Securing Unikernels in Cloud Infrastructures

Ioannis Sfyrakis

School of Computing

Newcastle University

A thesis submitted for the degree of

Doctor of Philosophy

August 2019

I would like to dedicate this thesis to my parents for always being there for me. I also want to dedicate this thesis to my partner Foteini for her patience and support and to my newborn son Petros. I hope he will find this thesis as interesting as I have found it.

Acknowledgements

I would like to acknowledge my supervisor, Dr. Thomas Groß, for giving me the opportunity to pursue my PhD studies at Newcastle University and for guiding me throughout this journey.

I wish to thank my examiners Peter Pietzuch and Stephen McGough for their encouraging and constructive feedback. I am grateful for their thoughtful and detailed comments on improving aspects of this thesis.

I would like to thank my friends both here in Newcastle and in Greece, for all their support and encouragement throughout these years.

Certainly, I would like to thank my family, my father and mother. I would not have a chance to finish this research without their unconditional and continuous support. I would also like to express my utmost thanks to my loving partner Foteini, for her support and patience during my research. Last but not least, I would also like to thank my newborn son Petros for making life more joyful.

Abstract

Cloud computing adoption has seen an increase during the last few years. However, cloud tenants are still concerned about the security that the Cloud Service Provider (CSP) offers. Recent security incidents in cloud infrastructures that exploit vulnerabilities in the software layer highlight the need to develop new protection mechanisms. A recent direction in cloud computing is toward massive consolidation of resources by using lightweight Virtual Machines (VMs) called unikernels. Unikernels are specialised VMs that eliminate the Operating System (OS) layer and include the advantages of small footprint, minimal attack surface, near-instant boot times and multi-platform deployment. Even though using unikernels has certain advantages, unikernels employ a number of shortcomings. First, unikernels do not employ context switching from user to kernel mode. A malicious user could exploit this shortcoming to escape the isolation boundaries that the hypervisor provides. Second, having a large number of unikernels in a single virtualised host creates complex security policies that are difficult to manage and can introduce exploitable misconfigurations. Third, malicious insiders, such as disgruntled system administrators can use privileged software to exfiltrate data from unikernels. In this thesis, we divide our research into two parts, concerning the development of software and hardware-based protection mechanisms for cloud infrastructures that focus on unikernels. In each part, we propose a new protection mechanism for cloud infrastructures, where tenants develop their workloads using unikernels.

In the first part, we propose a software-based protection mechanism that controls access to resources, which results on creating least-privileged unikernels. Current access-control mechanisms that reside in hypervisors do not confine unikernels to accepted behaviour and are susceptible to privilege escalation and Virtual Machine escapes attacks. Therefore, current hypervisors need to take into account the possibility of having one or more malicious unikernels and rethink their access-control mechanisms. We designed and implemented VirtusCap, a capability-based access control mechanism that acts as a lower layer of regulating access to resources in cloud infrastructures. Consequently, unikernels are only assigned the privileges required to perform their task. This ensures that the access-control mechanism that resides in the hypervisor will only grant access to resources specified with capabilities. In addition, capabilities are easier to delegate to other unikernels when they need to and the security policies are less complex. Our performance evaluation shows that up to request rate of 7000 (req/sec) our prototype's response time is identical to XSM-Flask.

In the second part, we address the following problem: how to guarantee the confidentiality and integrity of computations executing in a unikernel even in the presence of privileged software used by malicious insiders? A research prototype was designed and implemented called UniGuard, which aims to protect unikernels from an untrusted cloud, by executing the sensitive computations inside secure enclaves. This approach provides confidentiality and integrity guarantees for unikernels against software and certain physical attacks. We show how we integrated Intel SGX with unikernels and added the ability to spawn enclaves that execute the sensitive computations. We conduct experiments to evaluate the performance of UniGuard, which show that UniGuard exhibits acceptable performance overhead in comparison to when the sensitive computations are not executed inside a enclave. To the best of our knowledge, UniGuard is the first solution that protects the confidentiality and integrity of computations that execute inside unikernels using Intel SGX.

Currently, unikernels drive the next generation of virtualisation software and especially the cooperation with other virtualisation technologies, such as containers to form hybrid virtualisation workloads. Thus, it is paramount to scrutinise the security of unikernels in cloud infrastructures and propose novel protection mechanisms that will drive the next cloud evolution.

Contents

List of Figures

Chapter 1

Introduction

The emergence of cloud computing has given corporations and individual users the ability to consume resources and services on-demand from a CSP using a pay-per-use model [8]. CSPs lower their costs by sharing the same hardware and software layers to multiple tenants. Virtualisation provides sharing and multiplexing of resources and is considered the driving force behind cloud computing. According to the World Economic Forum (WEF), 72% of corporations across the world, which operate in different industries, will adopt cloud computing by the year 2022 [142].

The advantages of cloud computing such as scalability, on-demand resources, shared infrastructure and reduced costs for businesses make cloud computing an attractive solution for business transformation and innovation. However, there are still concerns over the security cloud computing offers to its tenants. Cloud Security Alliance (CSA) in a recent report discusses the top twelve cloud computing threats, which includes system and application vulnerabilities, malicious insiders and shared technology vulnerabilities [29].

In this thesis, we focus on two main research threads for the security of cloud infrastructures. The first research thread focuses on a malicious Virtual Machine that attempts to compromise the hypervisor and access unauthorised resources such as memory, storage or network. *Privilege escalation* to kernel mode in a VM and *Virtual Machine escape* to the hypervisor [82, 39, 85, 83] are two of the vulnerabilities that malicious users have exploited in the past.

For instance, the Virtualized Environment Neglected Operations (VENOM) [44, 86] is a recent vulnerability that allows an attacker to escape from the isolation barrier that a VM provides by exploiting the floppy device driver that resides in the emulator. Having the isolation barrier violated, an attacker can elevate her privileges to access the hypervisor and in turn attack other VMs that reside on the same host. Even if the device driver is not needed it still can be loaded and used by an attacker. Therefore, this attack can be prevented if the access to the device driver is regulated. Another way this attack can be prevented is by removing support for the device driver, which results in decreasing the attack surface of the VM.

The second research thread investigates a malicious insider [38] that has access over privileged software such as the host OS and attempts to compromise the confidentiality and integrity of data that reside in a VM. For instance, a disgruntled system administrator can use memory scanning tools to compromise the confidentiality of data that reside in the memory allocated to VMs [96]. A memory dump can show sensitive information in clear text such as the root password and private keys information. The large attack surface in cloud infrastructures increases the probability that a malicious insider exploits any shared infrastructure vulnerabilities. For instance, a malicious insider can intercept information exchanged between VMs and other software components [56].

We note that VMs exhibit a large attack surface, since each VM includes a full OS and a number of applications that the tenant requires. This large attack surface is prone to introduce vulnerabilities in the software components of the VMs that malicious users can exploit. A recent research thrust aims to minimise the attack surface of VMs by creating lightweight VMs, where the build system compiles the application code with only the OS libraries and device drivers that the application needs to function. These lightweight VMs are called *unikernels*. We now follow with a precise definition of unikernels:

Definition 1.1. (Unikernel [72]) A *unikernel* is a single-address-space executable image that considers application code and device drivers as libraries and the build system compiles them with a kernel and only with the required OS libraries into a single binary file.

Even though, unikernels exhibit a number of advantages, such as minimal attack surface, reduced footprint, increased consolidation of virtualised resources [74], and near-instant booting there are disadvantages of using unikernels in terms of security.

First, unikernels execute application code without any context switching from user mode to kernel mode, which is different to the way regular OSes operate. This does not prevent the application code calling any low-level functions that interface with the hypervisor. There is not a method to evaluate if the application code has the privilege to access the kernel-level function and then to access the hypervisor-level functions. With a regular OS, a malicious user would need to perform a privilege escalation attack to gain access to the kernel that resides in a regular VM. Then the malicious user might attempt to escape the isolation barrier of the VM and attempt to infiltrate the rest of the cloud infrastructure. In contrast, a malicious user does not need to perform a privilege escalation attack in a unikernel to gain access to kernel-level functions. She can directly access low-level kernel functionality and then access hypervisor-level functions [80].

Second, the ability to minimise the attack surface does not reduce the probability of introducing a software bug in a library that is added as a dependency in a unikernel during compilation. In particular, a library that leaks information to other parties is a serious concern that would need to be addressed.

Third, unikernels do not protect from a malicious insider such as a disgruntled system administrator that has control over privileged software and can compromise the confidentiality and integrity of a tenant's computation. In addition, unikernels are given access to resources without considering regulation, since there is not any context switching from kernel to user mode. Therefore, a malicious unikernel could still exploit a vulnerability in the hypervisor and elevate its privileges [78, 99, 68]. Fourth, a single virtualised host can support thousands of unikernels. Authoring and managing complex security policies is error prone in OSes and hypervisors [4]. This is exacerbated when we create and manage security policies for a large number of unikernels, which can introduce misconfigurations and result in a vulnerability for the virtualised host. Thus, there is a requirement to design and implement software and hardware protection mechanisms for cloud infrastructures that focus on unikernels.

Protection from a malicious unikernel. One way to prevent privilege escalation in virtualised hosts that use unikernels is to use a software protection mechanism to regulate access to resources that the cloud tenant requests. One such protection mechanism can be build using an access control mechanism that uses the concept of capabilities to regulate access to resources and adhere to the Principle of Least Privilege.

Capabilities is a flexible and dynamic way to pass privileges to other processes that execute in an OS. Their inherent ability to delegate privileges to other processes and use only the required privileges makes it an ideal match for cloud infrastructures that provision unikernels. This is the main reason, we have chosen the capability-based access control model for this research project. Another reason is that capability-based access control is a mature research topic, which has been used in different contexts throughout the years. Furthermore, a cloud infrastructure with thousands of unikernels require an efficient and simple mechanism to delegate and check privileges of unikernels. By using capabilities, we can support the security policies that arise in a dynamic cloud infrastructure. Capabilities can be delegated to other unikernels according to the requirements of the application that is packaged in the unikernel. The main goal of using capabilities is to confine the behaviour of unikernels to only the rights they need in order to function. Thus, we can create least-privileged unikernels.

Protection from a malicious host. Cloud tenants do not blindly trust cloud providers to host sensitive data and applications in the cloud. It has been shown that malicious administrators [28] can use their elevated privileges to gather information regarding a VM. Similarly, a compromised OS or hypervisor can intercept a memory region of a VM and steal sensitive information. This issue also afflicts unikernels, while executing in a virtualised host as a VM and as a regular OS application. Hardware-based protection mechanisms have been used in the past to protect memory regions that contain sensitive information. These mechanisms provide isolated execution environments called Trusted Execution Environment (TEE) [48]. One example of a TEE is Intel Software Guard Extensions (SGX) that provides confidentiality and integrity of sensitive computations and data that reside in a process.

We define the problem statement for this thesis as: **how to protect cloud infrastructures that focus on unikernels using software and hardware-based mechanisms.**

1.1 Aim and Objectives

The aim of this work is to investigate software and hardware-based protection mechanisms for virtualised hosts, where cloud tenants implement their workloads using unikernels.

Even though there are no reports of exploits that specifically use unikernels to direct attacks on cloud infrastructures so far, there are numerous attacks documented using regular VMs with a severe impact on the security of cloud providers and the level of trust cloud tenants have for them. There are also attacks that use privileged software to pry on software running on the physical computer and also using certain hardware attacks.

The strategy that we follow in this thesis for realising our main goal is to first evaluate the current access control system that Xen uses while executing unikernels. After exposing the shortcomings of the current access control system we develop a new access control system that is tailored for unikernels. First, we develop an access control system that uses capabilities, and then we build support for hardware-based protection for unikernels using Intel SGX.

This thesis contains the context and specifics for the objectives we determined as necessary to present our final solution. These objectives provide a clear definition of the scope of our work, demonstrate the security problem we studied, and present our proposed solution, results, and its security and performance evaluation. This document should help the reader understand our motivation for software and hardware-based attacks in virtualised hosts, the reasons supporting the design decisions for our solution, and how the suggested solution improves on the current state of the art. Therefore, this document focuses on the background and previous literature, elaborating on how contemporary access control systems are not integrated with unikernels and do not leverage their advantages, and also discussing our solution and its applicability.

The background and literature review content provides information on key subjects such as cloud computing, security, access control, and an analysis of the current state of the art in virtualisation access control using hardware-based mechanisms. This analysis focuses on how effectively cloud security mechanisms confine unikernels to their specified behaviour. These chapters provide a clear view of the scope of our research. The concepts and ideas explained in these sections are of prime importance for understanding the chapters that follow.

After the background and literature review chapters, we introduce and explain the current Xen access control mechanism and its pitfalls when using unikernels. Another important point in this chapter is that it demonstrates how unikernels are not accountable for their behaviour.

The remainder of the chapter describes our solution in great detail, discussing how we implemented and evaluated its efficacy, and the way this solution can improve the access control of current approaches. Our solution first introduces a software-based mechanism that introduces the capability-access control model in the Xen hypervisor. This mechanism regulates access to resources and guarantees that unikernels will not breach confinement according to the capabilities they hold. This method restricts the access of a unikernel only to the required privileges needed to finish the task.

We then introduce a hardware-based mechanism that leverages Intel SGX to provide confidentiality and integrity for computations executing inside unikernels. Using this approach, unikernels can withstand attacks that originate from privileged software and hardware.

In summary, this research project aims to satisfy the following objectives:

Objective 1: Create a new Xen Security Module that introduces capability-based access control for the Xen hypervisor that we can use as a better alternative of the current Flask module.

Objective 2: Create a unikernel library for enabling the capability-based paradigm in unikernels.

Objective 3: Create a security architecture leveraging Intel SGX to provide confidentiality and integrity for computations inside unikernels.

Objective 4: Create a shim library that sanitises information to and from the enclaves in unikernels.

1.2 Thesis Contributions

This thesis contains a number of contributions related to securing unikernels in cloud infrastructures, which we outline below.

1. A detailed literature review of the current state of the art in access control and

protection mechanisms for cloud infrastructures. Our analysis focuses on solutions that provide software or hardware-based protection mechanisms. The insights from this review led us to identify the research gap we targeted in this thesis.

2. In protecting the virtualised host from a malicious unikernel, we contribute the following:

   - We design and implement a novel software-based access control system that enforces the POLP, and guarantees confinement of unikernels hosted in a virtualised host.
   - We conduct a performance evaluation to show the efficiency of the access control system in relation to other software-based access-control mechanisms that the hypervisor can use.

3. In protecting the unikernel from a malicious host, our contributions are as follows:

   - We design and implement a novel security architecture that protects unikernels against privileged software and certain physical attacks from a compromised host.
   - We conduct a performance evaluation to elicit the overhead of our research prototype.

1.3 Publication History

The contributions in this thesis were first published in peer reviewed conference papers written or co-authored by the author.

- I. Sfyrakis and T. Groß. VirtusCap: Capability-based Access Control for Unikernels. *IEEE Cloud Engineering Conference (2017)*, 2017
- I. Sfyrakis and T. Groß. UniGuard: Protecting Unikernels using Intel SGX. Technical Report CS-TR, Newcastle University, September 2017

- I. Sfyrakis and T. Groß. UniGuard: Protecting Unikernels using Intel SGX. *IEEE Cloud Engineering Conference (2018)*, 2018

In addition, the following journal, conference papers and technical reports were produced during the course of the PhD on related topics but do not form part of the thesis.

- E. Solaiman, I. Sfyrakis, and C. Molina-Jiménez. Dynamic Testing and Deployment of a Contract Monitoring Service. In *5th International Conference on Cloud Computing and Services Science (CLOSER 2015)*, pages 463–474. SCITEPRESS, 2015
- E. Solaiman, I. Sfyrakis, and C. Molina-Jiménez. High Level Model Checker Based Testing Of Electronic Contracts. In *Cloud Computing and Services Science*, volume 581, pages 193–215. Springer International Publishing, 2016
- E. Solaiman, I. Sfyrakis, and C. Molina-Jiménez. A State Aware Model and Architecture for the Monitoring and Enforcement of Electronic Contracts. *18th IEEE Conference on Business Informatics*, 2016
- E. Solaiman, I. Sfyrakis, and C. Molina-Jiménez. High Level Model Checker Based Testing Of Electronic Contracts. Technical Report CS-TR-1490, Newcastle University, January 2016
- E. Solaiman, I. Sfyrakis, and C. Molina-Jiménez. Dynamic Testing and Deployment of a Contract Monitoring Service. Technical Report CS-TR-1460, Newcastle University, January 2015
- C. Molina-Jiménez, E. Solaiman, I. Sfyrakis, I. Ng, and J. Crowcroft. On and Off-Blockchain Enforcement Of Smart Contracts. In *International Workshop on Future Perspective of Decentralized Applications (FPDAPP 2018)*, 2018
- C. Molina-Jiménez, E. Solaiman, I. Sfyrakis, I. Ng, and J. Crowcroft. On and off-blockchain enforcement of smart contracts. *arXiv:1805.00626 [cs.CY]*, 2018
- C. Molina-Jiménez, I. Sfyrakis, E. Solaiman, I. Ng, M. W. Wong, A. Chun, and J. Crowcroft. Implementation of smart contracts using hybrid architectures with on- and off-blockchain components. *arXiv:1808.00093 [cs.SE]*, 2018

- C. Molina-Jiménez, I. Sfyrakis, E. Solaiman, I. Ng, M. W. Wong, A. Chun, and J. Crowcroft. Implementation of smart contracts using hybrid architectures with on- and off-blockchain components. *8th IEEE International Symposium on Cloud and Services Computing (IEEE SC2)*, 2018

- T. Groß and I. Sfyrakis. Specification of the Graph Signature Cryptographic Library and the PRISMACLOUD Topology Certification Version 0.9.2. Technical Report CS-TR-1523, Newcastle University, July 2018

1.4 Thesis Outline

The remainder of this thesis is organised as follows: Chapter 2 provides the background context, it discusses the key concepts required to understand later discussions in the thesis. Chapter 3 provides a literature review of related research work. This chapter identifies the research gap that we target in this work, and also discusses how this gap relates to similar research. Chapter 4 discusses our approach to designing and implementing a capability-based access control system for unikernels. Chapter 5 presents the security architecture that protects unikernels from an untrusted cloud. Chapter 6 presents our conclusions and suggestions for future work.

Chapter 2

Background

This chapter introduces the required concepts to understand the remainder of the thesis. Additional specific concepts and related technologies are introduced in the relevant sections.

In this thesis, we are concerned with two research threads for the security of cloud infrastructures. According to a recent report 91% of organisations are concerned about cloud security [35]. For this chapter, we provide an overview of the main security requirements and design principles that were followed during the research project. The security requirements provide a starting discussion point for the remainder of this thesis. We also need to discuss the design principles, since they guide the design of the two proposed solutions in this thesis. In addition, we discuss the main technologies used in this research project such as cloud computing and its main characteristics alongside virtualisation, hypervisors and unikernels.

The first research thread regards the protection of a hypervisor from a malicious unikernel. The main issue for the protection of hypervisors from a malicious unikernels is that there is not a context switching mechanism present in a unikernel. The kernel libraries, application code and system libraries are bundled inside a unikernel machine image. This introduces the opportunity for a malicious library to have access to the whole interface of the hypervisor. The malicious unikernel can then take advantage of the unprotected interface and launch attacks targeting the hypervisor and taking advantage of any vulnerability of the hypervisor. This kind of attacks have the potential to cripple any virtualised host and in extension the whole cloud infrastructure. Another issue with unikernels is the complex security policies that arise from the dynamic nature of cloud infrastructures. Thus, it is of paramount to resolve this issue for cloud infrastructures that focus on unikernels. Therefore, we would need first to define the security requirements and design principles that underline this research project. In addition, we define the main terms for the cloud computing research area, which is relevant to this research project. For the first research thread, we define the access control model that we are using to develop the access control mechanism for unikernels.

The second research thread discusses the protection of a unikernel from a compromised privileged software such as a hypervisor. Regarding this research thread, we are interested in the case that a malicious insider takes over privileged software and attempts to exfiltrate information from a unikernel that is not protected. Therefore, we need to discuss the different types of Trusted Execution Environments and in particular Intel SGX. A detailed overview of Intel SGX discusses the main features and architecture needed for the second solution.

The remainder of this chapter is organised into four sections. It starts with the introduction of the security requirements and the design principles. The second section presents the capability-based access control. The third section provides the background information regarding cloud computing and next section discusses the concept of virtualisation, and introduces the Xen hypervisor, the KVM, and the concept of unikernels. The final section presents an overview of TEEs such as the Trusted Platform Module, ARM TrustZone and the Intel SGX.

2.1 Security Principles

This initial section introduces the requirements for the security of a system. Subsequently, we present the reader with the design security principles that a system should employ to protect its information.

2.1.1 Security Requirements

The main requirements for the security of a system are confidentiality, integrity, and availability. We follow the terminology of the National Institute of Standards and Technology (NIST) [95] for these security requirements.

Confidentiality is the security requirement that private or confidential information should not be disclosed to unauthorised individuals. For instance, military systems provide access to information on a need-to-know basis only [19]. Users of a military system are given access to information according to their security clearance. They cannot read any information above their security level. This practice is regularly employed in military and government institutions.

Integrity means that the system state cannot be altered by unauthorised users. Integrity includes data integrity, system integrity, and origin integrity (authentication).

Data integrity means that alteration of information and applications must only happen in a specified and authorised manner. For instance, Alice sends an amount of money to Bob as payment for purchasing goods using a bank's interface of fulfilling payments. In the absence of data integrity a malicious user is able to modify the information regarding the payment, this person could divert the money to a different bank account.

System integrity means that the system must execute the expected functionality without any unintended or deliberate unauthorised modification. For instance, if an intruder gains access to the central processing unit, it is usually possible to reboot the system and bypass logical access controls. This can lead to information disclosure, fraud, replacement of system and application software, the introduction of a Trojan horse, and more. Moreover, if such access is gained, it may be very difficult to determine what has been modified, lost, or corrupted.

Availability means that legitimate users have the ability to access the required data or resource. An adversary can use a Denial of Service (DoS) attack to compromise the availability of a system by denying legitimate users access to data or resources. The adversary floods the system with an enormous number of illegitimate requests with which the system cannot cope. When a legitimate user attempts to access data or resources, the system cannot fulfil the request.

2.1.2 Design Principles

Saltzer et al. [104] introduced a number of design principles during the early days of computer security. These principles were crafted after designing systems that included security requirements. The experience of designing such systems led to the design principles that guide new software designs. The design principles are the following: *fail-safe defaults*, *economy of mechanism*, *complete mediation*, *open design*, *separation of privilege*, *least privilege*, *least common mechanism*, and *psychological acceptability*. We now discuss each design principle and how it can be used to evaluate the system designs.

The **Economy of Mechanism** principle states that the design of the security mechanisms should be as simple as possible. Large and complex systems with a wide interface result in a large attack surface that an adversary can exploit to compromise the security of the system. Realising such a principle leads to a mechanism with which it is easier to verify security properties compared to a complex design.

The **POLP** principle states that a subject should be given only those privileges needed to complete their task. The main aim of this design principle is to contain the damage that can happen when there is improper use of privilege.

Another design principle to take into consideration is the **Separation of Privilege**. This principle states that a system should grant approval for an operation only when checking several conditions. This means that a single privilege is broken into several other components and multiple agreements are needed to execute an action. For instance, in a system that involves multiple sensors, such as that used in biometric authentication, all must agree on certain conditions to grant approval for an operation.

The **Complete Mediation** design principle states that every access to an object should be checked to ensure access is allowed. This implies that a foolproof method of identifying the source of an access request must be created. In this way, a system-wide view of access control is created, which includes the initialisation, recovery, shutdown, and maintenance stages of operation, and also the stage of normal operation. Hence, this is a foundation on which to build a protected system, and protection mechanisms should be deployed to identify the source of all requests.

For the **Fail-Safe Defaults** design principle the default result when a subject is given explicit access to an object is that her access should be denied. Incorporating this principle in the design of the system enables the possibility to refuse a permission that has failed due to a design or implementation mistake.

The **Least Common Mechanism** design principle states that the amount of mechanisms shared by more than one subject should be kept to a minimum. A mechanism that is shared among many parties provides an information path that must be considered in such a way as to prevent the breaking of the security of the system.

The security of a mechanism should not depend on secrecy of design or implementation. This is what the **Open Design** principle states. An open design can improve a security design when multiple stakeholders can review it, provide feedback, and improve the design of the system.

The design principle of **Psychological Acceptability** states that security mechanisms should not make the resource more difficult to access than if security mechanisms were not present. Keeping the balance between security and usability is of paramount importance to the research community. A system that is secure but very difficult to use will not be easily accepted by users and we could also encounter a decrease in user productivity.

The most relevant design principles for the research presented in this thesis are economy of mechanism, complete mediation, and least privilege. We associate the economy of mechanism principle with a minimal *Trusted Computing Base*. This implies that having a system with a small TCB creates a minimal attack surface which is important when executing trustworthy computations with strong security properties. The principle of least privilege is the cornerstone of our research and ensures that a software computation does not access any additional functionality or data that is not necessary to accomplish its task. Assuring the complete mediation design principle in a system results in unnecessary accesses being prevented and accounted for when needed. This creates a secure system that includes security assurance and logging of information that captures the access requests.

2.2 Capability-based Access Control

This section introduces the concept of the capability and by extension, the capability-based access control paradigm. We start by presenting the first research paper that mentions capabilities and define a capability and a capability list. Subsequently, we discuss the ways in which we can use capabilities and how to protect them.

