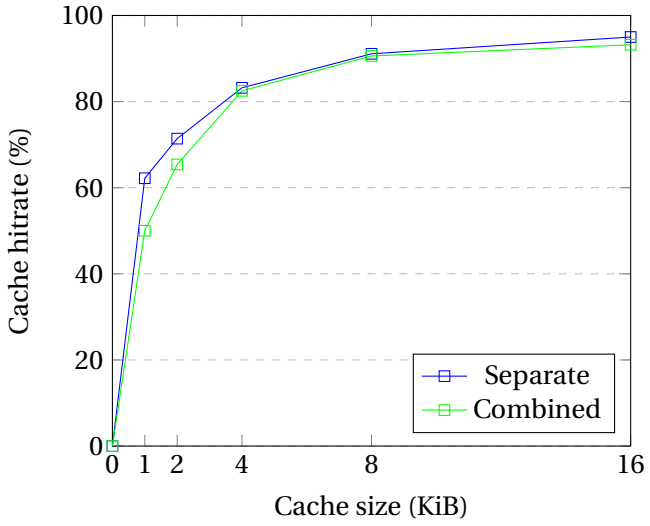


Figure 4.3: Hitrate of tag-cache

4.5. Conclusion

As stated before in Section 3, IPC has a large role in a microkernel. The speedup in execution time of the IPC and memory tagging are respectively, 1.03 and 0.2. The memory tagging is a large overhead and will make the total system less usable. By implementing a tag cache, this overhead can be decreased to an overhead of 26 percent.

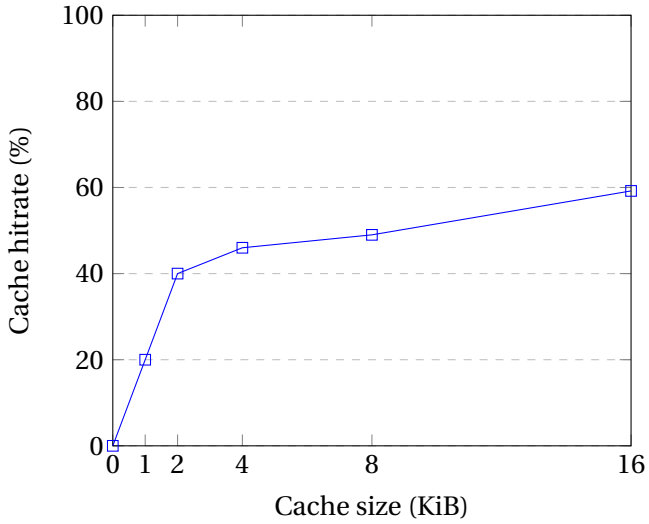


Figure 4.4: Hitrate of TLB from IPC requests

5
Conclusion

5.1. Summary

In Chapter 2 we layed-out possible attacks and defences on low-level programming languages. We concluded that one the fitting category defences for this question is *process isolation.* This is because the defence limits the freedom of processes and the kernel.

Chapter 3 showed that it is possible to give the Kernel less privileges trough a combination of memory tagging and changes to the kernel.

IPC has large role in a microkernel. The increase in execution time of the IPC in combination with the memory tagging is 500 percent. This is a large overhead and will make the total system less usable. Therefore we presented in section 3.5 methods to speed up the proposed design.

These methods prove to be effective, as we see a high hit-rate with the tag cache. The minimal slowdown is 26% with a fully mapped cache. This ultimately makes the proposed system both viable and have a low performance overhead.

5.2. Main contributions

We started with the following research question: *Can we reduce the amount of privileges of the kernel in a performance efficient manner?*.To answer this question we introduced three sub-questions which we have subsequently dealt with.

- *"What vectors are there to attack a kernel?"* This question was approached with a literature study into the different attack vectors that exist on software. This resulted in a overview of possible attack vectors and defences thereof.
- *"How can we reduce the number of privileges of the kernel?"* We developed and implemented a kernel isolation defence, which ultimately lowered the amount of privileges of a kernel.
- *"What is a performance efficient manner to implement such a restriction?"* We have reviewed the proposed system found and developed in the previous actions. This resulted in possible enhancements and delivered promising results.

5.3. Future work

Memory tagging is a widely used technique to implement a variety of defence mechanisms. An in-depth survey of those defences would be interesting as well as investigating possibilities to combine several defences with one system. This would increase the usability of a general memory tagging unit and would further justify the implementation in common hardware.

Bibliography

[1] Dirty cow (cve-2016-5195), August 2018. URL `https://dirtycow.ninja/`.

[2] National vulnerability database, Jan 2018. URL `http://nvd.nist.gov`.

[3] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[4] Lillian Ablon and Andy Bogart. *Zero Days, Thousands of Nights: The Life and Times of Zero-Day Vulnerabilities and Their Exploits*. Rand Corporation, 2017.

[5] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. Architectural support for run-time validation of program data properties. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(5):546–559, May 2007. ISSN 1063-8210. doi: 10.1109/TVLSI.2007.896913.

[6] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*, volume 41, pages 158–168. ACM, 2006.

[7] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. Tytan: tiny trust anchor for tiny devices. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.

[8] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified process replicæ for defeating memory error exploits. In *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa*, pages 434–441. IEEE, 2007.