The term "capability" was first coined by Dennis and Van Horn [36]. According to the authors, capabilities is a data structure that locates a computing object using an effective name, and indicates the actions that the computation may perform on that object. In essence, a *capability* is an unforgeable token that is used to regulate access to a resource, according to a set of rights. It is a value that uniquely references an object and is associated to a set of rights. When a process or, more formally, a subject possesses a capability that references the object, then the capability token grants the process the capability to access the object in certain methods.

In their seminal work on capabilities, Dennis and Van Horn [36] illustrate how capabilities are used which we outline in the following. Every process in an OS is associated with a capability list (C-List), which can only be manipulated via meta-instructions implemented by the supervisor. In order to authorise an action of the process, a capability must be present in its C-List which is accessed by index via meta-instructions. The capabilities permit a number of access modes to system objects such as memory segments, processes, input/output devices and capability storage. In their model capabilities can be transferred when creating new processes, during inter-process communication, or when loaded and stored in directories. In addition, their model allows the creation of entry capabilities from a segment capability defining a protected procedure and a set of capabilities to be made available to the procedure while it executes. The state specified in the entry capability initialise the protected entry point with the extra capabilities.

Keys are the most frequent metaphor for describing systems that use capability-based access control mechanisms. Capabilities can be seen as keys to locked objects. For instance, if you want to enter your house or start your car, you need to possess the right key, which is unique for this object. Keys can be copied and distributed to other people, sent through the mail, or stored securely in a safe. The same operations can be used with capabilities. A key that is well made is extremely difficult to forge, and secure locks are difficult to circumvent.

A *Capability list (C-List)* for a subject is a set of pairs. Each pair contains an object and a set of rights. The only way that a subject can access a particular object is according to the rights outlined in the capability list for that object. There is a capability entry in the C-List according to the index. In order to find a particular capability for the object we first search the C-List. The capability entry in the C-List includes the identifier, which is an address or a name to a resource such as a segment of memory or a file, and an access right, which can be read, write, execute, etc. The next step is to check the capability and the rights encoded in the capability to find, if access to a particular resource object is permitted or not.

There are two ways of using capabilities. First, we can use the capabilities in an explicit manner. This means that a subject has to show the capabilities she possesses explicitly to gain access to the object. A simple example is when we show a ticket to the person in charge at the cinema so that she can allow us access to the auditorium and we can watch the film. Second, the implicit use of capabilities means that it is not necessary to present the capabilities and the system will automatically evaluate if a particular subject has the correct capabilities. An example of this approach is the C-List, mentioned earlier. Each process has a list of capabilities attached. If the process tries to access an object, the access control mechanism traverses the C-List to evaluate that the process has the correct capability. Comparing the two approaches, we can see that the second approach of using a C-List would not be efficient when the list is very long. The second approach is easier to use since the capabilities are transparent to the subjects and the subject is not required to present the capabilities.

One question that arises during the implementation of a capability-based access control mechanism is where to store the capabilities in a system. Capabilities play a very important role in system security. Whenever a capability is issued to a subject, the subject must be prevented from tampering with the capability. There are three main methods to solve this issue. First, the capabilities can be stored in a protected place. Using this approach, the subject cannot adjust the capabilities and can only use them implicitly. An example of this approach is using the kernel of an OS to protect capabilities that reside in a C-List. The second approach is to place the capabilities in an unprotected place and use cryptography to prevent them from being tampered with. The third approach is a combination of the first two methods. This approach includes the subject using the capabilities explicitly, but keeping them stored in a protected place. This would work as follows: capabilities are stored in a list that is placed in a safe place such as a kernel. The subject is given the index to the capabilities and present it to the system to use a capability explicitly. The advantage of this approach is that forging an index does not grant the subject any additional capability.

The main idea to consider for the security of capability-based access control mechanisms is to prevent tampering or spoofing of capabilities. The system needs to guarantee that a capability remains an unforgeable token. If capabilities can be adjusted by any process or actor in the system, then the access rights can be amplified and allow access to other parts of the system. There are different methods of ensuring the security

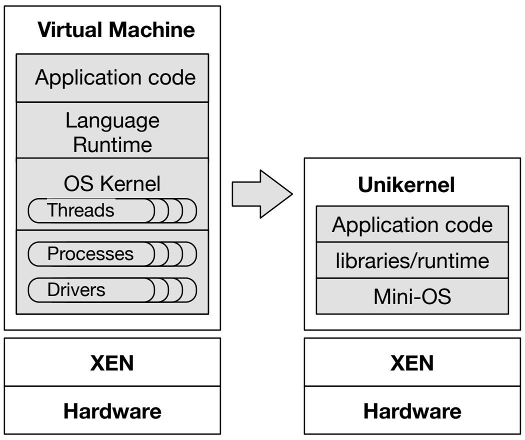


Figure 2.1: MirageOS Unikernel

of capabilities using software and hardware-based approaches.

In the following, we discuss three mechanisms that protect capabilities. First, tagging is a hardware mechanism that has a set of bits associated with each hardware word. Each tag has a set and unset state. If the tag is set, then it cannot be changed. The tag can only be set by the Central Processing Unit (CPU) in a privileged mode and not by any user process.

Second, protection bits are used in paging or segmentation. Capabilities are stored in a page or memory segment and cannot be modified by the process. Even though, this mechanism does not require any specially built hardware, it does require processes to reference capabilities indirectly through the use of pointers.

Third, cryptography can be used to protect capabilities. One of the ways that this is accomplished is by using a cryptographic hash function. The main goal is to safeguard the integrity of capabilities. This works by having a checksum associated with each capability that is digitally enciphered using a cryptographic key known to the OS. A user process sends a capability to the OS and recomputes the checksum related to the capability. Then, the OS enciphers the checksum using the cryptographic key and checks if the checksum matches that stored in the capability. The capability is not modified if there is a match. Otherwise, the capability is not taken into consideration by the OS.

Unikernel OS	Clean-slate approach	Support for legacy applications
MirageOS [71, 72]	✔	✕
HaLVM [43]	✔	✕
LING [30]	✔	✕
EsseOS [123]	✔	✕
IncludeOS [22]	✔	✕
OSv [61]	✕	✔
Rump Kernels [60]	✕	✔
Drawbridge [98]	✕	✔
ClickOS [75]	✕	✔

Table 2.1: Categorisation of unikernel OSes, whether or not they support the running of existing software applications in a unikernel.

2.3 Unikernels

Current OSes are designed to execute generalised software applications making use of the hardware resources, multiple processes, kernel services, and system libraries. OSes are intended for multiple purposes and are multi-user environments. Unikernels on the other hand, are specialised single address space VMs that are created using library Operating Systems (libOSes) [71]. Library operating systems bundle application code and configuration and system services such as the scheduler or device drivers that are implemented as libraries. These libraries are directly linked with the application. The library operating system method has a different structure than monolithic OSes, improves performance, and decreases the footprint the applications require since we do not need a full OS to execute the application. Also, we do not need to have multiple users in a unikernel, as we would have in a regular OS. The different software layers that comprise a unikernel in comparison to the software layers of a regular OS are illustrated in Figure 2.1. In this figure, we depict a unikernel created using MirageOS, which we discuss in the next subsection.

Unikernel OSes are minimal OSes that use mainly the Xen hypervisor as a hardware abstraction to execute unikernels. Other hypervisors have been used to support unikernel OSes, such as the KVM hypervisor. In this thesis, we focus on the MirageOS unikernel OS that uses the Xen and KVM hypervisor. The MirageOS architecture is illustrated in Figure 2.2, and what follows is an outline of the advantages and disadvantages of MirageOS unikernels.

In the past, researchers have demonstrated different methods that restructure a general purpose OS or construct a new OS that uses libraries of the OS and application in the same address [98], [41]. More precisely, the application is linked together with a set of libraries that provide a high-level set of abstractions. These abstractions take the place of OS functionality such as processes, files, Inter Process Communication (IPC), network, and threads.

Recently, we have witnessed the repurposing of library operating systems to cloud computing in order to improve performance and security. There are many efforts in this research area to build unikernel OSes. We separate unikernel OSes in two categories depending on whether they use a clean slate approach or if they can accommodate current applications in their unikernels. This is illustrated in Table 2.1.

A recent development is the introduction of cloud Operating Systems (cloud OSes) [71, 72] that construct unikernels. We define a *Cloud Operating System* as a method to construct OSes whereby we replace the OS with a language runtime and construct single-purposed application VMs targeting cloud computing services. Most of these cloud OSes rely on Xen, KVM, or NetBSD to provide the hardware abstractions, i.e. MirageOS [71, 72], EsseOS [123], Erlang on Xen [30], OSV [61], and rump kernels [60].

Both MirageOS and EsseOS take a clean-slate approach and build safe and secure unikernels. In contrast, OSv and rump kernels provide compatibility with legacy applications and OSes. Hence, we can group cloud OSes in two categories: first, cloud OSes that use a clean-slate approach and use only one programming language to create unikernels such as OCaml in MirageOS and Haskell in EsseOS; second, cloud OSes that aim for compatibility with legacy software and focus on supporting OS interfaces such as the *Portable Operating System Interface (POSIX)*.

The main premise of unikernels is that they have a smaller file size, a smaller attack surface, and near-instant booting. Unikernels can be created with a small file size that often does not surpass 1MB. Thus, we can have thousands of unikernels in a single physical host. To deal with this situation, we would need to have a dynamic and flexible architecture that uses a robust access control system to regulate access to resources.

Of course, it is still an open research question as to how we can build an access control system that focuses on unikernels and supports their advantages and behaviour. Furthermore, how can we leverage unikernels to construct isolated coalitions of unikernels that span multiple physical hosts? Finally, how can we provide a capability-based approach to manage access and create least-privileged architectures? Unikernels are built following the principles of library OSes. Therefore, a unikernel only uses the libraries it needs to function properly.

For this research on unikernels, we have chosen to design and implement our access control system on top of MirageOS. As mentioned in the previous subsection, MirageOS is a cloud OS that constructs unikernels. MirageOS, like most of the other cloud OSes, relies on Xen to provide an abstraction over physical resources. This makes it easier to reuse device driver models that already exist for Xen rather than reinventing the wheel and developing device drivers from scratch, which can be time consuming and error prone. Figure 2.2 illustrates the MirageOS architecture in relation to Xen.

What follows is a discussion of the MirageOS unikernel OS and our rationale for choosing it.

2.3.1 MirageOS

In this subsection, we provide an overview of the unikernel OS that was used during the course of this research project.

MirageOS is a unikernel OS that started as a research project at Cambridge University. It is a library OS that generates specialised single-address space machine images called unikernels. It promotes a clean-slate approach with a focus on security and safety.

The application code is (mostly) independent of the used back-end. To achieve this, the language that expresses the configuration of a MirageOS unikernel is rather complex, and has to deal with package dependencies, setup of layers (network stack starting at the (virtual) Ethernet device, or sockets), logging, and tracing.

The abstraction over concrete implementation of, for example, the network stack is done by providing a module signature in the mirage-types package. The socket-based network stack, the tap device based network stack, and the Xen virtual network device based network stack implement this signature (depending on other module signatures). The unikernel contains code that applies those dependent modules to instanti-

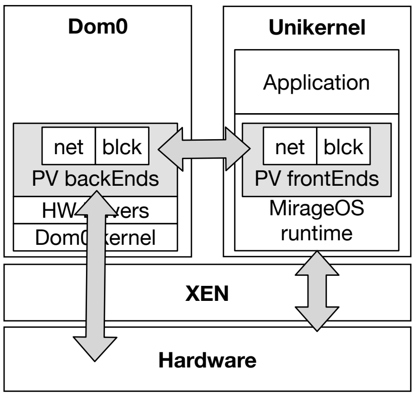


Figure 2.2 MirageOS architecture. MirageOS utilises the paravirtualized device drivers that Xen provides for network and block devices. Xen has a split device model which means that each device driver is split into two. One is the back-end driver that communicates with the physical hardware and the other is the front-end driver that is used by the unikernel. MirageOS has implemented fron-tend device drivers using OCaml language to ensure type-safety and to minimise runtime errors that might exist in the implementation.

ate a custom-tailored network stack for the specific configuration. A developer should only describe what their requirements are, and the user who wants to deploy it should provide the concrete configuration. The developer should not need to manually instantiate the network stack for all possible configurations as this is what the mirage tool should embed.

The only language that MirageOS developers can use is OCaml. The main reasons for choosing a higher-level language instead of using C are the following:

1. Static type checking means that unsafe code is rejected at compile time instead of execution time.

2. Automatic memory management that eliminates resource leaks.

3. Promotes modularity and abstraction

4. Metaprogramming allows compilers to optimise the program much more if the runtime configuration is understood.

MirageOS uses OCaml's powerful module system and strict static type checking to create native OS libraries that can be utilised in a unikernel. For instance, a unikernel that serves static HTTP content can have the following structure: First, the application MyBlog depends on an HTTP signature that is provided by the Cohttp library. Unikernel developers first test and debug their code using a Unix-style environment. Therefore, the Cohttp library needs a TCP implementation to satisfy its module signature, which can be provided by the UnixSocket library. After the developer finishes the development of the unikernel, the Unix side is dropped and the Xen-related libraries are used during recompilation. Now, the unikernel developer can deploy the unikernel as a regular VM at a cloud service such as Amazon EC2.

The MirageOS architecture is depicted in Figure 2.2. It uses the Xen hypervisor as the underlying hardware abstraction. MirageOS unikernels include the MirageOS runtime, the required paravirtualized front-end device drivers for network, or block devices that communicate with the back-end device drivers that reside in Dom0.

The MirageOS runtime includes the MiniOS base layer that provides the necessary low-level functionality and interface to the Xen hypervisor. MiniOS is coded in the C programming language and the higher software layers are programmed in the OCaml language.

The main advantages of using MirageOS unikernels are the following: First, unikernels have a small footprint and file size since there is no need for a full-blown OS. Second, unikernels have low booting times since they do not need a full OS. Third, there is better utilisation of resources since only a minimum of resources are needed. For instance, a unikernel that serves a static HTML web page needs only 16MB of memory. Also, there is no process switching overhead since there is only one process. Fourth, unikernels have a minimised attack surface since they only need a number of libraries to link with the application rather than a full OS. The interface used in unikernels is minimal in relation to a VM with a full OS in which a vulnerability can exist. Fifth, unikernels are further protected using sealing. Sixth, Address Space Randomization (ASR) is used in order to make it impossible for malicious users to circumvent the protection of unikernels. Seventh, MirageOS unikernels can be booted on-demand to service a request from a client [70].

However, there are also certain disadvantages when we utilise MirageOS unikernels in the cloud. First, the unikernels use only one programming language and toolchain for developing the unikernels. Of course, there are some unikernels that can use different programming languages for development, but the majority of unikernels use only

Advantage	**Disadvantage**
Smaller footprint.	Need to rearchitect legacy applications.
Faster booting times.	No multiple processes/fork.
Better utilisation of resources.	Lack in debug/logging mechanisms.
Minimised attack surface.	Lack in management/orchestration platforms.
Sealing.	Only one programming language.
Address Space Randomisation(ASR).	Consumes the whole memory allocated.
Just-in-time summoning [70].	Not best used for device drivers.
Multiple building targets	-

Table 2.2: Advantages and disadvantages of MirageOS unikernels

one. Second, there is a lack of orchestration and management systems that coordinate and manage a large number of unikernels in a single host. Third, as a new development paradigm, it lacks the libraries for developing any application. However, this will improve in time and as more libraries are developed. Fourth, there is a lack of debugging tools in runtime and a way to test unikernels with little overhead. In addition, there is a lack in logging mechanisms that could showcase where an error happened during the execution of a unikernel. The main advantages and disadvantages of using MirageOS unikernels to implement cloud workloads instead of regular VMs are illustrated in Table 2.2.

2.4 Cloud Computing

In this section, we define cloud computing that unikernels use a platform and discuss its main characteristics and deployment models. According to the National Institute for Standards and Technology (NIST), cloud computing is defined as follows:

"*Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.* " [79]

Cloud computing, or the cloud, is considered the placeholder of executing code and data in servers placed in virtualised data centres throughout the Internet. It provides

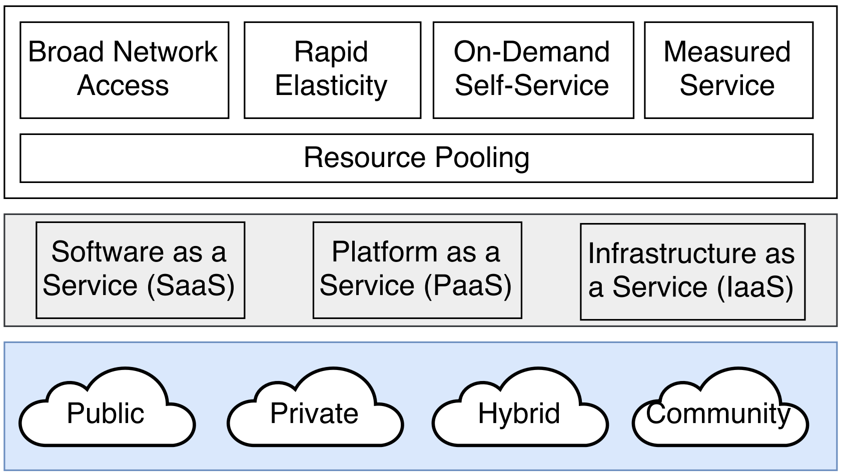


Figure 2.3: NIST model of cloud computing [79]

IT services to any company or individual using a pay-per-usage model, which resembles the way we purchase electricity or gas. Examples of these services include the Amazon Elastic Cloud Computing [1], for provisioning CPU power to different users, and Google App Engine [50], which facilitates the development of web services and applications alongside the virtualised infrastructure.

Figure 2.3 depicts the cloud computing model as specified by NIST. There are five essential characteristics, three cloud service models, and four deployment models. We discuss the elements of this model in the following subsections after defining the entities in cloud computing. The main entities of cloud computing are the cloud provider and the cloud tenant. The cloud provider is the entity that owns the physical infrastructure that allocates resources from a shared pool to cloud tenants. The second main entity is the cloud tenant or tenant, who is the consumer of the cloud computing service; this can be an individual user or a company. Other entities include the users that are responsible for the management of the infrastructure and resources of the cloud. There are administrators at different levels of the cloud such as the cloud administrator, who is responsible for the whole infrastructure. The tenant administrator is responsible for the management of a tenant's infrastructure. Each host in the infrastructure is managed by a host administrator.

2.4.1 Features

The main attributes of cloud computing as depicted in Figure 2.3 are the following: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service.

- **On-demand self-service**: The computing resources in cloud computing are provisioned automatically without any human intervention.
- **Broad network access**: There are standardised access control mechanisms and the network access to resources that use the client platforms.
- **Resource pooling**: The resources in cloud computing are gathered in a resource pool where computing resources are distributed to individual tenants. All the resources are assigned to a tenant in a dynamic way.
- **Rapid elasticity**: The tenant can request resources that are satisfied elastically, which gives the impression that there is an unlimited pool of resources.
- **Measured service**: Cloud computing embraces a utility model whereby the resource usage of a tenant is automatically measured and billed.

2.4.2 Deployment Models

According to NIST, there are four different cloud deployment models: private cloud, public cloud, hybrid cloud, and community cloud. The following list provides an analysis of each model and discusses the advantages of each one.

- **Public cloud**: This deployment model is used by the general public. The cloud provider is responsible for the management and operation of the cloud infrastructure and can be a company, a government, or a research organisation. The cloud infrastructure for this deployment model resides on the premises of the cloud provider. Even though this deployment model is open to the public, security risks exist for tenants and providers.
- **Private cloud**: This deployment model is used by a single organisation, for instance, a company that provides resources to tenants. Tenants in this model

can be different business units and even individual employees. The location of the cloud infrastructure can be on or off premises. The main disadvantage of this model is that it requires a higher investment in equipment and resources.

- **Community cloud**: This model involves a cloud infrastructure that is shared by a set of organisations that have shared objectives. The cloud infrastructure for this model can be on or off premises. The shared stakeholders of the community cloud can be involved in the management and operation in any combination. One advantage of the community cloud is that the costs are shared among the different stakeholders. In contrast, sharing the infrastructure with different organisations includes security risks for tenants.

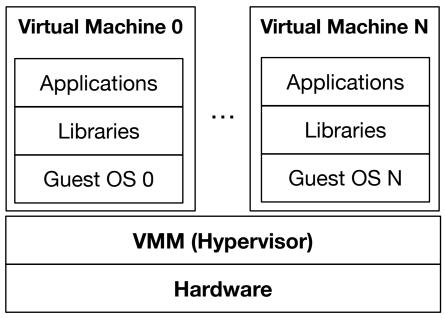
- **Hybrid cloud**: Two or more cloud infrastructures are combined using standardised or proprietary methods to realise the application execution in different clouds. The advantage of this model stems from the standardisation of the interactions between different cloud infrastructures. However, the disadvantage of this model is that the cost and the level of security measures in each of the clouds can vary depending on which cloud models are combined to realise the hybrid model.

2.4.3 Service Models

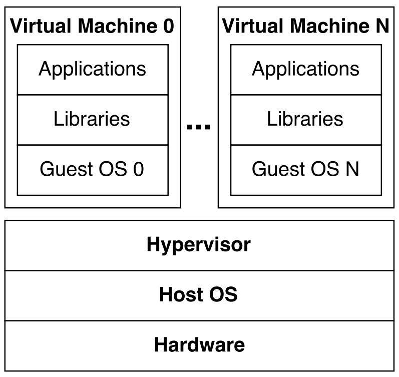
The three service models according to NIST are the following: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

- **Software as a Service**: In this higher-level software service layer, the tenant can control only a limited set of application configuration settings. Hence, the tenant can only use applications and services the provider offers.

- **Platform as a Service**: This medium-level service layer gives tenants the ability to deploy applications they have developed to the cloud infrastructure. Apart from deploying their applications, tenants select applications that have been implemented using software and tools that the provider offers. Tenants in this layer do not have any control over the physical or virtualised infrastructure. They only control the deployed applications and their configuration.



(a) Type I hypervisor



(b) Type II hypervisor

Figure 2.4: Different types of hypervisors

- **Infrastructure as a Service**: This lowest service layer provides the tenant with the list of computing resources that are available. The tenant can select the appropriate processing, storage, and network resources according to her requirements, which are under her control. Consequently, the tenant can execute and deploy any software such as OSes and application to the provisioned resources. However, the provider controls the physical and virtualised infrastructure.

2.5 Virtualisation

The objective of this section is to introduce the main concepts of virtualisation. This is complemented by an introduction to two popular virtualisation solutions: the Xen hypervisor and the Kernel-based Virtual Machine (KVM), which are used in today's cloud infrastructures. The last part of this background section focuses on a new technology called unikernels based on the concept of the library OS.

Virtualisation is the main driving force of cloud computing. We define virtualisation as the method of having a simulated hardware machine appear in place of the real one [49]. We call these simulated machines *virtual machines (VMs)*. VMs contain the software that is included in a physical machine and we can have multiple copies of the software running on top of the hardware layer. VMs offers a number of advantages that make them an integral part of cloud computing such as on demand scaling of VMs, complete life cycle of VMs (start, stop, sleep, running), efficient usage of hardware resources, easy migration of software to other physical hardware, and isolation of resources.

The simulator software is called a *hypervisor* or a *Virtual Machine Monitor*. Hypervisors provide an abstraction layer to hardware or software resources that exist on the lower level. The main goal of hypervisors is to manage, regulate and partition hardware resources among VMs that exist on the same physical host and in other physical hosts that are part of the virtualised infrastructure. Resources are hard disks, processors, memory, network devices and other required devices such as USB. Hypervisors maintain isolation between VMs, according to the requirements of a cloud tenant. The hypervisor needs to accommodate different policies and configurations to meet the requirements of cloud providers and tenants. For instance, a tenant can specify a security policy whereby two VMs must be isolated and must not communicate with each other. Another requirement would be that two VMs can communicate if they are in the same physical host or in the same coalition of VMs. Thus, depending on the requirements of cloud providers and tenants, both strong isolation and secure sharing among VMs are needed.

There are two main approaches in designing hypervisors which are illustrated in Figure 2.4. The first approach uses *Type I hypervisors* that are running on top of the real host's hardware, as depicted in Figure 2.4a. These hypervisors are also called native or bare-metal hypervisors. Examples of this type of hypervisor include Hyper-V, Xen, and VMware ESX. The second approach uses, *Type II hypervisors* that run on top of a typical OS such as Windows or Linux. Figure 2.4a illustrates the architecture of this hypervisor type. The Kernel-based Virtual Machine (KVM) is a popular example of a Type II hypervisor currently used in cloud infrastructures.

Currently, the most common platform for virtualisation used in cloud infrastructure is x86. However, there is growing interest in providing virtualisation solutions using ARM's virtualisation extensions [132], which supports embedded devices. In this thesis, we are concerned only with virtualisation related to the x86 platform.

There are three methods of virtualisation for the x86 platform: *full virtualisation*, *paravirtualisation*, and *hardware-assisted virtualisation*. We provide a brief introduction to each of these virtualisation methods in the following paragraphs.

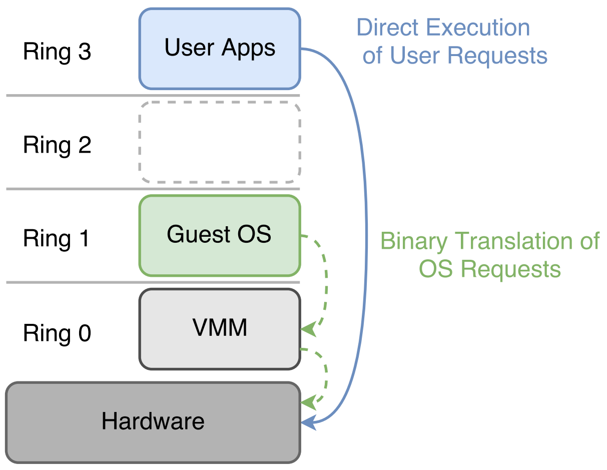

Ring 3
User Apps
Direct Execution of User Requests
Ring 2
Ring 1
Guest OS
Binary Translation of OS Requests
Ring 0
VMM
Hardware

Figure 2.5: Full virtualisation - binary translation

2.5.1 Full Virtualisation

VMWare was the first company to introduce full virtualisation through binary translation for the x86 platform in 1998 [134]. VMWare found a way to virtualise the x86 platform using a combination of binary translation and direct execution on the CPU. Using this approach, multiple guest OSes were able to execute in full isolation on the same hardware.

Figure 2.5 illustrates the binary translation of sensitive x86 instructions and the direct execution of user applications; i.e. non-privileged user-level instructions. The kernel code of the guest OS contains non-virtualisable x86 instructions. For example, the binary translation procedure consists of translating the kernel code to replace the non-virtualisable x86 instructions with the new instructions code, from which it can generate the desired requests to the virtual hardware. User applications have permission to execute their non-privileged user-level instructions directly on the physical hardware to assure high throughput performance.

The combination of binary translation and direct execution guarantees full virtualisation for the guest OSes running on virtual machines. Hence, the OS does not require any modifications and is not aware that it is executing on top of a virtualisation layer. The hypervisor is responsible for providing the guest OS of its VM with the physical hardware resources it needs, such as a virtual Basic Input/Output System (BIOS), virtual devices, and virtual memory management.

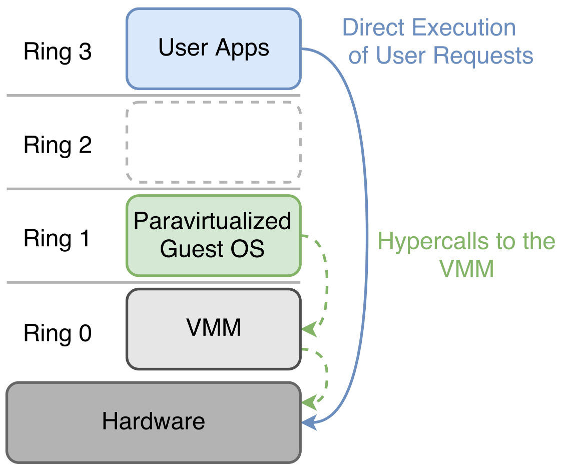


Figure 2.6: Paravirtualisation

2.5.2 Paravirtualisation

Paravirtualisation introduces a new technique to handle the problem of the non-virtualisable privileged x86 instructions. This method relies on the co-design of virtual machine monitor and OS. The former offers an interface slightly different from the physical machine. The latter is modified to interact with the virtual machine monitor so that the set of privileged x86 instructions can be virtualised.

Figure 2.6 illustrates the handling of instructions in a paravirtualisation environment. *User applications* continue to execute their non-privileged instructions set natively for high performance, while the privileged OS-level instructions from the *paravirtualised guest OS* generate an *hypercall*. Hypercalls are similar to the system calls user applications use in non-virtualised environments to request privileged operations from the kernel. In a hypercall, however, it is the OS requesting a privileged operation to the hypervisor.

The paravirtualisation approach enhances performance, scalability, and simplicity [136]. Another advantage is that it does not require hardware-assisted virtualisation technology (discussed below). However, it also has its own disadvantages. For example, a guest OS can only execute in a paravirtualised environment if its kernel is modified so that it uses hypercalls to interact with the hypervisor. These changes increase maintenance costs and prevent off-the-shelf OSes from executing in paravirtualised platforms.

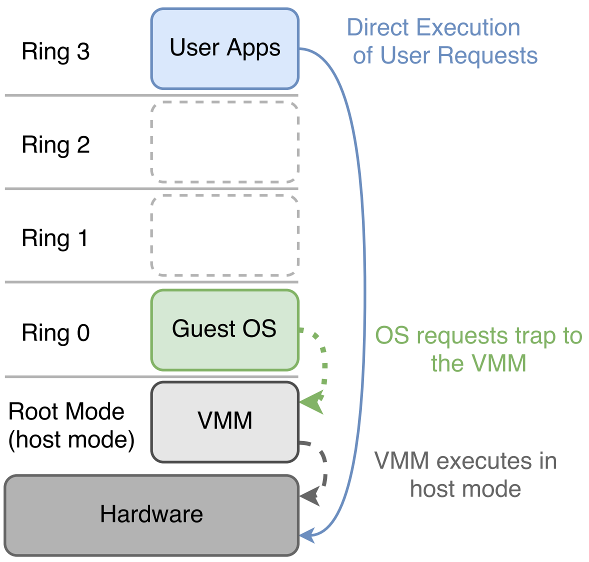

Ring 3
User Apps
Direct Execution of User Requests
Ring 2
Ring 1
Ring 0
Guest OS
OS requests trap to the VMM
Root Mode (host mode)
VMM
VMM executes in host mode
Hardware

Figure 2.7: Hardware-Assisted Virtualisation

2.5.3 Hardware-Assisted Virtualisation

Binary translation and paravirtualisation were envisioned to solve the problem of virtualising privileged x86 instructions due to rigid architecture requirements. The x86 architecture provides four levels of privilege through rings 0, 1, 2, and 3. The problem arises from moving the OS away from ring 0 so that the VMM can take its place as the software with highest privilege execution level. Since the architecture was designed for a single OS, when this change happens the privileged OS instructions do not behave as designed because they are no longer executed in ring 0. Therefore, an architecture redesign considering virtualisation requirements would improve its support.

Hardware-assisted virtualisation involves the redesign of the x86 architecture to better accommodate the requirements of virtual machine systems. Figure 2.7 shows the existence of hardware support for a new root privilege level (root mode) where the hypervisor can execute at a higher privilege level than ring 0. In this architecture, when a hypervisor is present, privileged x86 instructions from the guest OS are set to automatically transfer platform control to the hypervisor. This feature removes the need for binary translation or paravirtualisation.

Examples of hardware-based virtualisation are Intel's virtualisation technology (Intel VT) and AMD's virtualisation technology (AMD-V).

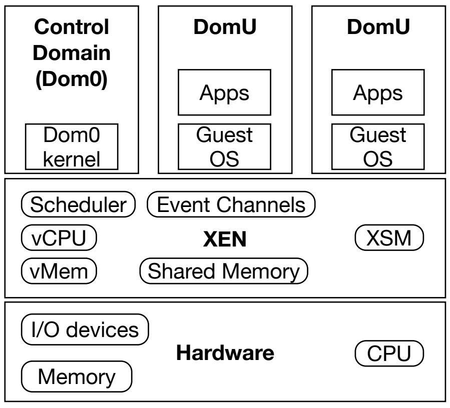


Figure 2.8: Xen Architecture

The set of features includes fast transfer of platform control from the guest OSes to the hypervisor, and the ability to uniquely assign physical Input/Output devices to a particular guest OS.

In the following subsection we present the Xen hypervisor and KVM virtualisation solutions that are currently used by many cloud providers such as Google and Amazon.

2.5.4 Xen hypervisor

The Xen hypervisor is an open-source bare metal hypervisor. As we have seen in section 2.5, a bare metal hypervisor is a software layer between the hardware and VMs. The Xen hypervisor makes it possible to execute different OSes on a single host. Its main features are that it has a small footprint and interface, uses many different OSes, driver isolation with device driver domains, and paravirtualisation [144]. In addition, Xen is used in industry to power cloud infrastructures such as Amazon's Elastic Compute Cloud (Amazon EC2) [2].

Figure 2.8 depicts the Xen architecture. VMs in the Xen hypervisor are called *guest domains* or *user domains (DomUs)*. There are two main types of guest domain that the Xen hypervisor can run: *fully virtualised (HVM) guests*; and *paravirtualized (PV) guests*. *Domain 0* (Dom0) is the first VM that is booted after the hypervisor and contains hardware drivers and management interfaces to control VMs. The management interfaces, called *toolstacks*, provide different front-ends that manage the guest domains.

The Xen hypervisor multiplexes resources such as event channels, virtual memory, and virtual CPU. Of these virtual resources only virtual memory and virtual CPU cannot be shared between two VMs. All the other resources such as network and block storage can be shared. For Input/Output (I/O) and network devices Xen architecture provides a split-device model. Native device drivers are a part of Dom0 kernel and sharing them between VMs is accomplished using *device channels* that are implemented using shared memory. Dom0 makes the native device driver available to other VMs using a *back-end device driver* and any VM that wants to communicate with the back-end device driver exports a virtual device driver called the *front-end device driver*. In order for the front-end device driver to become active, it has to be connected to the corresponding back-end device driver. Therefore, many front-end virtual devices are mapped to only one back-end virtual device. For instance, two VMs share the same physical network adapter that is owned by Dom0 or a device driver domain. A virtual network driver that exists in each VM enables network sharing for these two VMs. The Xen hypervisor uses event channels and shared memory to communicate with the physical network device.

2.5.4.1 Xen Security Modules (XSM)

The Xen Security Modules (XSM) [31, 143] is a generalised security framework that enables an administrator to control a guest domain and its behaviour. This can be achieved by defining the set of permissible interactions between guest domains, the Xen hypervisor, and resources such as memory and I/O devices.

The main mechanism that the XSM uses is a set of *security check functions* also known as *hooks*. The XSM checks if a subject can perform operations, such as sharing a grant table on an object. In this type of operation, VMs can be either a subject or an object. These operations are called *security-sensitive operations*. An example operation is *evtchn_send* in *event_channel.c* and its security check function *xsm_evtchn_send* which aims to determine whether the subject is authorised to use the event channel. In order to grant access to a security-sensitive operation in Xen, the security hook must be added before the operation.

Each security hook is a function pointer in a structure, *xsm_operations*, that resides in *xen/include/xsm/xsm.h*. This *xsm_operations* structure is initialised to a set of hooks from a dummy security module. In addition, the XSM adds a function to register a security module and another function to use the default security module. More precisely, a *register_xsm* function exists in */xen/xsm/xsm_core.c* that allows a security module to set *xsm_operations* structure to refer to its own hook functions. An *unregister_xsm* function is used to revert *xsm_operations* to use the dummy module hooks.

One feature of the XSM is that we can add security modules that provide different access control mechanisms. However, only one security module can be active at a time, since it has to be compiled with the Xen hypervisor. There are three security modules that have been included in the XSM, Flux Advanced Security Kernel (Flask) [121], sHype [103], and dummy. Since Xen 4.2, only the Flask security module is used. sHype is not included, as it is no longer maintained. An introduction to XSM-Flask follows.

2.5.4.2 XSM-Flask

XSM-Flask is a security module that can be enabled at the compilation time of the Xen hypervisor. It is a *Mandatory Access Control (MAC)* mechanism built upon Flask security architecture that combines Role-based Access Control [42], Type Enforcement [21] and Multi-Level Security (MLS) [15]. Administrators can impose security policies to regulate access to resources. The security policies define a security context that contains a user identity, a role, a type, or an optional MLS level. By convention, each element of the security context, apart from an MLS level, ends with an appropriate suffix. For instance, user identity ends with '_u', a role ends with '_r', a type ends with '_t'. Therefore, a security context is represented as variable-length strings, as follows:

```
user_u:role_r:type_t[:range]
```

A subject in the XSM-Flask is an active process (e.g. a VM) that causes information to flow between objects or state transitions and has a security context associated with it. An object in the XSM-Flask is a resource, such as files, sockets, network interfaces, or memory, that can be accessed by subjects. Objects are assigned a security context. Each object comprises a class identifier and a set of permissions that define the interactions the object can provide (read, write, send etc..). When an object is created, it is allocated a name and a security context. In order to compute a security context for

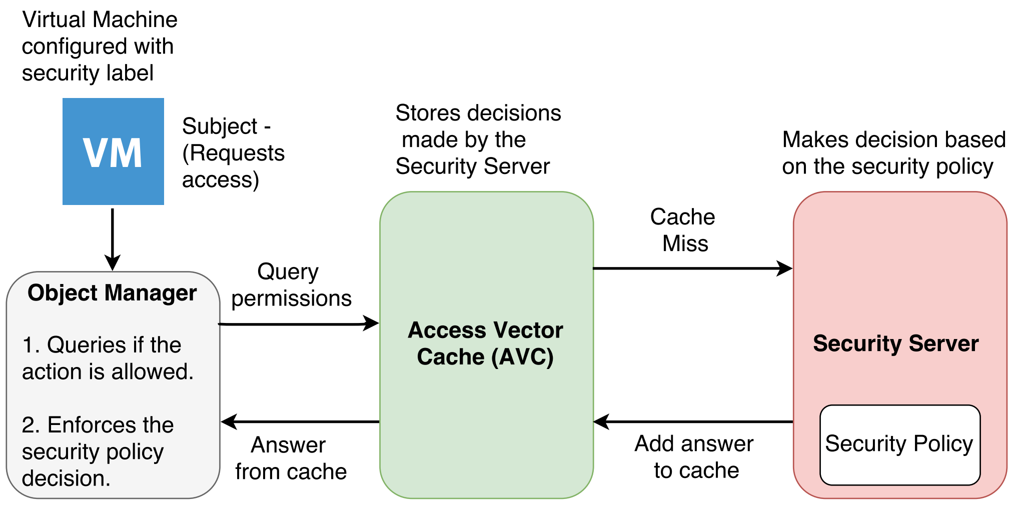

Virtual Machine configured with security label
VM
Subject - (Requests access)
Stores decisions made by the Security Server
Makes decision based on the security policy
Cache Miss
Query permissions
Object Manager
1. Queries if the action is allowed.
2. Enforces the security policy decision.
Access Vector Cache (AVC)
Security Server
Answer from cache
Add answer to cache
Security Policy

Figure 2.9: XSM-Flask Decision Making Control Flow.

an object, we need three components: a source context, a target context, and a target class.

When the XSM-Flask is enabled, VMs have to be created using a security label in their configuration file. The configuration file parameter is called *seclabel*. For instance, the default policy distributed with the Xen hypervisor uses *seclabel='system_u:system_r:domU_t'* for a typical VM. This means that the VM is part of the system user, has a system role, and is of type domU.

Policy Types A type can be used to specify which hypercalls the type is allowed to execute and the method it is allowed to use to execute them. Each type can be assigned to an attribute category, such as xen_type, domain_type, or resource_type. An example type declaration is the following:

type xen_t, xen_type, mls_priv;

This policy assigns the xen_t type the attributes xen_type and mls_priv.

In order to specify which hypercalls are allowed with each defined type, the *"allow"* word is used. For instance, the following policy:

allow dom0_t security_t:security check_context;

enables the dom0_t type to execute the check_context hypercall in the security class targeting a security_t type. In general the structure of this policy is as follows:

allow (source type) (target type) : (security class) (hypercall);

Policy Roles With policy roles we can specify sets of types that are part of a role. A role belongs to the security label. For instance, the following policy specifies the system role.

```
role system_r;
role system_r types { xen_type domain_type};
```

The first line defines a new role and the second line specifies the types that belong to that role using attributes

Policy Users The default policy creates three users: system_u, customer_1, and customer_2. A user can be assigned multiple roles. However, it is convention to assign the system_u user to the system_r role and the customer_* users are assigned to the vm_r role. For instance, we can create a new user customer 3, as follows:

user customer_3 roles {vm_r};

Policy Constraints Constraints are used to disallow undesired operation that the policy may allow in certain situations. The sample policy specifies two constraints to prevent event channels and grants between two customers. An example constraint follows:

```
constrain grant { map_read map_write copy} (
   u1 == system_u or
   u2 == system_u or
   u1 == u2
);
```

The above constraint specifies that the hypercalls that are part of the security class are only allowed when the expression is true. The expression can contain any user, role, and type that is defined in the security policy.

Security Classes Security classes are defined in *xen/xsm/flask/policy/ access_vectors*. They categorise each hypercall into one of the security classes. Each security class can have up to 32 members. Example classes include the xen class, domain and domain2 classes, hvm class, event class, grant class, mmu class, shadow class, resource class, and security class.

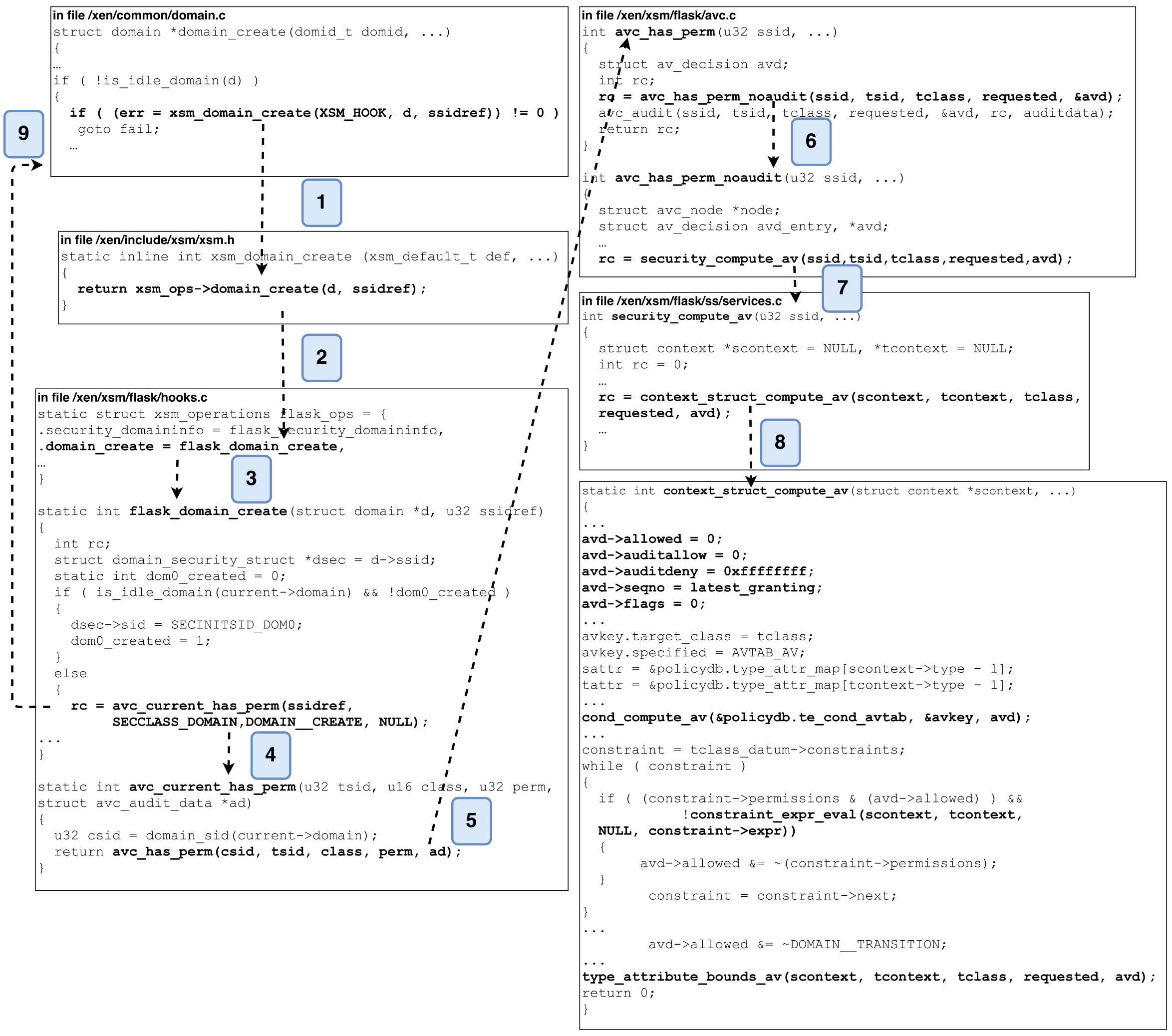
Using XSM, the Xen hypervisor uses a *mandatory access control* mechanism built upon Flask security architecture. The main goal of XSM-Flask is to separate security enforcement from security policy and isolate components of security systems. We can define permissible interaction between domains, hypervisor, and resources using XSM-Flask policies.

The XSM-Flask has to be compiled with Xen and an XSM-Flask policy must be specified. In addition, the Dom0 boot-loader must be adjusted to enable XSM-Flask at boot time and load the authored security policy. XSM-Flask can be configured at boot time to be in a permissive, enforcing, late, or disabled state.

If the XSM-Flask is in a permissive mode then the policy provided by the boot loader will be loaded, any errors will be reported to the shared memory and the booting of guest domains will proceed. In the enforcing mode, the XSM-Flask will report any errors to the shared memory and enforce the security policies that have been provided to the boot loader. If a security policy is not loaded then the Xen hypervisor will not continue booting. During default mode any loading of security policies to the boot-loader is disabled. The XSM-Flask is enabled and it will start enforcing access control when a security policy is loaded using `xl loadpolicy`. The disabled mode of the XSM-Flask uses the dummy module that enables the same security policy as if the Xen hypervisor is compiled without any support for the XSM-Flask.

The main entities of the XSM-Flask are subjects and objects. Subjects can be processes in the system or VMs. Objects can be files, socket, ports, devices, I/O, and resources. Subjects with permissions can interact with objects via a security policy. Objects controlled by a security policy are labelled with a set of attributes that we call that security context. Another term that the XSM-Flask uses is the target class, which means that a subject and an object belong to a class. It has a source context and a target context. The decision to allow access to a resource, or not, is based on security labels rather than users. We can define security attributes for entities using labels. Permissions for Subject interacts with Object via a security policy.

XSM-Flask architecture The architecture of XSM-Flask consists of four elements: the security hooks, the object manager, the security server and the Access Vector Cache.



in file /xen/common/domain.c
```
struct domain *domain_create(domid_t domid, ...)
{
…
if ( !is_idle_domain(d) )
{
  if ( (err = xsm_domain_create(XSM_HOOK, d, ssidref)) != 0 )
   goto fail;
  …
```

in file /xen/include/xsm/xsm.h
```
static inline int xsm_domain_create (xsm_default_t def, ...)
{
  return xsm_ops->domain_create(d, ssidref);
}
```

in file /xen/xsm/flask/hooks.c
```
static struct xsm_operations flask_ops = {
.security_domaininfo = flask_security_domaininfo,
.domain_create = flask_domain_create,
…
}

static int flask_domain_create(struct domain *d, u32 ssidref)
{
  int rc;
  struct domain_security_struct *dsec = d->ssid;
  static int dom0_created = 0;
  if ( is_idle_domain(current->domain) && !dom0_created )
  {
    dsec->sid = SECINITSID_DOM0;
    dom0_created = 1;
  }
  else
  {
    rc = avc_current_has_perm(ssidref,
         SECCLASS_DOMAIN,DOMAIN__CREATE, NULL);
...
}

static int avc_current_has_perm(u32 tsid, u16 class, u32 perm,
struct avc_audit_data *ad)
{
  u32 csid = domain_sid(current->domain);
  return avc_has_perm(csid, tsid, class, perm, ad);
}
```

in file /xen/xsm/flask/avc.c
```
int avc_has_perm(u32 ssid, ...)
{
  struct av_decision avd;
  int rc;
  rc = avc_has_perm_noaudit(ssid, tsid, tclass, requested, &avd);
  avc_audit(ssid, tsid, tclass, requested, &avd, rc, auditdata);
  return rc;
}

int avc_has_perm_noaudit(u32 ssid, ...)
{
  struct avc_node *node;
  struct av_decision avd_entry, *avd;
  …
  rc = security_compute_av(ssid,tsid,tclass,requested,avd);
```

in file /xen/xsm/flask/ss/services.c
```
int security_compute_av(u32 ssid, ...)
{
  struct context *scontext = NULL, *tcontext = NULL;
  int rc = 0;
  …
  rc = context_struct_compute_av(scontext, tcontext, tclass,
  requested, avd);
  …
}
```

```
static int context_struct_compute_av(struct context *scontext, ...)
{
...
avd->allowed = 0;
avd->auditallow = 0;
avd->auditdeny = 0xffffffff;
avd->seqno = latest_granting;
avd->flags = 0;
...
avkey.target_class = tclass;
avkey.specified = AVTAB_AV;
sattr = &policydb.type_attr_map[scontext->type - 1];
tattr = &policydb.type_attr_map[tcontext->type - 1];
...
cond_compute_av(&policydb.te_cond_avtab, &avkey, avd);
...
constraint = tclass_datum->constraints;
while ( constraint )
{
  if ( (constraint->permissions & (avd->allowed) ) &&
            !constraint_expr_eval(scontext, tcontext,
  NULL, constraint->expr))
  {
       avd->allowed &= ~(constraint->permissions);
  }
        constraint = constraint->next;
}
...
        avd->allowed &= ~DOMAIN__TRANSITION;
...
type_attribute_bounds_av(scontext, tcontext, tclass, requested, avd);
return 0;
}
```

Figure 2.10: Call walkthrough for the hypercall *domain_create*.