[9] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[10] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160. USENIX Association, 2006.

[11] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[12] Karl Chen and David Wagner. Large-scale analysis of format string vulnerabilities in debian linux. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 75–84. ACM, 2007.

[13] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 5, 2005.

[14] Fernando J Corbató, Marjorie Merwin-Daggett, and Robert C Daley. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 335–344. ACM, 1962.

[15] MITRE CORPORATION. Cve-2002-0639: Integer overflow in sshd in openssh., June 2018. URL `http://cve.mitre.org/cgibin/cvename.cgi?name=CVE-2002-0639`.

[16] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[17] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Michael Frantzen, and Jamie Lokier. Formatguard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, volume 91. Washington, DC, 2001.

[18] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard tm: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.

[19] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *international conference on Information security*, pages 346–360. Springer, 2010.

[20] Jack B Dennis and Earl C Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[21] Solar Designer. "return-to-libc" attack. *Bugtraq, Aug*, 1997.

[22] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in c/c++. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):2, 2015.

[23] Úlfar Erlingsson. Low-level software security: Attacks and defenses. In *Foundations of security analysis and design IV*, pages 92–134. Springer, 2007.

[24] Norman Feske. Genode operating system framework foundations, 2018. URL `https://genode.org/documentation/genode-foundations-18-05.pdf`.

[25] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26. ACM, 2008.

[26] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 575–589. IEEE, 2014.

[27] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379. ACM, 2016.

[28] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 1970.

[29] Les Hatton. Reexamining the fault density component size connection. *IEEE software*, 14(2): 89–97, 1997.

[30] Galen C Hunt and James R Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[31] Bruce L Jacob and Trevor N Mudge. A look at several memory management units, tlb-refill mechanisms, and page table organizations. In *ACM SIGOPS Operating Systems Review*, volume 32, pages 295–306. ACM, 1998.

[32] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[33] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*, number 001, pages 13–26. Citeseer, 1997.

[34] Paul A Karger and Roger R Schell. Multics security evaluation: Vulnerability analysis. In *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, pages 127–146. IEEE, 2002.

[35] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.

[36] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *USENIX Security Symposium*, pages 957–972, 2014.

[37] Samuel C Kendall. Bcc: Runtime checking for c programs. In *Proceedings of the USENIX Summer Conference*, pages 5–16, 1983.

[38] Pankaj Kohli and Bezawada Bruhadeshwar. Formatshield: A binary rewriting defense against format string attacks. In *Australasian Conference on Information Security and Privacy*, pages 376–390. Springer, 2008.

[39] Henry M Levy. *Capability-based computer systems*. Digital Press, 1984.

[40] Jochen Liedtke. Improving ipc by kernel design. *ACM SIGOPS operating systems review*, 27(5): 175–188, 1993.

[41] Jochen Liedtke. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, December 1995. URL `http://l4ka.org/publications/`.

[42] Matt Messier. Safe c string library, 2005. URL `http://www.zork.org/safestr/`.

[43] David A Patterson and John L Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013.

[44] Nick L Petroni Jr and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 103–115. ACM, 2007.

[45] Erik Poll. Lecture notes on language-based security. 2011.

[46] William K Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Run-time detection of heap-based overflows. In *LISA*, volume 3, pages 51–60, 2003.

[47] Jerome H Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, 1974.

[48] MD Schroeder, DD Clark, JH Saltzer, and D Wells. Final report of the multics kernel design project. 1978.

[49] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC 2012*, 2012. URL `https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker`.

[50] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[51] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[52] Zili Shao, Jiannong Cao, Keith CC Chan, Chun Xue, and Edwin H-M Sha. Hardware/software optimization for array & pointer boundary checking against buffer overflow attacks. *Journal of Parallel and Distributed Computing*, 66(9):1129–1136, 2006.

[53] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[54] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: hardware-assisted data-flow isolation. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 1–17. IEEE, 2016.

[55] Ana Nora Sovarel, David Evans, and Nathanael Paul. Where's the feeb? the effectiveness of instruction set randomization. In *USENIX Security Symposium*, 2005.

[56] Andrew S Tanenbaum, Jorrit N Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

[57] PaX Team. Documentation for the pax project. *Homepage of The PaX Team*, 2003.

[58] Arjan van de Ven. New security enhancements in red hat enterprise linux v. 3, update 3, 2004. URL `http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`.

[59] Stack Shield Vendicator. A stack smashing technique protection tool for linux. *World Wide Web, http://www. angelfire. com/sk/stackshield/info. html*, 2000.

[60] TN Vijaykumar, Carla E Brodley, Benjamin A Kuperman, Ankit Jalote, et al. Smashguard: A hardware solution to prevent security attacks on the function return address. *IEEE Transactions on Computers*, (10):1271–1285, 2006.

[61] Jidong Xiao, Hai Huang, and Haining Wang. Kernel data attack is a realistic security threat. In *International Conference on Security and Privacy in Communication Systems*, pages 135–154. Springer, 2015.

[62] Suan Hsi Yong and Susan Horwitz. Protecting c programs from attacks via invalid pointer dereferences. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 307–316. ACM, 2003.

[63] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *OSDI*, volume 8, pages 225–240, 2008.