Security Hooks Security hooks act as policy enforcement points that mediate and protect access to virtual resources for VMs. For instance, there is a security hook attached to the *create_domain* hypercall, which is called by the xl application to create a new guest domain. Figure 2.10 illustrates the call walk-through for the *create_domain* hypercall. The main steps to check if the creation of a new VM is allowed are the following:

1. The file */xen/common/domain.c* has a hook that links to the XSM function *xsm_domain_create*, which is called to check access permissions.

2. The *xsm_ops* table points to `domain_create` function for the XSM.
3. The *flask_ops* table has been set to point to the *flask_domain_create()* function in *hooks.c* since the Flask is initialised as the security module.
4. The *flask_domain_create()* function checks if the domain can be created. This function calls *avc_current_has_perm* to perform the access check.
5. This results to a call to the AVC via the *avc_has_perm()* function in *avc.c* file, which checks if the permission has been granted before or not.
6. First, using *avc_has_perm_noaudit()*, the cache is searched for an entry.
7. If an entry has not been found then the *security_compute_av()* function in */xen/xsm/flask/ss/services.c* is called. The *security_compute_av()* searches the Security Identification (SID) table for the source and target entries. If the entries are found then the *context_struct_compute_av()* function is called.
8. The *context_struct_compute_av()* executes a number of checks to evaluate if access will be granted. These checks are the following:
   (a) Initialise the AV to default values.
   (b) Check if there are any type enforcement rules.
   (c) Check if any conditional statements are used via the `cond_compute_av` function.
   (d) Remove any permissions that are defined in constraints via the `consttraint_expr_eval()` function. Also check any MLS constraints.
   (e) Check if a process transition is being requested or not. If it is, then DOMAIN_TRANSITION is checked alongside a current role, new role pair.
   (f) Finally, check if there are any constraints applied via the `type_attribute_bounds_avc()` function.
9. When the result is computed then it is returned to the *xsm_domain_create()* function. In our case the domain creation is allowed.

2.5.4.3 Libvchan

In this subsection, we discuss the libvchan [140] library that enables VMs to create inter-domain communication channels. VMs running on the same hypervisor can communicate with each other using shared memory. Xen hypervisor supports the grant table mechanism, which is a handle table for each VM. Each of the entries in the table points to a memory page that can be shared between VMs. Another mechanism that Xen hypervisor provides is the event channel, which is an asynchronous notification mechanism. Both of these mechanisms are used in the libvchan interface to provide shared memory communication between two parties, the server VM and the client VM. We now outline the main steps required to establish communication between the client and server parties:

1. The server boots and registers the event channel and grant references in xenstore.
2. The server offers three pieces of memory: the first to store the data in transit coming from the client, the second to store data from the server to the client side, and the third to hold information regarding the state of the communication.
3. The server instructs the Xen hypervisor to give access to the allocated memory to the client.
4. The client requests to map the memory into its own virtual address space. At this moment both the server and client can access the mapped memory simultaneously.
5. Each side can send data to the other by writing to the correct ring buffer and then updating the counter in the shared memory.
6. If one side is waiting, then the other side can signal it by issuing an event channel.

In addition, libvchan uses a producer-consumer shared ring. When two VMs want to exchange data using shared memory, they create a circular buffer data structure, which includes the read and write indexes. Only one end of the connection is responsible for the write and only the other end the read operation. Each time the sender VM wants to send data to the other end, it copies them in the shared memory and updates the producer index to the current write record. Using the same approach, the consumer

VM receives the data from the shared memory and updates the producer index with the current read record. The two VMs can control if there are new data to read or if there is space for a new record, comparing the producer and consumer indexes. Since the data storage is limited, the indexes are placed at the initial position.

During the establishment of the communication between the two VMs there are a number of hypercalls used to manage the grant table entries and the lifecycle of the event channel mechanism. The main hypercalls that libvchan uses are the following:

- `HYPERVISOR_grant_table_op(GNTTABOP map grant ref,...)`:
  Used for mapping pages.
- `HYPERVISOR_grant_table_op(GNTTABOP unmap grant ref,...)`:
  Unmaps pages.
- `HYPERVISOR_event_channel_op(EVTCHNOP alloc unbound,...)`:
  Creates a new event channel port.
- `HYPERVISOR_event_channel_op(EVTCHNOP bind interdomain,...)`:
  Connects with remote event channel.
- `HYPERVISOR_event_channel_op(EVTCHNOP send, &send)`:
  Sends a notification on an event channel, which is attached to a port.
- `HYPERVISOR_event_channel_op(EVTCHNOP close,...)`:
  Closes event channel.

2.5.5 KVM

A Kernel-based Virtual Machine is a Linux kernel-based virtualisation technology [66]. It supports full virtualisation on x86 hardware containing virtualisation extensions such as Intel VT or AMD-V. The main component of the KVM is a loadable kernel module, called *kvm.ko* and a processor-specific module, called *kvm-intel.ko* or *kvm-amd.ko*. If the Intel VT is supported, then the vmx flag is present and the KVM loads the *kvm-intel.ko* module. The same approach exists when the hardware supports the AMD SVM virtualisation extension and the svm flag is present. This means that the *kvm-amd.ko* module is loaded. Figure 2.11 depicts the high-level architecture of KVM.

The device file */dev/kvm* is the main interface that the KVM exposes to application to use the ioctls() function. There are three types of ioctl in the KVM API, which are used to manage various aspects of VMs. The first type is the system ioctl, which queries and sets global attributes for the whole KVM system and creates VMs. The second type is the VM ioctl, which queries and sets attributes for a whole VM. This type is used to instantiate vCPUs and run VM ioctl from an identical address space that was utilised to create the VM. The third type is cpu ioctl, which queries and sets attributes that manage the execution of a single vCPU. The vcpu ioctl can be executed from the same thread with that used to create the vCPU.

The KVM can execute numerous VMs that run unmodified Linux or Windows images. Every virtual machine includes virtualised hardware such as a network card, a disk, or a graphics adapter.

KVM is open-source software and KVM's loadable kernel module has been included in the Linux kernel since version 2.6.20. The user space component of the KVM has been included in the Quick Emulator, since version 1.3.

Each virtual machine in the KVM is a Linux process and it interacts with the *kvm.ko* kernel module. All hardware access for the virtual machine is managed by the kernel module using a character device called */dev/kvm*, and by the QEMU process, which is modified to support KVM. All code handling and exception handling is delegated to one kernel module. In the Linux 2.6.33 kernel, the KVM device driver for Intel VT–x can be measured to be around 4,000 lines of C code. However, the KVM also has other components such as an emulator, interrupt controller, memory management, and page table management, which increase the overall size of the KVM. It should be noted that these features are necessary once real applications are built on a hypervisor.

However, for obtaining a code block without vulnerabilities, these features can be eliminated and added later when required by each application. This also gives an application the freedom to choose how to implement the above features. As the KVM is coupled with the Linux kernel, the security features provided by the KVM are affected by the security features a standard Linux kernel provides. For instance, sVirt can be used to further secure access to resources. Another option would be to properly configure SELinux to protect the KVM kernel module.

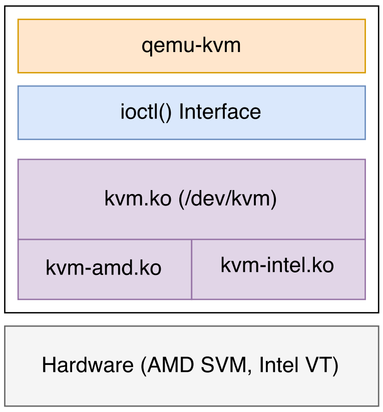


Figure 2.11: KVM Architecture

2.6 Trusted Execution Environment

This section discusses the Trusted Execution Environment (TEE) reference architecture proposed by the standards development organisation called GlobalPlatform and three other system architectures that create TEEs. First, we discuss the Trusted Platform Module (TPM) and how it can used to create TEEs. Second, we consider the ARM TrustZone and its architecture. Third, we discuss the Intel SGX solution and its architecture.

A TEE Trusted Execution Environment (TEE) [48] is defined as a system security primitive that enforces isolation using hardware to execute sensitive application logic. Figure 2.12 depicts the *TEE System Architecture* [48] proposed by GlobalPlatform. This architecture contains two main execution environments: first, the Rich Execution Environment (REE) includes most of the platform software including the OS; second, the Trusted Execution Environment that isolates hardware, and software resources from the REE. Each execution environment contains a set of applications according to the goals of the execution environment. There are general purposed REE applications, and *client applications* (CAs) that communicate with applications in the TEE called *trusted applications* (TAs). The aim of the TEE is to ensure the confidentiality and integrity of the TA's execution state and data from other software that executes on the platform.

The GlobalPlatform organisation has described four standards for TEE architec-

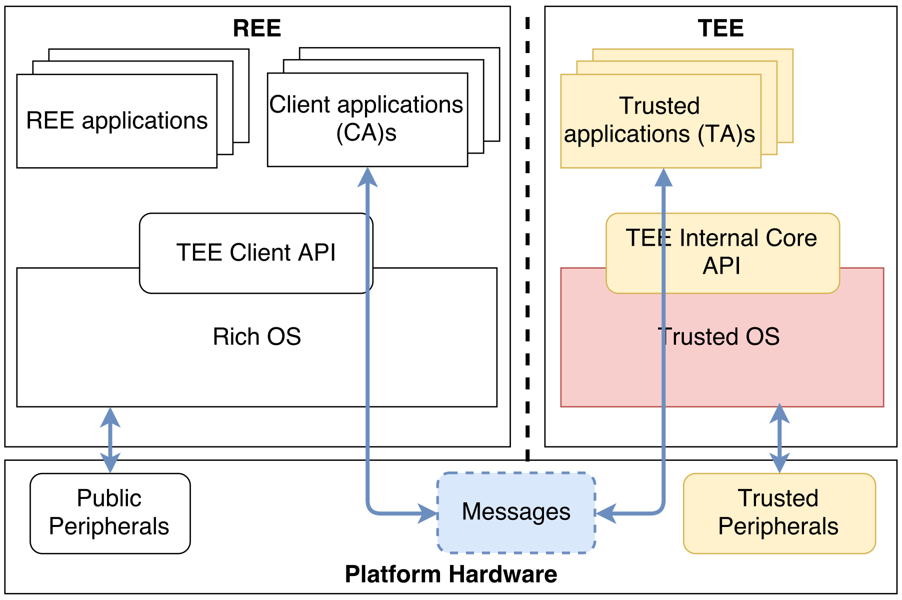


Figure 2.12: GlobalPlatform TEE System Architecture [48]

ture, interfaces, and management. First, the *TEE System Architecture* [48] illustrates the software and hardware components that form the TEE. Second, the *TEE Internal Core API* [46] describes the programming interfaces that are inside the TEE. These interfaces include memory management, secure storage, and cryptographic primitives interfaces for the development of TA. Third, the *TEE Client API* [45] defines the interfaces for the OS driver that initialises a TA and provides access for a CA to create a new session with a TA to exchange data. Fourth, the *TEE Management Framework* [47] presents a framework for the administration of TEEs, the administration of TAs, and how the life cycle of TAs is managed by the different stakeholders.

Current TEE platforms include security primitives that either create a TEE using only the CPU, such as the ARM TrustZone and Intel SGX, or require an additional hardware component, such as the Trusted Platform Module (TPM). These TEE platforms are described in the following subsections.

2.6.1 Trusted Platform Module

The Trusted Computing Group (TCG) is the organisation that provides the TPM [129, 128] standard specifications. Currently, there are two versions of TPM specifications

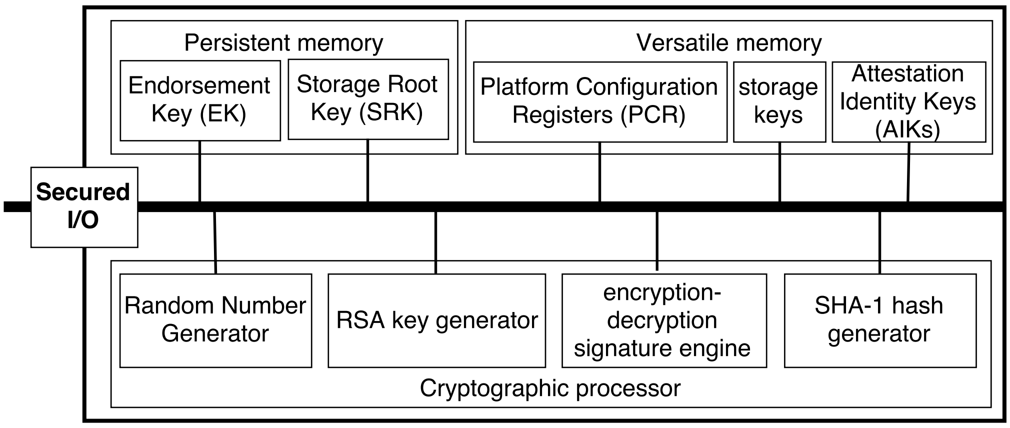


Figure 2.13: Trusted Platform Module (TPM) 1.2 architecture [129]

the main specification for TPM 1.2, and the family of TPM 2.0 library specification. These specifications define the TPM as a micro controller that is capable of secure storage of keys, passwords, and digital certificates.

TPMs do not allow application-specific logic to run inside a TPM, in contrast to other TEE platforms. TPMs specify interfaces that enable the OS, and user applications to use the TPM functionality, and create trustworthy systems. The TPM binds the device platform and it can be used to identify the specific device. Another feature of the TPM is to allow the reporting of its platform configuration to another party, which enables the attestation to the device.

The TPM includes tamper protection measures, which can make it a *root of trust*. This means that the user implicitly trusts the hardware, or software mechanism. There are three roots of trust, a *Root of Trust for Measurement*, a *Root of Trust for Storage*, and a *Root of Trust for Reporting*.

The RTM includes an implementation of a secure hash function, which can be inside the TPM or not. This trusted hash function enables integrity measurements that demonstrate the trustworthiness of the system. The integrity measurements are the collection of the result hash process on software, or data. The RTS is the set of TPM components that are storing the hash results from the integrity measurements. The reporting of the state of the system in a verifiable manner is the goal of the RTR.

The above roots of trust are the basic items that maintain the trustworthiness, or integrity of a system that uses TPMs. The main components of a TPM are depicted in

Figure 2.13. In the following paragraphs, we briefly describe the main components of the TPM.

TPMs include a number of *Platform Configuration Registers (PCRs)* that persist measurements of events that are related to the security state of the system. There is also a non-volatile storage for securing data from exterior entities. The *Endorsement Key (EK)* is a permanent RSA key pair, which is fused on the chip when it is built. The EK can identify and authenticate a TPM. A certificate for EK is also added, which is signed by a Certificate Authority (CA).

The *Storage Root Key (SRK)* is created when a user takes an ownership of the TPM, and provides the root of a key hierarchy in the TPM. The *Attestation Identity Key (AIK)* is a key pair that the TPM generates, and is used for attestation. Using the AIK a remote entity can verify that is communicating with a TPM without disclosing the identity of the TPM. The TPM 1.2 can use Direct Anonymous Attestation (DAA) [24] to create anonymous signatures.

TPMs can also bind encrypted data locally to the TPM by making use of the SRK. The only way for the encrypted data to be decrypted is by using the TPM. Sealing encrypts data that can only be decrypted, if the system is in the same state. This means that the data can be decrypted on the same computer, only if it was encrypted and running the same software.

The main focus of the TPM is to protect from software attacks. However, attacks have been demonstrated in the past. For instance, The Time of Check/Time of Use (TOCTOU) vulnerability in the Remote Attestation model of the TCG architecture is exploited, and a TPM reset attack are demonstrated in [120]. A number of successful hardware-based attacks have been demonstrated against BitLocker boot process that uses the TPM sealing process. These attacks use the Evil Maid scenario where an attacker has physical access to the victim's computer, and adds a compromised bootloader [131].

The TPM can create a TEE using the Static Root of Trust Measurement (SRTM) where the hash of all the software that is loaded since BIOS is measured and the OS is responsible for providing isolation. This approach was used by Microsoft in the Next-Generation Secure Computing Base(NGSCB) [81]. However, the problem with this approach is that the TCB becomes too large.

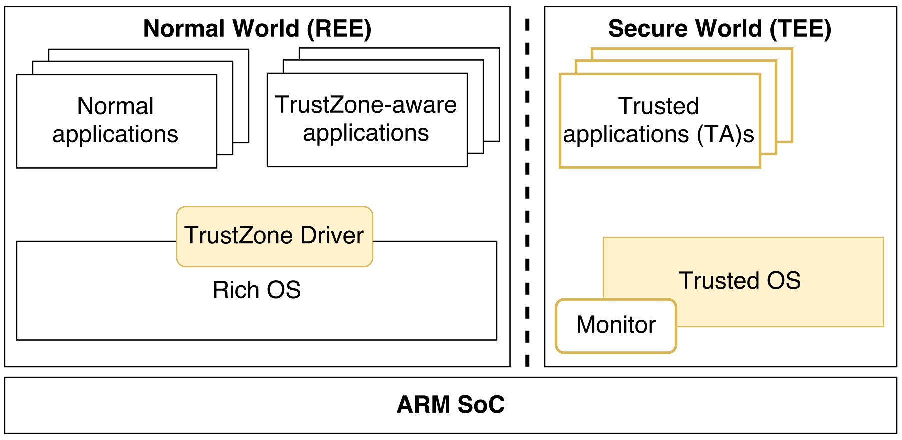


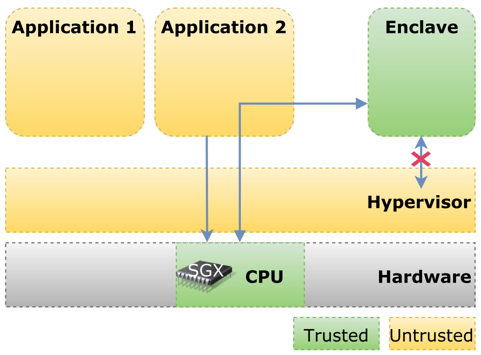
Figure 2.14: ARM TrustZone TEE [7]

2.6.2 ARM TrustZone

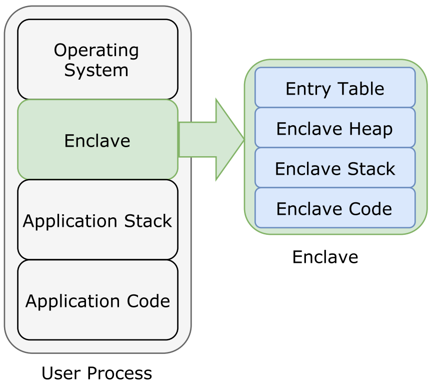
ARM TrustZone [7] is an architectural feature, which enables a system-on-chip (SoC) to run two semi-independent software stacks. This is achieved by dividing the CPU into two virtual processors. The first is called *Normal World (NWd)*, and the second *Secure World (SWd)*. They are implemented as separated operation modes in the CPU. Every one of the worlds, includes its own hardware, and software resources. The NWd is the REE that runs standard OSes, such as Linux, Android, and user applications. Figure 2.14 illustrates the main components of the ARM TrustZone architecture.

The boot loader from the SoC ROM is first executed and then the *trusted OS* in the SWd is initialised during the boot sequence. The next step involves the booting of the NWd OS. Secure booting is also supported, where the signature of each booting stage is verified previous to booting into the secure world. Thus, it establishes a chain of trust that prevents any malicious software to be executed in the secure world.

The *monitor mode* is another CPU monitor mode that regulates any entry to the SWd from the NWd. The monitor mode executes a specialised software called the *Monitor*, which aims to service exceptions, interrupts, and transaction that require a change between wolds. The correct implementation of this small piece of software is crucial to the security of the architecture, since any errors in the monitor can compromise the separation of the worlds. The *Secure Monitor Call (SMC)* is a specific instruction that NWd executes and is handled by the monitor mode to switch to the



(a) Intel SGX architecture [101]



(b) Enclave stack [57]

Figure 2.15: The main trusted components of the Intel SGX architecture and the structure of the enclave stack in a user process.

SWd.

The ARM TrustZone specifies, which hardware resources are accessible in a particular world. The mechanism that it uses is an extra control signal, called the Non-Secure (NS) bit, which exists in the *Secure Configuration Register*. This control signal is used in every read/write transactions to the system bus master and memory devices. The NS bit specifies if the CPU is currently executing in the SWd, or NWd.

The main memory of each world is isolated from each other using a virtual-to-physical address translation table in the Memory Management Unit (MMU) that is specific to each world. The NS bit is used to specify the identity of each world's entry in the Translation Look-aside Buffer (TLB). The SoC internal RAM inside the SWd is secret and private, and NWd is not allowed to access the SWd memory. In addition, it prevents an application executing in the NWd to observe memory access patterns of the SWd.

The main disadvantage of using the ARM TrustZone in a cloud computing context is that it only allows one secure enclave. Regarding performance the TEE code in the ARM TrustZone runs on the CPU, however, the RAM used may be limited. The ARM TrustZone focuses on protection from software attacks only and not on hardware attacks.

Supervisor Instruction	**Description**
ENCLS[EADD]	Add a page
ENCLS[EBLOCK]	Block an EPC page
ENCLS[ECREATE]	Create an enclave
ENCLS[EDBGRD]	Read data by debugger
ENCLS[EDBGWR]	Write data by debugger
ENCLS[EEXTEND]	Extend EPC page measurement
ENCLS[EINIT]	Initialise an enclave
ENCLS[ELDB]	Load an EPC page as blocked
ENCLS[ELDU]	Load an EPC page as unblocked
ENCLS[EPA]	Add version array
ENCLS[EREMOVE]	Remove a page from EPC
ENCLS[ETRACK]	Activate EBLOCK checks
ENCLS[EWB]	Write back/invalidate an EPC page

Table 2.3: Supervisor SGX Instructions [57]

2.6.3 Intel SGX

This subsection introduces the Intel SGX Intel Software Guard Extensions (SGX) [33, 3], and discusses the main elements of its architecture.

2.6.3.1 Introduction

A Intel SGX Trusted Execution Environment (TEE) [48] is an isolated environment that executes software at a higher privilege level than the OS. Intel SGX is a TEE that trusts only the hardware where the code is executed and adds a set of extensions to the Intel architecture. The main aim of Intel SGX is to provide confidentiality and integrity guarantees to security sensitive applications that reside in a system in which all privileged software, such as the hypervisor, OS, kernel, and device drivers are potentially malicious.

SGX adds 18 instructions to the x86 architecture. From these 18 instructions, 13 are supervisor instructions and are displayed in Table 2.3. The rest of the instructions are user instructions and are outlined in Table 2.4.

User Instruction	**Description**
ENCLU[EENTER]	Enter an enclave
ENCLU[EEXIT]	Exit an enclave
ENCLU[EGETKEY]	Create a cryptographic key
ENCLU[EREPORT]	Create a cryptographic report
ENCLU[ERESUME]	Re-enter an enclave

Table 2.4: User SGX Instructions [57]

2.6.3.2 SGX Architecture

SGX uses a secure container called enclave, which includes both the code and data encrypted. The enclave is isolated from other enclaves and other untrusted software, and there is a software attestation mechanism that allows a remote party to authenticate the software inside the enclave. Figure 2.15b illustrates how an enclave is structured inside a user process.

SGX leverages trusted hardware to achieve its confidentiality and integrity guarantees. This means that software executing in an enclave can withstand software and certain hardware attacks. The main hardware components that SGX trusts are the CPU, the MMU, the Memory Encryption Engine (MEE), and a memory region called the Processor Reserved Memory (PRM). Figure 2.15a shows a high-level view of the SGX architecture, including the main components we trust. PRM is a contiguous memory space in DRAM that cannot be directly accessed by any other software, even if it is a privileged one. PRM holds a data structure called **Enclave Page Cache (EPC)**, which we describe in the next subsection.

2.6.3.3 SGX Data Structures

In this subsection, we describe the main data structures that Intel SGX uses to realise the security guarantees for the enclaves. Table 2.5 shows the SGX data structures alongside a brief description.

The first data structure we describe is the EPC. The EPC stores the enclave contents and a number of related data structures and includes a number of 4KB pages that can be used by multiple enclaves. The system software manages the EPC data structure and SGX uses specialised instructions to allocate unused pages to enclaves, or to free

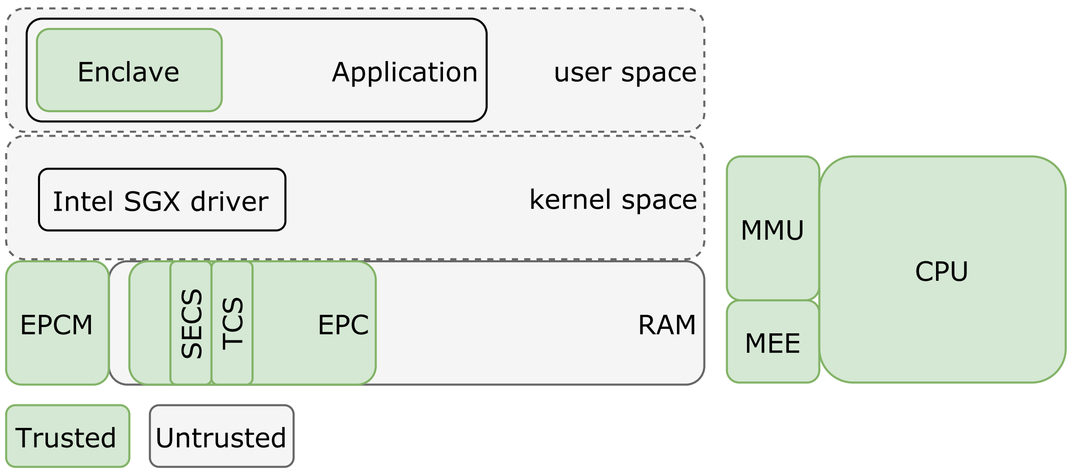


Figure 2.16: Enclave architecture [57]

any EPC pages that are no longer needed.

As the system software is not trusted, SGX uses the CPU to verify the correctness of the EPC allocations, and may refuse any EPC allocations that would compromise the SGX guarantees. In this situation, SGX uses an **Enclave Page Cache Map (EPCM)** to track the ownership of EPC pages.

Each entry in the EPCM array includes three fields: VALID, PT, and ENCLAVESECS. The first field contents are 1 when a corresponding EPC page has been allocated by an SGX instruction, or 0 if not. SGX will not allow the use of EPC pages that have already been allocated. The second field stores the page type of the corresponding EPCM entry. For instance, regular type (PT_REG) is used for EPC pages that store the enclave's code and data and PT_SECS is used when the EPC pages hold SGX Control Structures. The third field identifies the enclave that owns the EPC page.

SGX Enclave Control Structure (SECS) includes metadata for each enclave and is stored in a particular EPC page that has the page type set to PT_SECS. SGX instructions use the SECS to identify enclaves. Enclaves cannot access SECS pages.

SGX can execute concurrently the same enclave's code via different threads. The **Thread Control Structure (TCS)** is used for each logical processor that executes inside an enclave. The enclave author determines how many threads the enclave must support and provisions as many TCS instances. The TCS is stored in an EPC page

Data Structure	**Description**
SGX Enclave Control Structure (SECS)	Corresponds to one enclave.
Thread Control Structure (TCS)	Each executing thread relates to a TCS.
State Save Area (SSA)	The architectural state is saved in the thread's SSA, when an AEX occurs while executing in an enclave.
Page Information (PAGEINFO)	Architectural data structure that is used as a parameter to the EPC management instructions.
Security Information (SECINFO)	Holds metadata about an enclave page.
Paging Crypto MetaData (PCMD)	Keeps track of metadata related to a paged-out page.
Version Array (VA)	Stores version of evicted EPC pages.
Enclave Page Cache Map (EPCM)	CPU uses the EPCM to track the contents of the EPC.
Enclave Signature Structure (SIGSTRUCT)	Includes information about the enclave from the enclave signer.
EINT Token Structure (EINIT-TOKEN)	EINIT uses token to verify that the enclave has the right to launch.
Report (REPORT)	Output of the EREPORT instruction.
Report Target Info (TARGET-INFO)	Input parameter to the EREPORT instruction.
Key Request (KEYREQUEST)	Input to the EGETKEY instruction.

Table 2.5: SGX data structures [57]

with an entry field containing PT_TCS. The contents of the TCS cannot be accessed directly by the enclave code.

The **State Save Area (SSA)** is a secure placeholder to save execution context when a hardware exception is handled. Each TCS references a contiguous sequence of SSAs. The offset of the SSA array (OSSA) field points to the location of the first SSA in the virtual address space. The number of SSA (NSSA) fields shows the number of SSAs that are available.

2.6.3.4 Memory Encryption Engine

SGX uses DRAM to place the contents of an enclave. The memory layout of an enclave uses the **Enclave Linear Address Range (ELRANGE)** to designate a virtual address range. The virtual address area is used to map the code and the data stored in an enclave's EPC pages. All the address space inside ELRANGE is considered trustworthy and SGX guarantees confidentiality and integrity of the code and data residing in this address range. However, the address area outside ELRANGE is considered untrusted.

The main component that SGX utilises is the MEE [52, 53]. The MEE is designed to protect the confidentiality of the data written in the memory, protect data integrity, and prevent replay attacks. The design of the MEE relies on four elements to provide memory encryption to enclaves: an integrity tree, cryptographic primitives, the Message Authentication Code (MAC), and the mechanism to prevent replay attacks.

The CPU includes an internal cache that has a small amount of memory and is much faster than the system memory. The core of the CPU issues memory transaction during normal operation. The Memory Controller (MC) handles the transactions that miss the cache. The MEE acts as an extension to the MC and accepts all the cache-DRAM transactions that correspond to the protected data region. Another memory region that includes the integrity tree of the MEE is called the seized region. The combined memory areas are called the MEE region, which has a range of physical addresses that are fixed during boot time. The default memory size for the MEE region is 128MB.

Figure 2.17 illustrates how the MEE operates, which we describe in the following. All read/write requests to the protected region are sent to the MEE by the MC. Subse-

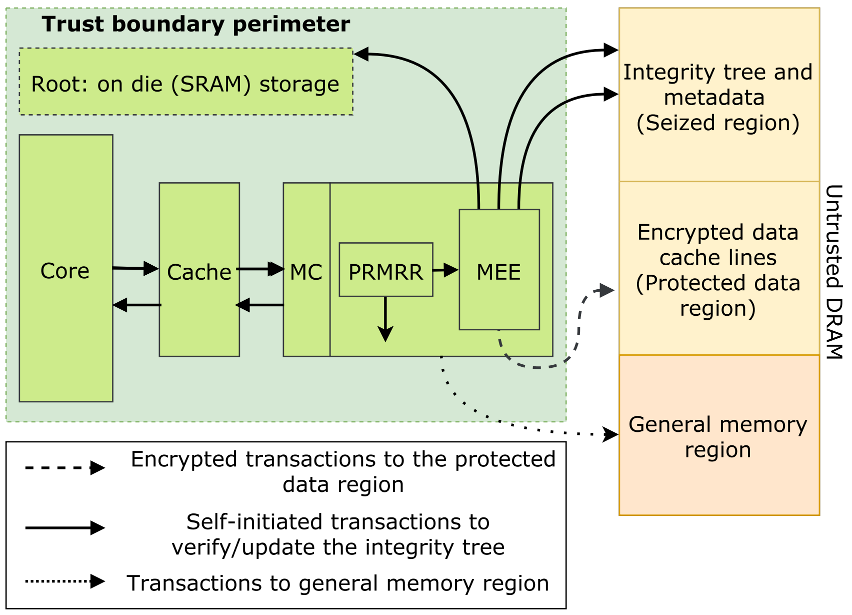

Trust boundary perimeter
Root: on die (SRAM) storage
Core
Cache
MC
PRMRR
MEE
Integrity tree and metadata (Seized region)
Encrypted data cache lines (Protected data region)
General memory region
Untrusted DRAM
Encrypted transactions to the protected data region
Self-initiated transactions to verify/update the integrity tree
Transactions to general memory region

Figure 2.17: Memory Encryption Engine (MEE) architecture [53]

quently, the MEE encrypts or decrypts the data before sending it to or fetching it from the DRAM. There are additional transactions originated by the MEE that either verify or update the integrity tree and rely on a construction of counter and MAC tags. These transactions can access the seized region on the DRAM, and the on-die SRAM array, which acts as the root of the integrity tree.

2.6.3.5 Enclave Attributes

The way an enclave is executed can be adjusted by using the enclave's SECS subfields, known as **enclave attributes**. There are three enclave attributes: DEBUG, XFRM, and MODE64BIT. The first attribute enables the execution of enclaves in debug mode, giving the ability to read and write the enclave's memory, which does not follow SGX's security guarantees if it is enabled. The second field guarantees that the XCR0 register is always set to the value indicated by the extended features request mark (XFRM). The third field indicates that enclaves use 64-bit architecture when the flag bit is set.

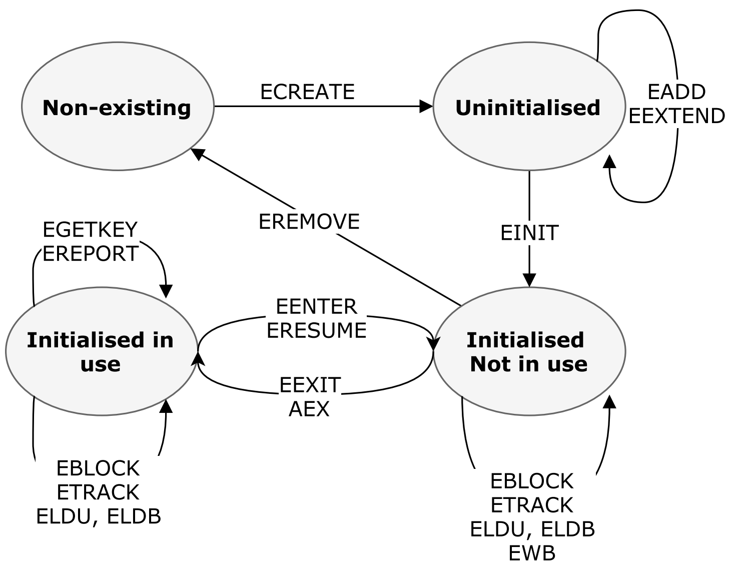


Figure 2.18: Enclave life cycle

2.6.3.6 Access Control for EPC Pages

SGX uses an access control mechanism for each EPC page that is determined when the page is allocated. There are four fields SGX uses to provide access control for EPC pages: readable (R), writable (W), executable (X), and address. The first bit flag allows read by enclave code, the second bit flag allows write by enclave code, and the third bit flag allows execution of code inside the page and inside the enclave.

2.6.3.7 Enclave life cycle

The enclave life cycle comprises of four stages: creation, loading, initialisation, and non-existing stages. During the creation stage, the system software issues the ECREATE instruction, which initialises a newly created SECS using information from a non-EPC page that includes all the required SECS fields. In addition, ECREATE adjusts the INIT attribute value to false. While this value is set to false, the code inside the enclave cannot be executed. The enclave code is executed when the attribute value is true.

The system software uses the EADD instruction to load the initial code and data into the enclave. The main input for the EADD instruction is the Page Information (PAGEINFO) data structure, which contains the virtual address of the EPC page (LINADDR), the virtual address of the non-EPC page that will be copied to the new EPC page (SRCPGE), a virtual address that points to the SECS of the enclave, and a virtual address that points to a Security Information (SECINFO) data structure. The SECINFO data structure contains the EPC page access permissions (R, W, X) and the EPCM page type, which is either PT_REG or PT_TCS.

During the loading stage, the system software will use the EEXTEND instruction to update the enclave's measurement, which is used for the software attestation process.

The initialisation stage includes the Launch Enclave (LE) that the system software must use to receive an EINIT Token Structure. The token is then added to the EINIT instruction, which adjusts the state of the SECS structure to initialised. After the EINIT instruction finishes without any exceptions, the enclave's INIT attribute is set to true. Then, the application software in ring 3 executes the enclave code using SGX instructions.

During the last stage of the enclave life cycle, the system software executes the EREMOVE instruction to clear the EPC pages that the enclave has used. This is achieved by adjusting the VALID field in the EPC page's EPCM entry to zero.

2.6.3.8 SGX Thread Lifecycle

A logical processor is in **enclave mode** when it executes code inside an enclave. During this state, the code that the enclave executes can access its EPC pages. In contrast, when the logical processor is not in enclave mode, it bounces any memory accesses.

Chapter 3

Literature Review

The previous chapter introduced the background information regarding the main technologies and concepts used in this thesis. In this chapter, the literature review illustrates two main challenges: the challenge of protecting resources in a virtualised infrastructure from a malicious unikernel and the challenge of protecting computations executed in a unikernel from privileged software. Even though cloud computing has seen widely adoption by companies, government, and regular users, it still has not resolved effectively the two aforementioned challenges. Providing assurances that, on the one hand, the virtualised resources are protected and, on the other hand, the computations of cloud users are not compromised by internal or external adversaries is a step in the right direction. Hence, we review state-of-the-art of solutions that attempt to prevent privilege escalation, information leakage, and provide trusted execution without loss of confidentiality and integrity.

There are two main areas that we analyse in this literature review. First, we discuss software-based protection mechanisms that use virtualisation to control access to resources in cloud infrastructures. Second, we review research works on hardware-based protection mechanisms that use hardware and trusted computing to protect resources in a cloud infrastructure even from privileged software.

Structurally, this chapter consists of two main thematic sections. In the first section, I review a number of access control systems that use virtualisation to regulate access to resources. The second section reviews works that are predominantly based on hardware to protect resources in a cloud infrastructure. This section is further divided in two parts. The first part focuses on reviewing hardware-based access control systems and the second discusses research works that are based on three Trusted Execution Environments (TEEs): the Trusted Platform Module (TPM), the ARM TrustZone and the Intel SGX. Finally, a summary of this chapter and focus for the following chapter are provided.

3.1 Virtualisation-based Solutions

One of the concepts devised during the early 1970s was the reference monitor [5]. This concept was developed to fulfil the requirement of controlled sharing for multilevel secure computing systems. The reference monitor enforces all the authorised access requests between a number of subjects and objects. The implementation of a reference monitor is the reference validation mechanism, which evaluates every reference to data or a program by any subject from a list of authorised types of reference for that particular subject.

Regarding virtualisation systems, the reference monitor concept is replicated in the hypervisor, which has to act as a reference validation mechanism using an access control module and validate all the access requests to objects. This method fulfils the complete mediation design principle. The following solutions present the state-of-the-art access control in virtualised infrastructures.

Even though, these solutions based on virtualisation provide a view of the current state of the art, they do not provide a way to manage the security of a cloud infrastructure that potentially uses thousands of VMs or in the context of this thesis unikernels. None of the following solutions develop an access control mechanism that is tailored to delegate access to resources in a dynamic way and create a simple mechanism that can be audited in the future.

The solutions discussed in this section are:

- SeL4 [63, 40]
- sHype [103]
- Shamon [76]
- CloudAC [146]

- PIGA-Cloud [20]
- Nova [122]
- XenCap [94]
- Xen on ARM ACM [67]
- CloudVisor [147]
- sVirt [91]

3.1.1 SeL4

SeL4 [63, 40, 62] is a general-purpose OS microkernel that has been formally verified to source code level. The seL4 is derived from the L4 microkernel, in which the implementation includes only address spaces, threads, timer, and a synchronous Inter-Process Communication (IPC) mechanism.

The seL4 microkernel incorporates a capability model for granting access to resources. Capability tables are used to store capabilities and are managed by the microkernel for a particular address space. For this microkernel, capabilities point to blocks of memory. Having a capability-based access control mechanism allows the delegation of access to the address space only by passing the capability to another process.

Memory in seL4 is divided between typed and untyped memory. A number of typed memory regions can hold capabilities and, to prevent tampering with the capabilities, they are managed by the microkernel. Untyped memory can be typed explicitly and used when defining an untyped memory capability.

Even though the first versions of seL4 targeted embedded platforms and CPUs as a general-purpose OS, it has recently gained support for virtualisation. Currently, Linux can be run in a virtual machine on top of ARMv7 processors, and includes experimental virtualisation support for the x86 platform. SeL4 acts as a hypervisor in ring–0 root mode for the x86 platform or hyp-mode for the ARM platform to support executing the virtual machines.

SeL4 provides a minimal TCB, which includes software components that are formally verified, and follows the principle of least privilege. However, it still hasn't been demonstrated that when seL4 is built as a hypervisor can support a large number of VMs in a cloud infrastructure. In addition, seL4 does not provide any protection when the hypervisor is compromised to ensure the security of the tenants' workloads. Therefore, it would be difficult to integrate unikernels with seL4 due to its verified model.

3.1.2 sHype

One of the earliest attempts to provide isolation between various workloads so that there is no potential data leakage is sHype [103, 102]. sHype extends the Xen hypervisor [12] with security mechanisms that have the ability to control sharing of virtual resources among virtual machines. The authors have implemented a mandatory access control policy reference monitor inside the hypervisor that enforces information flow among virtual machines in the same physical host.

The sHype architecture supports secure services, resource monitoring, access control for VMs, isolation of virtual resources, and TPM-attestation. This architecture includes the policy manager, the access control module (ACM), and the mediation hooks. The main aim of the policy manager is to maintain the security policies, and it is deployed as a VM. The ACM is the component in the hypervisor that checks every access request to resources according to security policies. The mediation hooks are added in the hypervisor to trap the VMs' access to resources.

An advantage of sHype is that it can employ a wide range of security policies such as Biba [18], Bell-LaPadula [14], Caernarvon [105], Type Enforcement [21], and Chinese Wall policies [23]. However, sHype is only concerned with inter-VM communication and not with VM coalitions that exist in multiple physical hosts. Even though this architecture reduces the TCB, authoring security policies for a large number of unikernels can be a complex and error-prone task for system administrators. Hence, there is a higher probability that an error is made and an adversary can compromise the data integrity and confidentiality due to a permissive security policy. Currently, the sHype ACM is not included with the Xen hypervisor as it is no longer maintained.

3.1.3 Shamon

Shamon [76] is an extension of sHype that provides a distributed Mandatory Access Control (MAC) architecture. Shamon is a shared reference monitor that is used to enforce MAC policies across a distributed set of VMs. For this reason, this solution implements a layered approach. The way Shamon enforces reference monitoring is from the Xen hypervisor and the OS, using SELinux and the IPsec network controls. Another feature of Shamon is that it supports coalitions of VMs that reside in multiple physical hosts. This works by using the SELinux and the IPSec network to enforce MAC policies at the VM level. sHype controls the access to inter-VM virtual resources.

This solution has a large TCB as it includes the hypervisor and the set of VMs that is monitoring in multiple physical hosts. Even though system administrators can author system-wide security policies for coalitions of VMs, the policies can still be complex and prone to error during editing. This solution is not currently integrated with unikernels, and using MAC policies for a large number of unikernels could make them difficult for system administrator to manage and edit. In addition delegation of privileges in a dynamic way is not supported, since Shamon relies on the static security policies that are authored for the whole system and need to be recompiled and reboot the virtualisation server for the new security policies to take effect.

3.1.4 CloudAC

CloudAC [146] uses a multi-layer access control system that regulates a Logical Virtual Domain (LVD) and Trusted Virtual Domains (TVD). The main focus of CloudAC is the implementation for inter-TVDs, and intra-TVDs, and access control for resource nodes. This solution has a large TCB and prevents implicit information flow. It uses a multi-layer access control requesting a deployment from an LVD master to an LVD agent factory that uses a TPM to generate a trusted measurement and then create an LVD agent. If every operation is successful, then we can create a virtual machine and set the local policy to the local policy driver.

This solution still uses VMs with a large attack surface that could potentially lead to an exploitable vulnerability. CloudAC has multiple layers of access control and security policies have to be authored in multiple levels and stored in different places. This creates a complex management issue when system administrators in different parts of the architecture have to adjust the security policies. The security policies do not support delegation of privileges in a dynamic way. Supporting a large number of unikernels would make the management of security policies for each LVD more difficult instead of having a small number of regular VMs.

3.1.5 PIGA-Cloud

Blanc et al. [20] introduce an integrated solution for access control in cloud systems called PIGA-Cloud. PIGA-Cloud focuses on providing an in-depth MAC protection that controls the flows between the virtual machines and the host, the internal flows of virtual machines and the flow between Java objects and the network flows. This solution extends SELinux and sVirt [91] MAC policies to the Java Virtual Machines(JVM) and OSes such as Linux or Windows. Although this solution offers several mechanisms to protect resources, it supports only the same protection for all layers of the cloud infrastructure using a MAC protection scheme. There is no way to define fine-grained security policies for a particular application since it is integrated with SELinux. In addition, the security policies provide a static view of the system and do not take into account the dynamic changes that occur in a cloud infrastructure. The TCB for this solution is very large due to the fact that the system includes VMs with a full OS and management VMs to provide a distributed access control scheme across physical hosts.

3.1.6 Nova

Nova [122] is a microhypervisor that adopts the microkernel design of OSes. Microkernel architectures remove much of the code in the kernel and keep only the minimum required code inside the kernel. For instance, code to support IPC, virtual memory, and scheduling execute in kernel mode, while all other code is executed in user mode.

Nova aims to reduce the TCB and provide a smaller attack surface. This solution takes advantage of Intel and AMD virtualisation CPU instructions to create an emulated physical host that gives the impression to the VMs that are hosted in real hardware with minimal performance penalties.

Nova employs two design principles. First, the microhypervisor must provide a fine-grained functional decomposition of the virtualisation layer to a microhypervisor. Second, this solution uses capability-based access control that enables the principle of least privilege, which is enforced between multiple components .

Although Nova reduces the current attack surface, it still has to use a full-blown OS for the virtual machines it can execute. There are no protection mechanisms for VMs in place, in the case that the microhypervisor is compromised. In addition, Nova currently does not support unikernels, which could be an interesting integration in which we leverage the already build capability-based access control and then integrate unikernels. Our approach in this thesis takes the opposite way to integrate a new access control paradigm to unikernels and the Xen hypervisor.

3.1.7 Xen-Cap

Xen-Cap [94] uses a capability-based access control model to protect virtual resources in the Xen hypervisor. This solution relies on the concept of capabilities, which is a record in a data structure that is protected by the hypervisor. This is similar to the way kernels in OSes protect capabilities. The data structure that contains the capabilities is called a CSpace. Capabilities are identified by a system-wide unique 64-bit name that is generated using a software random number generator. To gain access to a resource, the capability has to be attached to the CSpace of a VM.

VMs can only possess capabilities through a protected grant hypercall, which ensures that VMs cannot tamper with the capabilities. The application code in a VM can call the check hypercall to evaluate if the capability exists in the CSpace of the VM initiating the access request. If the result of this check is positive, then the application in the VM is permitted access to the resource. Xen-Cap is able to create least-privileged services by leveraging the advantages of capabilities.

This solution is similar to the access control system proposed in this thesis. However, there are distinct differences compared to our system in the way the access control system is implemented and evaluated. First, this work focuses on virtual machines that use a full OS, which creates a large attack surface. Second, this solution does not provide a clear integration path to unikernels and library OSes. Third, Xen-Cap does not provide a performance evaluation to illustrate the overhead of Xen-Cap and how it performs with a large number of VMs. Fourth, this solution does not provide a model or mechanism for using capabilities across a range of physical hosts in a cloud infrastructure. Fifth, this solution does not provide a revocation mechanism.

3.1.8 Xen on ARM ACM

Lee et al. [67] propose a multi-layer mandatory access control mechanism based on Flask that focuses on mobile devices. Supporting mobile devices requires the overheads incurred by the access control module (ACM) to be reduced and the mobile device to be protected from malware and limited resource drain attacks. This solution uses independent access control modules for the hypervisor and each VM. Using this approach reduces the performance overhead when processes in each VM invoke hypercalls to get an access control decision. Another feature of this solution is that it protects from DoS attacks from malware that resides in a VM to other VMs or the hypervisor. This is achieved by controlling the resources such as memory, CPU, DMA, event channels, and battery.

The ACM is based on Flask architecture and can support additional access control models and policies. However, the access control mechanism supports only Type Enforcement (TE) policy and a proprietary policy for preventing DoS attacks. Access requests to physical or virtual resources and management operations is controlled by the ACM via access control hooks. The access control hooks are placed inside hypercall routines to check every request for the resource and to ask another component if the VM has the correct permission to access the resource.

This solution incorporates multiple access control modules that we could adopt for the access control system proposed in this thesis to improve performance. However, this security architecture supports mobile devices and it is not clear how it would scale in a cloud computing context, or how it would deal with access requests originating from a large number of VMs. In addition, this research work supports system-wide security policies that are static in nature, and therefore does not support dynamic delegation of privileges.

3.1.9 CloudVisor

CloudVisor [147] proposes a minimal security monitor following the concept of nested virtualisation that exists underneath a regular hypervisor. The hypervisor is considered to be untrusted software in the architecture. CloudVisor protects the privacy and integrity of the resources that belong to VMs and the hypervisor manages the allocation of resources to the VMs. Hence, there is a separation between protection and resource management that makes it possible to design and develop CloudVisor and the hypervisor independently. The hypervisor has a privilege level that is the same as a regular VM. The highest privilege level is kept for the CloudVisor component.

This solution focuses on protecting resources from a malicious system administrator. The approach taken in this solution is for CloudVisor to intercept all control transfer events between the hypervisor and the VMs. Then CloudVisor converts and secures the event information and forwards the events to the hypervisor to handle them. The bootstrap for CloudVisor uses Intel TXT for delayed launch and TPM to store the hash of CloudVisor. Regarding persistent data, there is transparent VM image encryption and the I/O data is encrypted and decrypted by CloudVisor. Memory pages are protected in the VM from inspection by the hypervisor. For this outcome to happen, CloudVisor intercepts address translation from the VM's physical address to the host's physical address. Next, this solution evaluates whether the ownership of the memory page is the same as that in the page table and encrypts the contents of the page if they are not the same.

CloudVisor employs a minimal TCB and considers the hypervisor and the management VM to be untrusted. Since this solution uses hardware-assisted nested virtualisation to accomplish its goal, there are overheads for intercepting the events in between the hypervisor and the VM. The performance evaluation of CloudVisor shows that there is significant overhead for I/O intensive applications and in certain cases there is a 22% performance overhead in comparison with the Xen hypervisor. In addition, this research work does not support the regulation of access to resources from the point of view of the VM.

3.1.10 sVirt

sVirt [91] provides MAC for VMs that are hosted in a Linux-based physical host that uses Xen or KVM hypervisor. Currently, it is merged with the libvirt library, which is a multi-platform virtualisation library. The access control mechanism extends the SELinux labelling scheme to VMs. The focus of this solution is to enable VM isolation and secure sharing of resources. Isolating VMs protects the hypervisor and other VMs from compromised VMs that attempt to gain access to the host or other VMs.

sVirt labels are applied to resources, dynamically taking into consideration the cur-

Solution	**Conf.**	**Int.**	**CL**	**AC**	**LP**	**PD**	**CP**
seL4	✔	✔	✕	CAP	✔	✔	✔
sHype	✔	✔	✔	MAC	✔	✕	✕
Shamon	✔	✔	✔	Dist. MAC	✔	✕	✕
CloudAC	✔	✔	✔	MAC	✔	✕	✕
PIGA-Cloud	✔	✔	✔	MAC	✔	✕	✕
NOVA	✔	✔	✔	CAP	✔	✔	✔
XenCap	✔	✔	✔	CAP	✔	✔	✔
Xen on ARM ACM	✔	✔	✕	MAC	✕	✕	✕
CloudVisor	✔	✔	✔	Cryptography	✕	✕	✕
sVirt	✔	✔	✔	MAC	✔	✕	✕
VirtusCap	✔	✔	✔	CAP	✔	✔	✔

Table 3.1: Virtualisation-based solutions summary. ✔= addresses; ✕= not addresses; Conf. = Confidentiality; Int. = Integrity; AC = Access Control; CL = Cloud Computing; LP = Least Privilege; PD = Privilege Delegation; CP = Complex Policies

rently executing VMs or statically specified by an administrator. The types of sVirt labels include VM process, VM images and shared information between VMs. The VM can only access resources that are specified in the VM image file. This is comparable to the work presented in this thesis, whereby we specify policies for unikernels inside configuration files. Even though sVirt is targeting virtualisation platforms, it does not provide a specialised access control mechanism for unikernels and managing security policies for a large number of unikernels. In addition, sVirt provides static system-wide security policies that do not enable the dynamic delegation of privileges during runtime for VM.

3.1.11 Summary

In this subsection, we summarise the solutions that were analysed in this section. A table is included to illustrate how the virtualisation-based solutions correspond to the requirements and design principles we are concerned with in this thesis.

Table 3.1 outlines the solutions examined in this section in relation to the support of the confidentiality, integrity, access control type, support for cloud computing, least privilege, privilege delegation and complex policies. For the first two columns we are interested in finding out, if the solutions include the functionality to realise confidentiality and integrity for the resources. The next column shows the type of access control the solution uses to mediate access to resources. If the solution analysed supports cloud computing, it is shown in the cloud column. The least privilege column considers that the solution supports the POLP. The privilege delegation column considers the solution supports the delegation of privileges to other VMs or processes. The last column deals with solutions that require author complex policies to protect the resources in a system.

Most of the works analysed support either a MAC protection system or a capability-based access control mechanism. Xen-Cap supports capabilities on top of a hypervisor and sHype provides the first MAC architecture for virtualised infrastructures. CloudVisor demonstrates a minimal TCB that does not include the hypervisor or the management VM. More research needs to be conducted on supporting unikernels and library OSes with access control mechanisms, as none of the solutions support unikernels.

3.2 Hardware-based Solutions

In this section, we first review research works that focus on providing protection mechanisms based on hardware related to the second solution proposed in this thesis. This can be a specialised co-processor that deals with enforcing access control policies or uses current hardware modules such as Translation Look-aside Buffers (TLB). Second, we present solutions that are based on TEEs, which can be created using the TPM, ARM TrustZone, and Intel SGX. In particular, this section also focuses on TEEs that are integrated with library operating systems. Figure 3.1 illustrates the taxonomy for hardware-based solutions for protecting resources and sensitive data in virtualised infrastructures.

3.2.1 Hardware-based Access Control Solutions

3.2.1.1 CAP

One of the earliest works on creating a hardware-based access control is the CAP computer [137] developed at Cambridge University. This solution focuses on memory protection architectures using a capability-based approach. The CAP computer

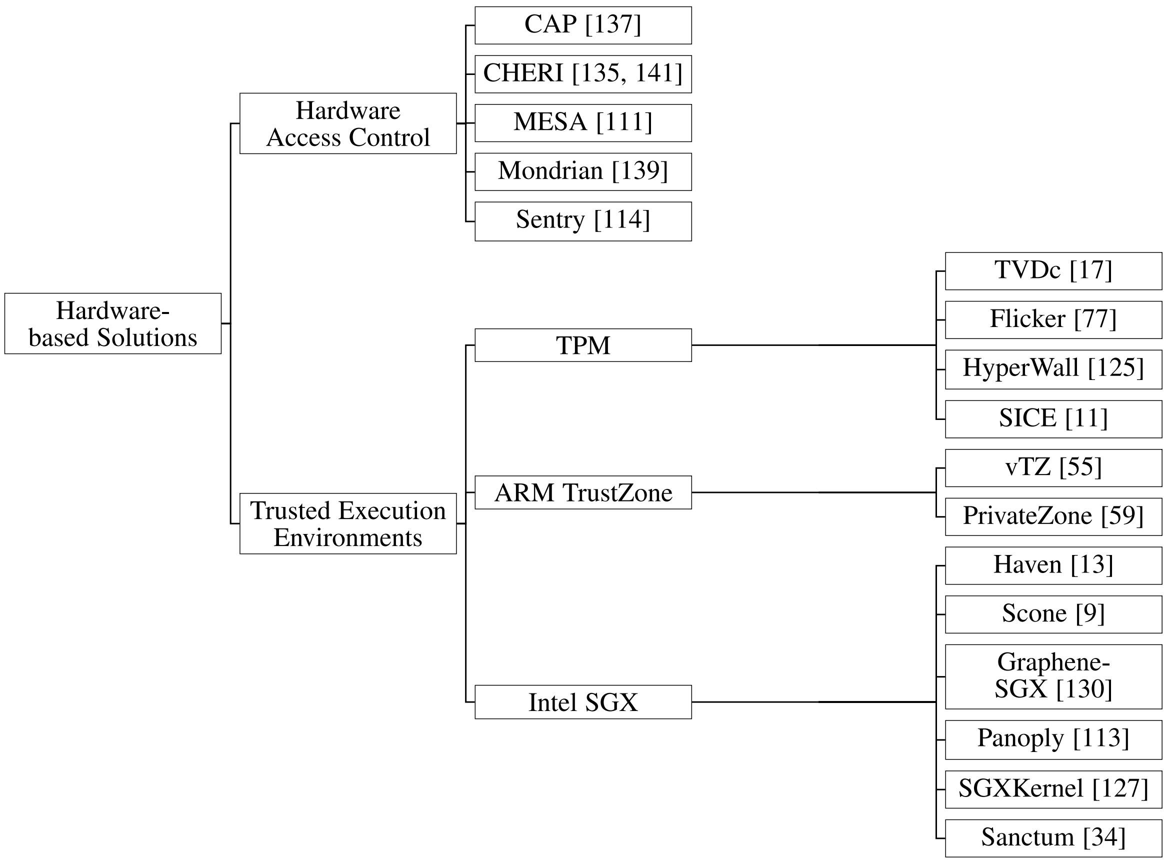


Figure 3.1: Taxonomy of hardware-based solutions

features a CPU that includes a 64-entry capability unit that contains evaluated capabilities. This type of capability includes the capabilities and the main memory locations of the memory segments they address. The memory of the computer is divided into segments with includes 64K 32-bits words. In this solution, the capabilities and data are partitioned such that a segment cannot contain a capability and data at the same time. This partitioning is enforced by using tagged memory, which means that each memory segment is tagged with a type bit that indicates whether it contains capabilities or data.

The CAP computer provides a process tree structure, where the initial memory of a child process is allocated from the memory of the higher-level process. The main execution and protection component is the process that includes the set of data structures and other required objects. Capabilities included in a process's segment can be used to address resource objects. The main reason for designing the process tree structure was to remove the need for a privilege mode of operation. Hence, parent processes have control over the addressing and execution of child processes without needing additional privileges.

The addressing between the parent and child processes requires a capability segment and the Process Resource List (PRL) data structure. The capability segment is the placeholder for capabilities. In order for any object to be accessed, the process must possess the capability for that object. In this solution, the capabilities refer to records in the PRL, which is table local to the process, instead of referring to entries in a system-wide table. One disadvantage of this design is that capabilities cannot be passed easily to other processes. The reason for this is that they address the PRL, which is a local data structure to the process. The TCB for this solution are mainly the hardware components used to realise the access control mechanism and the microcode.

Other early capability systems include the IBM System/38, which was the first major commercial capability system and the Intel iAPX 432. A review of such earlier capability-based solutions can be found in [69].

3.2.1.2 CHERI

Capability Hardware Enhanced RISC Instructions (CHERI) [135, 141] is a clean-slate hardware and software capability architecture. It is a hybrid capability system that combines capability-based addressing with an RISC ISA and MMU-based virtual memory. The hardware architecture employs a capability coprocessor for defining capability registers, and tagged memory for using memory capabilities. This solution implements a RISC memory capability model where a memory capability is an unforgeable reference that grants access to an address space. All memory accesses have to use memory capabilities to maintain memory protection.

This hybrid capability system has a number of advantages such as unprivileged use, fine-grained protection within an address space, pointer safety, and unforgeability of capabilities. The TCB remains minimal as we trust the hardware, the correct implementation of the ISA extension, and certain software components such as the compiler. However, CHERI focuses on RISC architecture and has not been evaluated in a cloud computing context. A possible research thread would be to investigate how CHERI can be integrated with contemporary library operating systems and, by extension, with unikernels.

3.2.1.3 MESA

MESA [112] is a security architecture for the protection of software confidentiality and integrity using a memory-centric design. The focus of this solution is on binding cryptographic properties and security properties to memory instead of each user process. This is a different approach to previous solutions using capabilities and per process capability lists. The authors claim three advantages of using this architecture: first, improved access control of software privacy; second, tamper-resistant secure information sharing; third, protection of software components with separate security policies to collaborate in the same memory address space.

There are two main concepts introduced in this solution: the memory capsule and the principle. The memory capsule is a virtual memory address range that includes a number of related security attributes. The information it can hold is code or data, or both. Each memory capsule can be shared by many processes.

The security attributes that define the memory capsule can include the security protection level, symmetric memory encryption keys, memory authentication signature, and other related security information. A list of memory capsules is managed by an OS kernel for a particular process.

The second concept for this solution is the principle that defines the execution context for memory capsules and is considered as the owner of memory capsules. All the memory capsules in a process that have the same security attribute are executed. The principle that is currently executing can request to access a memory capsule. If the principle has the privileges to access the memory capsule, the access is allowed. If the result is different, then an access violation exception is triggered.

The memory capsule is managed by the OS kernel. When a process is created, then a list of memory capsules is related to it. This is a similar concept with initial capabilities attached to a newly created process in capability-based systems.

This solution provides a method for a tamper-resistant shared memory using access control. Secure sharing is accomplished using the principle that owning the memory capsule grants certain access rights to other principles that are authenticated. This process is facilitated by micro-architecture components such as the Security Attribute

Table (SAT), the Access Control Mechanism (ACM), and the Security Pointer Buffer (SPB).

The ACM is the component that provides the hardware-based access control to protect the memory capsules from untrusted software components. All the information regarding access control is stored in a table that is checked in every memory operation. A fast cache is also used to improve the performance of the access control mechanism.

The TCB of this solution includes the hardware components used in this architecture and the OS kernel. Even though this solution does not target cloud computing it would be interesting to see how this architecture would scale under high processing loads. In addition, the performance evaluation is modelled using a simulation environment and shows that in certain test cases, there is a 5% loss on performance.

3.2.1.4 Mondrian

Mondrian [139] is a fine-grained memory protection solution that allows access control at the level of memory words in contrast to regular memory protection schemes that define access permissions at the page level. Hence, the access control happens at the granularity of individual words, whereby a compressed permissions table is used for improving performance and multi-level permissions caching.

In this solution, all memory regions that are allocated belong to a protection domain. Each protection domain is associated with a permission table that is stored in memory. The aim of the permission table is to define the permission of each address that the protection domain contains.

All memory accesses have to be evaluated to find out if the domain has the appropriate permission. When a memory reference arrives at the CPU, the CPU checks the permission in the address register's sidecar. In the event that the reference is out of range of the sidecar information or the sidecar is invalid, then the CPU tries to load the reference from the Permissions Look-aside Buffer (PLB). The PLB is a cache for protection table entries in the same fashion as the TLB is a cache for page table entries. If the PLB does not have the information regarding the permissions, then the permissions table is searched. If there is a match then the reference is placed in the PLB and reloads the address register sidecar with a new segment descriptor.

This solution does not reduce the TCB in a system and expects that the MMU and page tables have not been tampered with. If the kernel is compromised, then the Mondrian supervisor cannot enforce the security guarantees. In terms of performance, this solution provides an overhead due to the permission tables that reside in memory and they can generate extra memory traffic.

3.2.1.5 Sentry

Sentry [114] introduces a lightweight auxiliary memory access control that employs a virtual memory tagging model and can be adjusted at the user level. The tags are implemented in a permission metadata cache (M-cache) structure that intervenes on L1 misses and exists outside the CPU. The reason for this design is to avoid any modifications in the CPU. Another job of the M-cache is to manage whether the data is permitted to be stored in the L1 cache. The entries in the M-cache include permission metadata for a particular virtual page. Each entry specifies a 2-bit permission metadata for the access permission to a particular virtual page. The 2-bit permission metadata encodes the following four different access permissions: read-only, read/write, no access, and execute.

This solution creates an additional access control path and creates permissions metadata if the application requires fine-grain access control. There is an unused bit in the page table entry that is used to encode a bit; if it is set, the application uses the M-cache for accesses to the page, otherwise the hardware does not consult the M-cache. While the bit is set and there is an L1 miss, the M-cache is indexed using the virtual address of the access to evaluate the metadata. The advantage of this design is that it removes any need for checking whether the data access hits in the L1 cache

Sentry allows an application to specify its own protection models for different software modules without depending on the OS. The access checks are only executed when the application requires fine-grained or user-level access control. These checks are performed after the page-sized regular TLB enforcement.

3.2.2 Trusted Execution Environments

3.2.2.1 TPM

Trusted Virtual Datacenters The Trusted Virtual Data center (TVDc) [17, 16] is another solution that integrates virtualisation security, isolation, integrity and management. This research provides isolation and integrity guarantees for workloads called Trusted Virtual Domains (TVDs). The TVD is a coalition of VMs and resources that interact to achieve a common objective. The TVDc controls the isolation of the communications between TVDs and allows interactions without any restrictions between VMs inside the TVD that are part of the same Virtual LAN (VLAN).

The main components of the TVDc are the Trusted Platform Module (TPM), the virtual TPM (vTPM), sHype, and the system management software. The TPM provides the load time attestation mechanism to guarantee the integrity of the software that is executing inside the VM. Each VM is assigned to a (vTPM) that facilitates the integrity measurements of the software, since the physical TPM cannot support many VMs at the same time. sHype is the component that enforces mandatory access control policies between different TVDs. This is achieved by assigning a unique security label to the elements of a TVD that enforces the MAC policies. For instance, a VM accesses a resource only if the VM has the same security label as the resource, otherwise the access is not permitted. The system management software acts as a central point that oversees the correct administration of TVDs. The software provides a hierarchical view of the infrastructure and assigns different roles to users according to their requirements. For instance, an IT data centre administrator delegates the management of a TVDc to a TVDc administrator and the management of the TVD to the tenant administrator. Each type of user is assigned the appropriate set of privileges and access to the system.

This solution provides a coarse-grained isolation policy since its unit is the TVD rather than the VMs and the resources in a virtualised infrastructure. The isolation policy is composed of two parts. First, the labels that are assigned to VMs and resources. The labels define the security context for the TVD. Second, a set of anti-collocation definitions that enforce policies that restrict which VMs can be executed in the same physical host according to the TVD to which they belong.

This solution does not protect from malicious administrators and compromised hardware. Compromising the TPM means that the integrity measurements reported to the central management software are erroneous and could open the door for malicious users to compromise the software that resides in the TVDc. There is no evaluation on how this solution would perform if there was a large number of VMs in a physical host and how it could be managed efficiently by the different users of the infrastructure.

Flicker Flicker [77] is a security framework for executing sensitive code in isolation while having a minimal TCB and providing fine-grained attestation for the software executed to an external party. Flicker uses the TPM and CPU-based virtualisation extensions such as Trusted Execution Technology (TXT) from Intel and Secure Virtual Machine (SVM) from AMD. This solution uses the measured late-launch mechanism to run a portion of a program referred to as Piece of Application Logic (PAL) isolated from the OS, and then returning control to the OS after the execution is completed. The TCB includes only 250 additional lines of code that the PAL needs to trust for its security. Another feature of Flicker is that the TPM can attest to the correctness of the PAL by sending measurements to an external party. This is achieved even though the PAL does not have access to OS services during execution.

The PAL code executes at a high privilege level; therefore, the OS must trust the code. Flicker needs to switch between trusted and untrusted modes many times if the application has many PALs. This is exacerbated by the slow performance of the TPM with high cost during context switching and the fact that the system is halted during the execution of isolated code.

HyperWall HyperWall [125] proposes a new hardware architecture that strengthens system security by safeguarding a VM against a malicious hypervisor. The main hardware additions to the architecture facilitate the isolation of the VM's memory from the hypervisor, from the Direct Memory Access (DMA) path, and from other VMs.

This solution focuses on memory protection for VMs. There is a page table that uses address translation from the guest physical pages to the machine page. The hypervisor arranges the page table mapping when it assigns the memory for the VM. The hardware protects the VM's memory from updating the page table mapping by the hypervisor. Tenants can specify how memory regions of the VM can be protected by the hardware. After the hypervisor loads the VM, the VM image is loaded and the specified protections requested by the tenant are applied. Using this approach, the hypervisor cannot adjust the protected memory while the VM is running.

Another feature of HyperWall is the Confidentiality and Integrity Protection table (CIP table). This table includes the protection information for every machine memory page and for every VM. This information is stored in the DRAM and only the hardware is able to access it. The protections that the tenant specified are stored in the pre-CIP memory region. The pre-CIP information can be checked to find the necessary protections for that particular page.

This solution employs certain mechanisms that evaluate the integrity of each VM and whether or not the system is trustworthy. However, it only protects the VMs from a malicious hypervisor and requires additional hardware components that are not available in commodity CPUs. HyperWall does not protect from any form of physical attack.

Strongly Isolated Computing Environment (SICE) Strongly Isolated Computing Environment (SICE) [11] is a security framework that provides a hardware-based isolation and protection environment for sensitive workloads, at the same time as running untrusted workloads over the same hardware. This solution depends on a minimal TCB that consists of the hardware, the BIOS, and the System Management Mode (SMM). It includes an isolated environment that consists of the isolated workload and the security manager. The isolated workload is the software, such as a program or even a complete VM. The security manager is the minimal software layer that manages exceptions and the page tables. The aim of the security manager is to disallow the isolated workload to access the memory of the OS or other untrusted applications. The SMM prevents the untrusted OS and its applications from accessing the isolated environment. The only way that the SMM can be triggered is by using a single interface that calls the System Management Interrupt (SMI). The SMI handler can execute one of four functions when it receives an SMI, which can be to create, enter, exit, or terminate an isolated environment.

There are three stages to enter SICE from the host OS. First, a System Management Interface (SMI) triggers the SICE operations. Second, the SMI handler prepares the isolated environment and the new SMRAM structure. Third, upon entering the isolated environment, the security manager disallows the isolated workload to access the host

OS memory.

This solution can only provide one isolation zone and support only one VM. This means that it would not be easy to use in situations with a large number of VMs. The system depends on the correctness of the security manager and the device emulation in the hypervisor or the host OS. A VM that resides in the system is a possible attack vector to the security manager. If a malicious user compromises the security manager, then she could exploit its access privileges and infiltrate other parts of the system.

3.2.2.2 ARM TrustZone

vTZ: Virtualizing ARM TrustZone vTZ [55] proposes a security architecture that virtualises the ARM TrustZone. It securely provides a guest VM with its own virtualised guest TEE utilising hardware that already exists. The main idea behind this solution is to separate functionality from protection using a secure co-running VM that acts as a guest TEE. At the same time, the hardware TrustZone enforces the isolation between the guest TEEs and the hypervisor. This is accomplished by using a minimal security monitor that resides in the physical TrustZone and mediates the memory mapping virtualisation and the world switching. This solution takes advantage of protected code that exists in a Constrained Isolated Execution Environment (CIEE) to assist with the virtualisation and isolation among guest TEEs.

The final design of the vTZ includes a minimal TCB for the secured modules in the secure world. It uses one hypervisor and one VM to simulate a guest TEE. The physical TrustZone is used to enforce protection. vTZ uses the secure world to evaluate the booting sequence and executes the integrity checking to ensure that the boot process has not been tampered with. Memory protection uses the Secure Memory Mapping (SMM), which is a module in the secure world that controls all translation tables. For protecting CPU states during context switching and while executing guest TEEs, the Secured World Switching (SWS) hooks and evaluates every switch in the secure world. In addition, the virtual peripherals are isolated via securely virtualising resource partitioning of peripherals and interrupt.

This solution is not concerned with any physical attacks, as are other hardware-based solutions. It provides protection for booting, CPU states, memory, and peripherals. The authors have considered different attacks to their system such as tampering with the system code, core reuse, DMA attacks, and debugging attacks. vTZ does not protect from bugs that exist in CIEEs, even though it constrains the CIEE's privileges to prevent their misuse. Regarding performance, there is significant application overhead compared to a native environment and there is no evaluation of the scalability of the solution and how it would cope with a large number of guest TEEs.

PrivateZone PrivateZone [59] is a framework that uses ARM TrustZone to create a TEE for mobile devices. TrustZone uses hardware-based access control to isolate the secure world from the Rich Execution Environment (REE), which contains regular OSes. Using PrivateZone, developers are able to execute the Security Critical Logic (SCL) in a Private Execution Environment (PrEE). The SCL is specified by the developer in the application and then deployed to the PrEE. During execution of the application, the SCL can be called to execute securely without disclosing any information to the REE or any other SCLs that reside in the PrEE.

In essence, PrivateZone provides an extra execution environment, which resides between the REE and the TEE. PrivateZone creates two stage–2 page tables for the PrEE and the REE during booting. At the end of the booting process, there are three conceptual environments: the REE, the TEE, and the PrEE. The TEE is protected by hardware-based isolation, whereas the PrEE is isolated using software-based techniques. The PrEE is created by using both the ARM virtualisation and security extensions. The monitor mode that manages the switches between the REE and TEE includes the main framework for PrivateZone.

Developers create an application by using the PrivateZone library, which includes a set of operations. For instance, the deploySCL is one such operation, which sends a request to make a copy of the SCL and to position it inside the PrEE. Even though the API is limited to only basic operations needed for showcasing how the framework operates, one could add more operations if required.

This solution has a TCB that includes the software of the PrivateZone framework and the hardware that provides the TrustZone virtualisation and security extensions. The REE includes a regular OS that has a large attack surface and includes a number of PrivateZone components that could be exploited. This solution relies on the small number of lines of code for PrivateZone in the TEE, which can be manually evaluated for correctness. It does not currently protect from hardware attacks and supports only mobile devices.

3.2.2.3 Intel SGX

Haven Haven [13] is a system that uses shielded execution to protect an application from the platform that is executing it. This is accomplished by using Intel SGX hardware protection to mitigate attacks from privileged code, and physical attacks such as memory probes. Hence, this solution relies on trusted hardware rather than a hypervisor to confine the untrusted OS.

Each application is running in an isolated execution environment called an enclave. The enclave, apart from the application's code and data, includes the Drawbridge library OS [98] and the shield module. The Drawbridge comprises the picoprocess and the library OS. The picoprocess provides a minimal ABI in relation to the full Windows interfaces and protects the host from a compromised guest. The library OS includes a refactored Windows version as a set of libraries that is executed inside the picoprocess. The shield module secures the remaining dependencies from a compromised OS. Outside the enclave there are two components running. First, there is an untrusted driver running, which provides memory management and an interface to SGX. Second, there is a user mode software that is untrusted and connects the trusted code with the OS.

This solution includes a whole library OS that runs unmodified applications inside the TCB, which is millions of LOC. Executing a whole application inside an enclave has two disadvantages. First, SGX 1 only supports enclaves with up to 128MB of memory. This is changed in SGX 2 and can support a dynamic allocation of memory. However, there are no CPUs in production to test this feature. Hence, with SGX 1, when an application requires more that the maximum enclave memory, an in-memory paging between the trusted and untrusted regions is executed which results in a performance overhead since the pages have to be encrypted again. The second disadvantage is that using whole applications inside the enclave results in an increased TCB with higher risk of vulnerabilities in the application code.

Scone Scone [9] introduces a security mechanism for Docker containers using Intel SGX. It enables the protection of unmodified applications from outside attacks. The application needs to be recompiled and uses a special standard C library that assists with the execution of system calls. These system calls provide an external interface for the application to access the untrusted memory. Scone protects this external interface by evaluating the values that are received and sent before passing any values to the application.

In addition, Scone includes functionality to transparently encrypt and authenticate data using shields that statically link the shield library with the application. This solution provides a file system, network and console shield. The first shield ensures the confidentiality and integrity of files through encryption and authentication. The second shield uses TLS to secure all network communications to and from the enclave. The third shield encrypts unidirectional console streams.

This solution places all the code inside an enclave which creates a large TCB that executes in the same privilege level. The enclave contains both sensitive information and information that is not sensitive. SCONE only supports Docker containers and Linux platforms.

Graphene-SGX Graphene-SGX is a framework that adds support for Intel SGX to Graphene library OS. The aim of Graphene-SGX is to support unmodified rich applications on SGX. Additional features in Graphene-SGX include shielding dynamically loaded libraries, forking in an enclave, and secure inter-process communication (IPC).

Graphene-SGX expects an application to have a manifest whereby the allowed resources are specified. The manifest is also used to specify cryptographic hashes of trusted files. Multi-process applications are supported by executing a library OS in an enclave for each process and the state is managed using message passing. Local attestation is used to authenticate each enclave with the other and creating a secure communication channel.

An enclave in Graphene-SGX consists of the shield code and data, a library OS, the libc library, the manifest and file hashes, user libraries and other executable code and data. Graphene-SGX provides the implementation and shielding of the Linux ABI for applications in enclaves. It shields dynamic loading whereby a unique signature is created for a combination of executable and dynamically linked libraries. The Platform Adaptation Layer (pal-sgx) is an untrusted component that calls the SGX driver to initialise the enclave.

Graphene-SGX supports process creation inside an enclave whereby a parent enclave clones the process and creates and starts a new enclave, where they locally exchange an encryption key, validate each other's attestation, and then the parent enclave sends a process snapshot to the child process over secured RPC and the child enclave resumes execution. However, a compromised enclave could create numerous enclaves that can eventually starve resources in the physical host and create a denial of service for the users of the host. In addition, if a compromised enclave starts executing and abusing resources in a physical host, it is difficult to stop it [106].

Even though this solution can assist developers in quickly setting up their applications to execute inside an enclave they cannot be optimised from the beginning. An ideal scenario would be to assemble libraries that are already optimised for executing inside an enclave and use the final binary for the enclave. Using this approach, we ensure that the performance and behaviour of the binary is optimised to be used inside an enclave. Using an unmodified application inside an enclave requires an expensive swap of encrypted memory whenever the memory of the enclave grows more than 128MB. This issue is solved in the SGX 2 specification whereby we can dynamically allocate memory to the enclave. Currently, there is no hardware that supports SGX 2.

Panoply Panoply [113] is a system that introduces a new abstraction called a micro-container (micron) to provide enclave access to system calls and other OS abstractions. This solution focuses on creating a minimal TCB instead of improving performance, and follows a different design in relation to other solutions. Panoply delegates the implementation of OS system calls to the hosted OS and does not emulate them inside the enclave. This is achieved by redirecting all the system calls to the Panoply shim library, which in turn makes the actual call to invoke the requested functionality in the OS. The shim library includes mechanisms that detect and abort any suspicious reply originating from the OS.

Another feature of Panoply is that it enables partitioning of Linux applications to one or more microns. This works by annotating functions in the application's source code with different micron identifiers. The source code and the annotations are entered to the Panoply build toolchain to create micron binaries. Each micron binary consists of the shim code, the Intel SGX SDK library, and libraries that are used by the application. Panoply ensures the integrity of inter-enclave communications and the correct execution of the application even if there is a compromised OS.

This solution provides an architecture that could be improved if the focus was on creating a minimal library OS that does not need to support the whole POSIX API. Although this would create microns that are not compatible with other platforms, they would be optimised to be deployed inside an enclave with less overhead during execution.

Even though their design decision results in a smaller TCB in comparison with other solutions, their CPU overhead is 24% on average and 5%–10% in relation to Graphene-SGX. Using Panoply in a cloud computing context, creates a micron with less throughput for serving static web sites, since there is a need to encrypt data entering the enclave memory and decrypt data leaving the enclave memory. Hence, Panoply would not be a performant solution to implement secure services in the cloud.

SGXKernel SGXKernel [127] introduces a library OS that aims to execute single-process applications inside an enclave with security and efficiency in mind. The authors argue that the main reason for performance overhead in library OSes is the enclave transitions. The enclave transitions occur when the trusted code executed inside an enclave invokes system calls in the hosted OS that are untrusted and in the opposite direction. Hence, they propose a switch-less architecture for this solution.

This architecture comprises two parts: the trusted part, which is in the enclave, and the untrusted part, which executes outside the enclave. Both of these parts run at the same time using SGX or OS threads. SGXKernel provides asynchronous communication between the two parts using shared data structures, which exhibit two characteristics: non-blocking concurrency and resistance to tampering. There are also three types of call: the delegated calls (Calls), the I/O stream calls, and the virtual interrupt signals. All three types of call facilitate communication between the two parts without any enclave transitions. In addition, SGXKernel uses a multi-threading mechanism inside the enclave to prevent any enclave transitions.

This solution provides a minimal TCB that is considered smaller than other library OSes that support SGX. It is closely related to the work in thesis as the SGX is designed as a library for running single-process applications inside enclaves. This is similar to the concept of unikernels, which have a single-process philosophy. The difference is that, for MirageOS, the unikernels are single-threaded. SGXKernel is not a specialised library OS for cloud computing and still has a significant overhead over native execution of applications.

Sanctum Sanctum [34] is a software/hardware security architecture that emulates the Intel SGX design offering the same level of protection from additional software attacks that attempt to gain information from an application's memory access patterns. Sanctum eliminates unnecessary complexity, which leads to a simpler security analysis. Even though Sanctum is similar to Intel SGX, it does not cover physical attacks, since there is no MEE to perform the encryption of code or data that are then written in DRAM.

Sanctum protects from side-channels using two methods. First, an enclave in Sanctum manages its own page tables and page faults, in contrast to SGX, which requires the OS or the hypervisor to manage them. Second, Sanctum assigns a separate DRAM region for each enclave that corresponds to a particular set inside the shared Last-Level Cache (LLC). Using these two approaches, Sanctum is able to protect from malicious software that attempts to acquire information from the memory access patterns of an enclave.

This solution uses a trusted software component called security monitor instead of using microcode to implement the low-level functionality, as does SGX. Another software component used is the measurement root, which is placed inside an on-chip ROM. This component works during the booting process of a Sanctum system to calculate a cryptographic hash of the security monitor and create a monitor attestation key and certificate based on the monitor's hash. Afterwards, the security monitor provides an interface for enclave management and DRAM region allocation. The enclave management includes calls to create or destroy enclaves. Exiting or entering the enclave requires the appropriate monitor calls. The OS can use monitor calls to set a monitor-protected register such as the page-table register or the allowed DMA range register.

3.2.3 Summary

In this subsection, we provide a summary of the hardware-based solutions that were analysed in this section. A table is included to illustrate how the hardware-based solutions correspond to the requirements and design principles we are concerned with in this thesis. There is an explanation of the different table columns in subsection 3.1.11.

The works analysed in this section are divided into two broad categories. First, we included the solutions that predominantly use a hardware-based access control mechanism. In this area of research, there are solutions that implement capabilities on hardware such as CAP and CHERI. Other solutions, such as MESA, use a memory-centric approach whereby an address space is protected using cryptography. Mondrian focuses on providing a fine-grained memory access control at the level of a word. Sentry introduces a lightweight memory access control that uses a virtual memory tagging model. The above solutions demonstrate a wide variety of mechanisms for providing access control using hardware in terms of access control granularity, adjustments to hardware, and ease of configuring security policies. However, none of the above solutions satisfy all of our requirements as illustrated in Table 3.2.

Second, we analysed solutions that construct TEEs. These solutions not only use hardware-based access control but provide additional support for protecting computations from privileged software and even physical attacks. We analysed three main methods of constructing TEEs: TPMs, ARM TrustZone, and Intel SGX. Even though there are other ways of constructing TEEs, such AEGIS [124], which add specialised hardware, we chose to analyse only the above three methods. This thesis focuses on commodity hardware that cloud providers use in their infrastructure.

Hardware-based solutions are a promising way of researching how to regulate access to resources in a cloud infrastructure and how to protect computations in an untrusted cloud. However, none of these solutions have investigated the protection of security sensitive computations when the application in the cloud is partitioned in two parts. A trusted part that focuses on protecting the security sensitive computations in an application and the untrusted part. Our own solution achieves this requirement by partitioning applications executed in the cloud and protecting the trusted part.

This chapter discussed virtualisation and hardware-based works that implement access control mechanisms and protect computations in the cloud. We analysed several promising solutions such as sHype and Nova, which mainly use a virtualisation-based approach to provide access control to resources in the cloud. These solutions offer a way to regulate access using a MAC or capability paradigm.

There are several interesting solutions that mainly use hardware to provide access control and TEEs. All of these solutions create a secure foundation for protect-

Solution	**Conf.**	**Int.**	**TCB**	**PM**	**Cloud**	**AP**
CAP	✔	✔	×	CAP	×	×
CHERI	✔	✔	✔	MAC	×	×
MESA	✔	✔	×	MAC	×	×
Mondrian	✔	✔	×	MAC	×	×
Sentry	✔	✔	×	MAC	×	×
TVDc	✔	✔	✔	TPM	✔	×
Flicker	✔	✔	✔	TPM	✔	×
HyperWall	✔	✔	×	MAC	✔	×
SICE	✔	✔	✔	SMM	✔	×
vTZ	✔	✔	×	ARM TZ	✔	×
PrivateZone	✔	✔	×	ARM TZ	×	×
Haven	✔	✔	×	SGX	✔	×
SCONE	✔	✔	×	SGX	✔	×
Graphene-SGX	✔	✔	×	SGX	✔	×
Panoply	✔	✔	✔	SGX	×	✔
SGXKernel	✔	✔	✔	SGX	×	×
Sanctum	✔	✔	×	HW	×	×
UniGuard	✔	✔	✔	SGX	✔	✔

Table 3.2: Hardware-based solutions summary. ✔= addresses; ×= not addresses; Conf. = Confidentiality; Int. = Integrity; TCB = Minimal Trusted Computing Base; PM = Protection Mechanism; AP = Application Partitioning

ing resources and protecting from privileged software. In particular, SGXKernel and Graphene-SGX are solutions closer to our requirements, which use a library-OS inside an enclave. Furthermore, Panoply is another research work that is close to our second solution. It provides partitioning of Linux applications using microns and supports the POSIX API. However, this work does not focus on executing applications in a virtualised infrastructure.

Taking into account the results from our analysis, we argue that there is a research gap for protection mechanisms that on one hand protect resources in a virtualised infrastructure and on the other hand protect security sensitive computation from an untrusted cloud. Hence, our aim is to compose solutions that enforce the principle of least privilege for unikernels in virtualised infrastructures and create a security architecture that protects unikernels from an untrusted cloud.

Chapter 4

Capability-based Access Control for Unikernels

In this chapter, we study the security of virtualised hosts and propose an access control system for unikernels. The system regulates access to resources while at the same time preventing privilege escalation attacks in cloud infrastructures.

4.1 Introduction

Cloud infrastructures use virtualisation to multiplex physical resources across multiple tenants. A recent trend in virtualisation technologies is to create large numbers of lightweight VMs running on a server aimed at massive consolidation of resources and toward a superfluid cloud. Cloud computing enables the on-demand sharing of virtual resources to multiple cloud tenants. Virtualisation provides the backbone that multiplexes physical resources and enables cloud computing services. A recent trend in virtualisation technologies is to create large numbers of lightweight VMs running on a physical server. Using this approach, CSPs aim for a massive consolidation of resources and toward a superfluid cloud [74].

Recent security incidents in cloud computing demonstrate that there is an increased number of vulnerabilities. Insecure interfaces and APIs are some of the most common ways adversaries use to infiltrate and distort cloud services [64]. Privilege escalation from user space to guest kernel in VMs and VMs escaping to the hypervisor are two of the most prominent attacks in cloud infrastructures [82, 39, 85, 83]. Vulnerabilities or bugs in hypervisors or VMs can be exploited by malicious users. For instance, a recent vulnerability called *VENOM* [44, 86] enables an attacker to escape from the isolation barrier that a VM provides. Subsequently, an attacker can elevate her privileges and access other VMs that reside on the same host and access the network.

These security incidents call for two complementary strategies, which we shall discuss in turn:

- We can reduce the likelihood of intrusions, attempting to minimise the attack surface, hence depriving the adversary of opportunities to exploit vulnerabilities.
- Additionally, we can mitigate the effect of an intrusion, adhering to the Principle of Least Privilege (POLP), thereby minimising the possible privilege escalations and errors.

Following the argument by Manadhata [73], we posit that the greater the attack surface, the less secure the system. In a cloud computing context, virtualisation solutions suffer from a large attack surface in VMs and hypervisors that adversaries can probe for vulnerabilities to exploit. A typical Xen environment has a large TCB, where the Xen hypervisor has $\sim$ 150K SLOC [144] and its control domain (i.e. Domain0) adds 1M SLOC. Domain0 encompasses a full OS within and a wide interface that manages other guest VMs. In this work, we focus on the attack surface of the guest VMs.

VMs in a cloud infrastructure include a full OS that incorporates a large attack surface. Consequently, we aim to minimise the attack surface of VMs by removing any unnecessary software components. Unikernels have been developed in response to this observation. A **unikernel** [71] is a lightweight VM that includes only the necessary libraries it needs to function, but no guest OS. For example, vulnerabilities such as ShellShock [84] do not affect unikernels because there is no shell process for the adversary to exploit. In addition to the minimal attack surface, unikernels have a small footprint, faster booting times and just-in-time summoning [70]. We can construct unikernels for cloud computing using a library operating system, such as **MirageOS** [72].

Even though a unikernel implementation affords us a minimal VM attack surface thereby reducing the likelihood of intrusions, we still need an efficient access control system to regulate access to resources. A rogue unikernel could abuse virtual resources and distort services for other cloud tenants, hence constituting an intrusion that needs to be mitigated. This is exacerbated by the fact that unikernels do not employ a distinction between a user mode and kernel mode. In regular OSes there is a context switch between a user mode to kernel mode, every time a user application requests access to a kernel resource. In unikernels this local access check cannot be executed, since the unikernel does not have a user mode. Therefore, a potential malicious unikernel can probe the whole kernel API and in turn the hypervisor API to attempt a virtual machine escape. An access control system as a means to mitigate intrusions should account for the characteristics of unikernels, such as near instant booting, and provide least privileged unikernels. Hence, the goal of this work is to develop a protection mechanism that enforces the principle of least privilege for accessing resources that unikernels need during execution in a virtualised host. We call this protection mechanism **VirtusCap**.

4.1.1 Contribution

We contribute the first virtualisation-based access control mechanism that is focused on cloud workloads based on unikernels. As unikernels become more prevalent in cloud infrastructures and even in areas such as edge computing, there will be the need to provide protection mechanisms that confine unikernels only to the privileges they need to properly function. VirtusCap uses capabilities to enforce security policies on resource objects that reside in cloud infrastructures. We chose to implement the capability-based access control model that is known to follow the principle of least privilege. A salient feature of capabilities is the ability to delegate privileges to other processes in a dynamic, rather than a static system-wide way. This matches the changing nature of cloud infrastructures that employ unikernels. Capability-based access control requires a subject to possess a capability to access a resource object, which can be low level, such as accessing a particular memory region, or it can be high level, such as accessing files or network ports. VirtusCap is a protection mechanism that averts a malicious unikernel *(subject)* from accessing resources *(object)* that it does not need for its job.

4.1.2 Outline

The remainder of this chapter has the following structure. We first discuss the system and the threat model. Second, we present the system architecture of VirtusCap. Third, we discuss the implementation details of the access control module. Fourth, we evaluate the security of our system using a semi-formal method and using Saltzer's design principles. Fifth, we evaluate the performance of VirtusCap. Finally, we conclude this chapter and discuss how VirtusCap could be integrated with other hypervisors.

4.2 System and Threat Model

In VirtusCap, we consider a virtualised host that is part of a cloud service provider's virtualised infrastructure managed by a cloud administrator. The virtualised infrastructure is partitioned to a number of tenants and each partition is managed by a tenant administrator. For this research, we focus on a single virtualised host that is a member of the tenant's partition.

4.2.1 System Model

In the system model, we assume the following main entities inspired by the XACML policy architecture [93]:

Unikernel Developer (UD). This represents the users that implement a unikernel using a set of MirageOS libraries. The implementation can happen in a system running Linux or Mac OS X and then be recompiled as a Xen VM and deployed to the virtualised host.

VirtusCap Access Control Module (VACM). This software component represents the access control module that decides what resources can be accessed. It is the *Policy Decision Point (PDP)* for the VirtusCap architecture and resides in the protected environment of the hypervisor.

Capability Library (CapLib). This entity represents a MirageOS library that is used in every unikernel in our system and provides an interface for the unikernel to communicate with the hypervisor. This communication happens only via the capability API. Any other means of communication with the hypervisor is prohibited.

Security Hooks (SH). Security hooks act as *Policy Enforcement Points (PEPs)* that intercept access requests from unikernels and forward them to the Xen Security Module (XSM) and, in turn, to the VirtusCap access control module (VACM). If the access request is allowed according to the security policies, the security hooks allow the access request.

4.2.2 Threat Model

The goal of VirtusCap is to provide isolated cloud services with a flexible fine-grained access control system that focuses on unikernels. We use a capability-based approach to create least-privileged services that contain one or more unikernels. The TCB of our system includes the hypervisor and Dom0 VM.

We consider an adversary to be a guest domain that is part of the system. A guest domain in our system can either be a unikernel or a VM. A VM would have a higher probability to act as a malicious insider than a unikernel, primarily due to the larger attack surface of a VM. There are three scenarios for a malicious guest domain. First, a tenant misconfigures a guest domain accidentally. Second, a remote attacker that infiltrates a guest domain vulnerable to external attacks. Third, a malicious administrator that uses the management interface to gain extra privileges and misconfigures other tenant's guest domains.

A malicious user can interact with our system in three ways. First, the malicious user can link a number of libraries and can construct a unikernel using MirageOS. Second, the illicit user can change the code of the unikernel, adjusting the behaviour and intent of the unikernel. Third, the adversary has the ability to change the configuration of a unikernel attempting to access other unikernels or VM resources.

VirtusCap does not protect from side-channel attacks, such as timing, or power analysis and covert-channel attacks. In addition, protection against DoS attacks is not in the scope of this research. For VirtusCap, we are not concerned with the availability of unikernels and the quality of service that the system provides. Another limitation of VirtusCap is that it is unprotected when an adversary boots the hypervisor using a tampered kernel or a tampered MirageOS compiler. If the OCaml compiler or MirageOS toolchain could be tampered with, then a user creates a unikernel that is already compromised.

VirtusCap considers VM escape attacks in which the adversary attempts to escape to another VM/unikernel. In addition, VirtusCap considers escaping attacks to the hypervisor whereby the adversary tries to elevate her privileges and gain access to the hypervisor.

4.3 System Architecture

In this section, we discuss the design decisions we made for the VirtusCap architecture. Our research focuses on a multi-layer access control architecture that provides fine-grained security policies over a virtualised infrastructure. Our architecture can be seen in Figure 4.2.

VirtusCap was designed in such a way to accommodate cloud workloads that employ unikernels. We aim for a protection mechanism that regulates access to resources in a dynamic cloud infrastructure. The near-instant booting, just-in-time summoning of unikernels and not including a user mode are three of the characteristics unikernels exhibit. The architecture of VirtusCap address these characteristics by first using a capability-based approach to access control that enables the POLP and delegates privileges to other unikernels without the need for a static system-wide security policy. A system-wide MAC policy has the disadvantage that it might not represent the current state of the cloud infrastructure. In addition, complex security policies for a large number of unikernels increase the probability of a configuration error. VirtusCap is designed to provide a way for unikernel developers to author security policies for their applications included as a configuration file during the building process of a unikernel. Unikernels call the VirtusCap API to request access for a particular resource from VirtusCap and delegate privileges to other unikernels as well.

Second, using a layered architecture we address the fact that unikernel bundle together application code and kernel libraries in one blob that has access to the whole kernel API, which in turn can execute calls to the hypervisor. VirtusCap grants access only to resources that unikernels have the capability to use. Thereby restricting access to what calls can be made to the hypervisor. This is achieved by including a VirtusCap-enabled library in the unikernel, that gives the ability for applications to request access to resources using capabilities as tokens of access. A virtualised host includes multiple software layers that need to be protected.

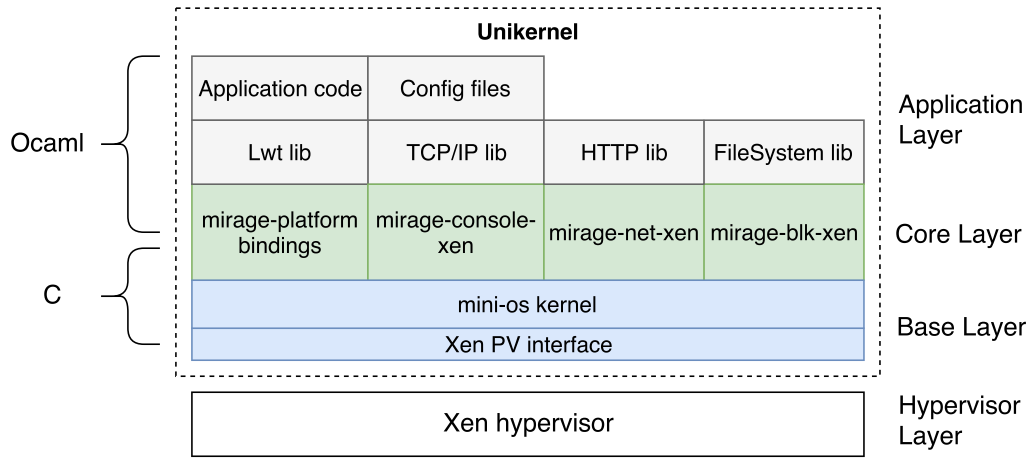


Figure 4.1: Software layers in MirageOS unikernels

A virtualised host that supports MirageOS unikernels is composed of four software layers, which are illustrated in Figure 4.1: the application layer, the platform bindings and drivers layer, the base layer, and the hypervisor layer which, we describe in the following paragraphs.

4.3.1 Application Layer

In this layer, we include all the application code that a unikernel requires to fulfil a particular task. For instance, if the unikernel we are building is an HTTP server, then, in this layer, we would add the code that deals with the incoming/outgoing requests. In addition, we add the configuration files and any high-level OCaml libraries, such as a TCP/IP library, that the application requires.

4.3.2 Core Layer

The second layer includes the core libraries for MirageOS. There are two types of libraries in this layer: the platform libraries and the drivers libraries. The first type contains the platform-specific code for building the unikernel, which is called the *mirage-platform*. The *mirage-platform* library contains the OS bindings for MirageOS, which can be accessed via the *OS* OCaml module. This module provides functionality for the Unix platform where MirageOS applications are executed as normal OS executables.

The *OS* module also supports the Xen hypervisor and provides the runtime for running MirageOS unikernels as Xen VMs.

The second type of library includes the libraries that support the drivers needed for unikernels. There are platform-specific libraries for each driver. For instance, *mirage-net-xen* is a Xen-specific driver that is developed in OCaml and supports the Xen paravirtualised split model for device drivers. More precisely, this library contains the Xen netfront ethernet device driver for MirageOS that enables an OCaml application and, by extension, a unikernel to read/write ethernet frames using the Netfront protocol.

4.3.3 Base Layer

This layer includes low-level libraries that MirageOS uses such as the mini-os kernel alongside low-level Xen paravirtualised primitives. MirageOS utilises the mini-os kernel to build unikernels that can be deployed as VMs on top of the Xen hypervisor. This kernel provides all the required functionality to build a guest OS and then boot it as a VM. The main functionality that MirageOS uses from mini-os is the access to the Xen paravirtualised interface where hypercalls can be initiated towards the Xen hypervisor.

4.3.4 Hypervisor Layer

The hypervisor layer acts as an abstraction for multiplexing physical resources such as the memory, storage, CPU, and network to higher layers. For VirtusCap, we use the Xen hypervisor. Section 2.5.4 provides background information regarding the Xen hypervisor and the XSM security framework.

4.3.5 VirtusCap Architecture

In this subsection, we discuss the high-level view of the VirtusCap architecture and leave the implementation-specific discussion for section 4.4. The VirtusCap architecture comprises of two main parts. The first part contains the unikernel and the changes we have made to each unikernel layer discussed in the previous subsections. In addition, it contains the changes made to Dom0. The second part is the hypervisor and the changes that we have made to the hypervisor. These changes mainly support the

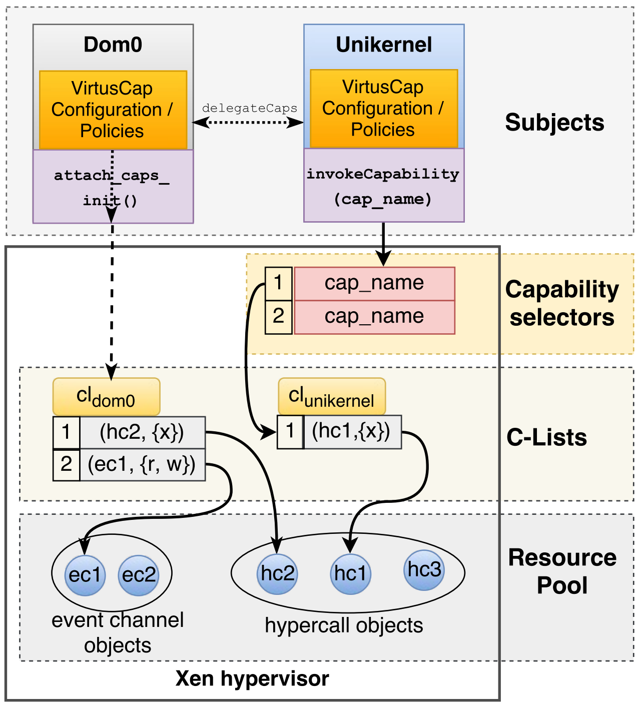


Figure 4.2: VirtusCap architecture. A capability list cl_s exists for each subject. In the capability list, we store the reference to the object alongside the access right. For instance, Dom0 is assigned the capabilities to communicate via event channels ec1 and execute hypercall hc2. Dom0 can then grant capabilities to other unikernels. The unikernel can only execute hypercall hc1 that exists in its capability list.

capability-based access control mechanism proposed in this thesis. Figure 4.2 depicts the VirtusCap architecture, which we discuss in the following paragraphs.

VirtusCap provides the ability to subjects such as the Dom0 and unikernels to add specific configuration and policies. Dom0 is preconfigured during the booting process with the capabilities required to function properly. The objective of Dom0, then, is to delegate capabilities to the unikernels. Therefore, only the capabilities that Dom0 has can be used for unikernels. For instance, a system administrator might want to prevent access to a resource from all unikernels or to use only a subset of rights available to unikernels. This ensures that the Dom0 configuration is tailored to the needs of the cloud infrastructure.

Unikernel developers can use VirtusCap configuration files to express security policies for unikernels. The security policies specify which resources the unikernel is allowed to access and what unikernel behaviour is permissible. If administrators or unikernel developers want to make any changes to the security policies, the unikernel is recompiled and deployed with the new set of security policies. Unikernels need to include the *mirage-virtuscap* library for unikernel developers to have access to the VirtusCap interface.

The Xen hypervisor includes the VirtusCap ACM, which enforces the security policies specified in the unikernel and does not allow any elevation of privileges while the unikernel is running. For instance, in order for a unikernel to gain a new capability, first, the capability has to be added to the C-List and the unikernel has to be compiled with the new security policy allowing access to the resource that the capability references. When a unikernel developer attempts to add a capability to the configuration of a unikernel that is not part of the unikernel's C-List, then the access control module will not allow the booting of the unikernel or it will boot the unikernel with reduced authority. If the unikernel is allowed to boot, then the privileges of the unikernel can only be reduced or remain the same. There is not a method to amplify the privileges of the unikernel. Thus, not only we are concerned with access control policies in a virtualised host but with the application security policies as well, which follow the principle of least privilege [19].

Capabilities are 128-bit record in a hash map structure that is efficient and can cater to a large number of capabilities. A capability record includes the IP address of the unikernel, the resource ID, and the random number generated for the capability ID.

In order for VirtusCap to provide sound security for cloud workloads, it requires to protect the integrity of the capabilities and the capability lists for each unikernel. We can be assured of the security only if we properly protect the capabilities. Our security design for capabilities is inspired by EROS [110], which introduces the partitioning of capabilities from data and the kernel protection of capabilities. In a similar way, we separate capabilities from data and rely on the hypervisor to protect capabilities. Hence, we create hypervisor-protected capabilities for unikernels.

Figure 4.2 depicts our current VirtusCap architecture. As we can see, Dom0 is assigned the capabilities to access event channels ec1, and hypercall hc2. Dom0 can assign capabilities to other VMs or unikernels. In our figure, the unikernel can only access hypercall hc1. Access control decisions are illustrated in Figure 4.3.

Dom0 is assigned to capabilities using the following process:

1. Xen hypervisor function checks all the resources that are available.
2. Checks which resources need to be available to capability according to the Dom0 configuration.
3. Generates 64-bit random numbers using the RDRAND instruction [54] to be used as capability ids.
4. Stores the capability id, the corresponding resource id and the rights in the capability data structure.
5. All of the capability data structures are inserted into the CList of Dom0
6. Xen hypervisor continues with the booting process of Dom0.

It should be noted that the reason we have used the RDRAND instruction is that there is no other form of randomness available during the booting process of the Xen hypervisor. Of course, we could add other sources of randomness that are similar of Linux's /dev/random. However, this approach is out of scope of this thesis since it would require additional design and development effort. We assume that the RDRAND instruction functions according to specification and the Digital Random Number Generator (DRNG) [54] hardware is not compromised.

4.4 Implementation

In this section, we first discuss the implementation of a capability-based access control module for XSM. Then, we discuss the implementation of a capability library for MirageOS. Finally, we describe unikernel policies and how VirtusCap makes access control decisions.

4.4.1 VirtusCap ACM

The XSM [31] is a generalised security framework to which we can add other access control modules. Currently, there are three main security modules that have been included in XSM: Flask [121], ACM (sHype) [103], and dummy. Since Xen 4.2, only

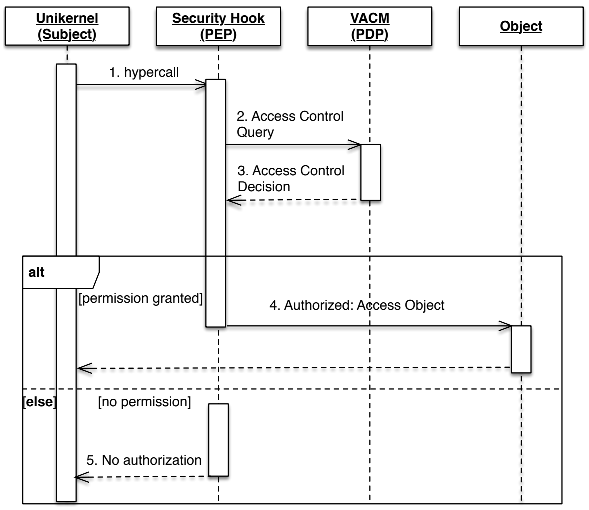


Figure 4.3: Access control decisions for VirtusCap access control module (VACM). A unikernel acts as a subject and issues a hypercall to access a virtual resource, which is represented as an object in this diagram. A security hook function sends an authorisation query to VACM to determine whether the unikernel has the capability to access the virtual resource (object). If the VACM decides that the unikernel has the capability capability, then permission is granted. Otherwise, if the authorisation fails, which means that the unikernel does not have the capability to access the object, then an error message is sent back to the unikernel stating that permission is denied.

the Flux Advanced Security Kernel (FLASK) security module is used. sHype is not included, as it is no longer maintained. Related background information regarding the Xen hypervisor and XSM can be found in subsection 2.5.4.

We have added a new security module to the XSM called VirtusCap ACM, which focuses on unikernels. VirtusCap ACM uses capabilities as the only access control mechanism for unikernels. The XSM uses security hooks that manage access to Xen resources. In principle, VirtusCap could be used as an access control mechanism in other hypervisors such as KVM. However, for this work, we focus on the Xen hypervisor as our selected implementation platform for our access control system and leave integration with other hypervisor platforms for future work.

An issue that we have come across while integrating capabilities with Xen is that there is no consistent method of accessing resources. Xen provides different interfaces to access resources. There is a hypercall interface for VMs, a shared memory, and an event channel interface. Additionally, recent Xen versions have introduced *libvchan* [140], which implements a higher-level protocol built on top of shared memory. *libvchan* is used as a bidirectional communication channel between services executing in different VMs or unikernels. Hence, there is a need to implement an access control module that protects each Xen resource with a unified interface. Currently, we are using capabilities to secure hypercalls and event channels.

First, hypercalls resemble the system calls of regular OSes and are the primary method that regular VMs and unikernels can use to gain access to virtual resources. By securing hypercalls, we can isolate the virtual resources that a unikernel can access. We create capabilities for each hypercall and attach each capability in a unikernel configuration. The unikernel can use any capability that is listed in the configuration.

Second, event channels are used as event notification mechanisms that resemble interrupts in OSes. We use capabilities to secure event channels and limit communications among unikernels. Any unikernel that has that particular event channel capability in its configuration can communicate with another unikernel. Whenever there is an event channel operation, we verify whether the unikernel that requested the operation has the event channel capability in its capability list data structure in the hypervisor.

VirtusCap ACM uses three stages to secure hypercalls. First, the hypervisor creates hypercall capabilities and attaches them to Dom0, which is the first booting VM. Second, Dom0 gives access to capabilities that are part of the Dom 1 configuration. Third, the hypervisor performs access checks for all the hypercalls that VMs invoke. Similarly, we use three stages to secure event channels. First, the hypervisor creates a capability and attaches it to an event channel object. Second, Dom0 gives a capability to the VM or unikernel we have created. Third, the hypervisor checks if a capability exists for all event channel operations.

We have added three hypercalls to Xen to facilitate management of capabilities. First, we added the *virtuscap_create* hypercall, which creates a new capability entry in a unikernel's capability list that resides as a data structure in Xen. Second, the *virtuscap_grant* hypercall can grant access to a particular virtual resource. Third, the *virtuscap_query* hypercall sends a request to the VACM to verify whether the unikernel has the capability to access the virtual resource. Fourth, the *virtuscap_revoke*, which revokes the capability from a unikernel.

4.4.1.1 Changes: Xen

The following changes were made to the Xen hypervisor to realise the VirtusCap ACM. The objective of these changes is to create another security module for Xen. This can be achieved by using VirtusCap specific code and creating the capability-based access control mechanism.

The initial change in Xen hypervisor was to adjust the configuration files and instead of building the XSM-Flask module, we now built the VirtusCap module. Hence, only one access control module can be executed. Our next change was to divert all calls from hooks inside the hypervisor to VirtusCap-specific functions that decide whether or not the access request is granted. Currently, VirtusCap can support all the hooks that XSM supports. Listing A.1 illustrates a partial list of these functions from the *xsm.h* file and their mapping to VirtusCap functions. Hence, when one of these operations is called, then the execution continues inside the corresponding VirtusCap function instead of the corresponding Flask function.

Each of the VirtusCap hypercalls was implemented in a similar fashion. As an example, we discuss the changes made to the Xen hypervisor to create the *virtuscap_query* hypercall.

Creating a new hypercall for Xen requires us to define the constant numbers of the hypercalls to the main Xen header file as depicted in Listing A.2. We have added the required definitions for the VirtusCap hypercalls and in the *entry.S* file to register the hypercalls in the hypercall table alongside the number of parameters. This is depicted in Listing A.3.

Then, inside the *hypercall.h* file, we include the signature of the functions that other parts of the code use.

Privileged domains can use hypercalls to request access to resources from VirtusCap ACM. The Listing A.5 provides example code on how this is achieved for the query hypercall. First, we add the function that will be called initially from the privileged domain in the *xc_private.c* file. Second, we create the implementation for query operation function in the *xc_private.h* file, which includes the *xc_interface* as a parameter that binds the function to the *xc_interface*. The actual hypercall is constructed populating the hypercall data structure with the required information such as the hypercall number for the query operation, hypercall specific parameters, and the domain number from which this call originated. The *do_xen_hypercall* function provides the actual call for the Xen hypervisor to execute this hypercall.

Listing A.6 shows the implementation for the query hypercall and the security hook that diverts execution to the VirtusCap ACM. First, we lock the domain data structure to prevent other processes from being able to write it by executing the *rcu_lock_domain_by_any_id* function. Subsequently, the *xsm_virtuscap_query_op* security hook is executed, which evaluates whether the unikernel executing this hypercall is allowed to use this hypercall. This means that the unikernel needs to have the appropriate capability in its C-List.

The security hook code for the query hypercall is illustrated in Listing A.7. In this code listing we show how the access check is performed for the query hypercall. The code we added performs the following steps to check access for a resource.

Algorithm 1: virtuscap_query(): Query capability list of unikernel with a particular domain_id

```
  Input: domain_id, capId
  Output: access granted
  /* Search for the C-List according to the domain id of
     unikernel                                                  */
1 clist ← findClistByDomainId(domain_id)
  /* Search if the capability exists in C-List                  */
2 isCap ← findCap(clist, capId)
3 result ← checkAccess(capId, domain_id)
4 return (result)
```

1. Search for the correct C-List in the main tree structure that holds all the C-Lists according to the domain id of the unikernel.

2. Traverse C-List to see if there is a capability for the resource.

3. Check if the correct rights are assigned in the capability object.

4. The access decision result is then returned to the hypercall. If the result is access denied return operation not permitted (EPERM), otherwise return 0.

If a unikernel wants to execute, for instance, the query hypercall, then it needs the execute right to use the query hypercall. If the execute right is not present in its capability, then the access request fails and the result is returned to the hypercall function, which created an exception. Otherwise, the access request succeeds and returns the result to the hypercall function, which continues its execution.

4.4.2 VirtusCap CapLib

MirageOS links only the libraries that the application needs to use. The MirageOS architecture is illustrated in Figure 2.2. For the VirtusCap system, we decided to attach an OCaml library in every unikernel that gives access to the resources we protect using capabilities. As we have already discussed, capabilities are securely stored inside the Xen hypervisor. However, we need a compatible library for the unikernels to access the capabilities and bind them to a particular unikernel. This library supports our architecture by using a capability interface API that communicates the capabilities to the hypervisor. Then, the hypervisor decides according to the capability list for a particular unikernel whether access is granted to the virtual resource.

4.4.2.1 Changes: MirageOS

The changes for MirageOS are in the three layers of a unikernel. First, there is a simple capability API written in OCaml, which can be used by unikernel developers to port their application to a capability way of accessing resources. The application level code is contained inside the *mirage-virtuscap* library, which can be used by any unikernel and provides access to the capability API. Second, there are changes inside the *mirage-platform*, which provides the glue code from the low-level code to the high-level code used by the *mirage-virtuscap* library. Third, changes are made to the *minios-xen* library, which contains the functionality for the unikernel base layer.

The *mirage-platform* library contains the OS interfaces for the Xen hypervisor. This library includes the initial function when the unikernel boots and bootstraps the whole process using functionality from the *minios-xen* library. In addition, this library includes the bindings that connect the high-level OCaml code with the C code. We have added a file that contains the stubs for VirtusCap hypercalls. Starting from the OCaml code, the unikernel can initiate a VirtusCap hypercall, execute the stub code, and then the code inside the *minions-xen* library constructs the actual hypercall. The hypercall is sent to the Xen hypervisor to manage the capabilities of the unikernel.

These changes are written in C language and the objective here is to create a uniform low-level API that actually calls the four hypercalls in the *minios-xen* library that are used in VirtusCap. This is achieved by using the facilities OCaml provides to interface with code written in C, and vice versa. OCaml has a number of C header files to facilitate writing C code that operates on OCaml values. Listing A.8 depicts the OCaml interface code of VirtusCap that application level code in a unikernel can use. Each OCaml function in the *virtuscap.mli* file is marked as external and corresponds to a C stub function in the *virtuscap_stubs.c* file. This file includes the code to translate the OCaml values to C values and propagate the call to hypercall in the *minios-xen* library.

More precisely, the *minios-xen* library includes the following changes. First, the hypercall constant numbers are defined in the *xen.h* file in the same way as in the Xen hypervisor. Second, the hypercall functions are also created in the same way as in the Xen hypervisor. Third, the *virtuscap.c* file includes the main functionality for each hypercall, which calls the actual hypercall function in the hypervisor with the correct parameters.

4.4.3 Security Policies

In this section, we first discuss the policies that XSM-Flask uses and demonstrate how security policies can be authored in configuration files for unikernels. XSM-Flask can define security policies to secure access to resources that combine RBAC [42], TE [21], and MLS [15]. A security policy authored for XSM-Flask defines a security context that contains a user identity, a role, a type, or an MLS level. It uses one domain configuration parameter seclabel that defines the label of the VM. For example, the default policy distributed with Xen uses *seclabel='system_u:system_r:domU_t'* for a typical VM. In a system that includes hundreds or even thousands of unikernels, developing XSM-Flask policies for each unikernel can be very complex and prone to error. Hence, we need a simpler method to define security policies for unikernels. This can be achieved by integrating capabilities with unikernels as the main access control mechanism.

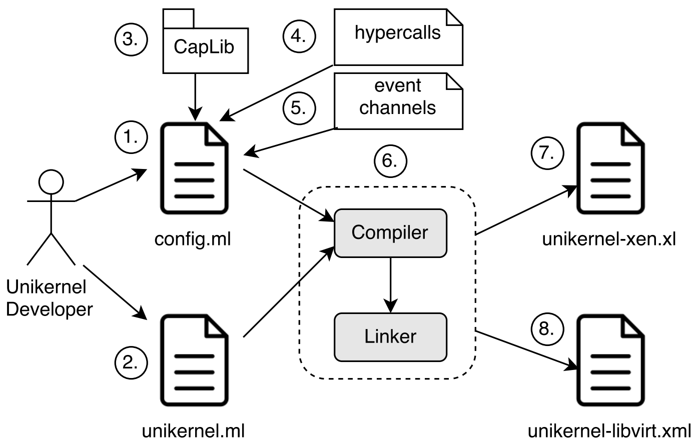


Figure 4.4: Workflow for constructing a unikernel with capabilities.

For a unikernel to use capabilities, we include the *mirage-virtuscap* library during the configuration phase of the unikernel. If this library is not present during the construction of a unikernel, then the unikernel cannot access capabilities and will not have any access privileges. This means that the unikernel will not have access to resources. This is enforced by the hypervisor, whereby each unikernel needs to have access to capabilities to function. The *mirage-virtuscap* library provides access to the four VirtusCap hypercalls related to capabilities.

Access control policies can be specified as part of the configuration of a single unikernel. These policies define all the resources the unikernel has access to and support capabilities that result in a fine-grained access control mechanism. Moreover, the policies could express whether a unikernel is member of a unikernel coalition. A complex cloud service can then be decomposed to a set of unikernels that reside inside a protected coalition. However, this feature is not currently implemented and we consider this as future work.

An example of an access control policy is illustrated in Listing 4.1. The main idea is to explicitly state the resources to which the unikernel may have access. In line 6, *virtuscap_hypercalls* expects a list of hypercalls that the unikernel requires in order to execute. In addition, *virtuscap_evtchn* in line 7 includes a list of domain IDs that the unikernel can communicate using event channels. The access checks in the hypervisor for both hypercalls and event channels will determine whether the unikernel is granted access to the resources. The *config.ml* file is the main configuration file for constructing unikernels in MirageOS.

```
open Mirage
let main =
  let libraries = ["vchan"; "virtuscap.xen"] in
  let packages = ["vchan"; "virtuscap.xen"] in
  (* configure access to hypercalls and event channels *)
  let virtuscap_hypercalls = ["evtchn_send"; "evtchn_unbound"; "evtchn_interdomain";
       "evtchn_status" ] in
  let virtuscap_evtchn = ["0"] in
  foreign
    ~libraries ~packages
    ~virtuscap_hypercalls
    ~virtuscap_evtchn
    "Unikernel.Client" (console @-> job)
let () =
  register "vchan_client" [ main $ default_console ]
```

Listing 4.1: MirageOS config.ml configuration file for unikernel.

A typical workflow for creating a unikernel integrated with capabilities is displayed in Figure 4.4 and includes the following steps.

1. The unikernel developer writes the initial configuration file *config.ml*.
2. The developer adds the *unikernel.ml*, which defines the behaviour of the unikernel. The development process is iterative, whereby the developer adds more functionality to the unikernel and adds configuration items as the need arises.
3. Add the *mirage-virtuscap* library in the configuration file so that we can use the VirtusCap API.
4. Add a list of hypercalls that we need to execute.
5. Add the event channels to which we require access to the configuration files.
6. Compile and link the unikernel with the new configuration using the MirageOS toolchain.
7. MirageOS builds a toolchain that creates a Xen specific VM configuration and a more generic libvirt configuration.
8. We can then boot the unikernel using either the xl file or the libvirt specific xml file as a regular Xen VM.

4.5 Security Evaluation

The evaluation in this section is based on the system and security model of section 4.2 and the security design principles of Saltzer et. al [104].

VirtusCap ACM component is a PDP that issues an accept/deny decision to access requests that are intercepted by the security hooks (PEP) in the Xen hypervisor. We establish an argument that shows that a subject cannot escalate its rights based on a small set of assumptions.

4.5.1 Assumptions

Limited Access [access]: We assume that a malicious user does not have physical access to the virtualisation host and cannot manipulate hardware in the cloud datacenter. An adversary cannot use side-channel attacks, such as time, resource or access analysis [148] to compromise the system and especially the hypervisor.

Trusted Hardware and Software [trustedhwsw]: We consider the hardware in cloud datacenters where the virtualised server is hosted to be trustworthy. Another assumption for VirtusCap is that we consider MirageOS and the OCaml compiler as trustworthy software and that they have not been tampered with.

Trusted Boot [trustedboot]: We consider that the hypervisor is booted creating a trusted execution environment using SecureBoot [6], a TPM device [126], and Intel TXT technology [58] that verifies the integrity of the hypervisor.

Hypervisor Integrity [hypintegrity]: We assume that the Xen hypervisor is integrity-protected during runtime following the rules outlined in [133].

4.5.2 Security Requirements

We require that the access requests are sent to the VirtusCap ACM by the security hooks without being tampered with. Further, we require that the C-List of a unikernel has not been altered by a malicious user, or a malicious software component. The goal here is to prevent the privilege elevation of a unikernel by adding capabilities to its C-List whereby gaining more access to resources.

4.5.3 Security Proofs

Lemma 4.5.1 (Access Requests in the VirtusCap ACM). *If the security hooks (PEP) have accepted an access request, then the VirtusCap ACM (PDP) has previously intercepted and received the access request, verified it, and issued an* ***accept*** *decision based on the subject's C-List, which is the base for this acceptance decision.*

We pursue the argument by back-tracking starting from an access request VirtusCap ACM receives.

Proof: As a subject S has received an access decision it must have been forwarded by the security hooks upon an accept decision from the VirtusCap ACM (PDP). The access decision must have been determined after verifying the access request and consulting the C-List of S and issuing an access decision.The VirtusCap ACM can only have issued an accept decision if a corresponding capability existed in the C-List of S. Both the VirtusCap ACM and the security hooks are protected from the adversary's intrusion by the assumptions *[access]* and *[trustedboot]*. The security hooks and the VirtusCap ACM communicate their access requests and responses with integrity by the *[hypintegrity]* assumption. There is no tampering with the access requests and responses. Moreover, the VirtusCap ACM receives the same access request that the security hooks received from S. The access decision received by S corresponds to the same access request. The access request result received by S is the same submitted to the security hooks and issued a decision by the VirtusCap ACM which could only be forwarded if a capability existed in the C-List of S.

Finally, *[access]* condition assures that the access request can only be fulfilled through the VirtusCap ACM and that the adversary cannot access the hypervisor and virtualised host in a direct way. Consequently, the access request must have been fulfilled by the VirtusCap ACM after the verification and an accept decision. □

Lemma 4.5.2 (Subject's Rights are bound to the VirtusCap ACM). *If a subject S cannot escalate its rights R over resource E, it must have been granted the capability either to reduce its rights or retain the same rights.*

Proof: A subject has been granted a capability that is not escalating the rights over a resource. This means that, following Lemma 4.5.1., if a subject has been granted a capability that does not escalate the rights over a resource E then such a capability must have been added to its C-List. Otherwise, the VirtusCap ACM must have decided to not allow elevated access to E. Consequently, the security hooks would not have allowed access to E. □

4.5.4 Evaluation of Design Principles

The main goal of VirtusCap is to provide sound security protection. This has the requirement that our security architecture employs design decisions that will decrease the attack surface and ensure that a unikernel cannot escalate its privileges. Therefore, we want to make sure that unikernels cannot escape the confinement of capabilities.

One way to evaluate the security architecture of a system is by using a number of important design and evaluation criteria that Saltzer et al [104] first introduced to the security research community. Of course, we could provide a formal model approximation of our security architecture; however, this would take a great deal of effort to finish, and we leave it as future work. We now evaluate our security architecture according to Saltzer's design principles:

Economy of Mechanism: Security mechanisms should be as simple as possible. Economy of mechanism in VirtusCap is achieved by using a simple capability-based design that is implemented in 10% less LOCs than XSM-Flask, and by focusing on unikernels that provide a minimal attack surface instead of using VMs that contain a full OS.

Least Privilege: A subject should be given only those privileges needed to complete its task. We ensure that unikernels and, by extension, cloud services are running with the least privileges possible by carefully granting capabilities. This stems from the implementation of VirtusCap, our design choices that focus on granting unikernels only the capabilities they need, and the security analysis, where we prove that a unikernel cannot escalate its privileges.

Separation of Privilege: A system should not grant permission based on a single condition. Security mechanisms in VirtusCap are separated such that an access request from a unikernel passes through different stages. First, a security hook acts as a PEP, and second, the VACM acts as a PDP. Additionally, we facilitate separation of privilege by controlling inter-VM communication. Finally, our goal is not to place our trust in only one security mechanism.

Complete Mediation: All accesses to an object should be checked to ensure access is allowed. The principle of complete mediation is applied in layers of protection throughout the system by using security mechanisms that place restrictions on how capabilities are shared throughout our system. VirtusCap enables this by sending all the access requests that are coming from the security hooks to the VirtusCap ACM for evaluating, if access is granted or not.

Fail-Safe Defaults: Unless a subject is given explicit access to an object, access should be denied. In VirtusCap this principle is achieved by explicitly defined policies that only allow access to resources if the appropriate capability exists in the unikernel's C-List. Otherwise, access to the resource is denied.

Least Common Mechanism: Mechanisms used to access resources should not be shared. We minimise the interference of different unikernels by assigning capabilities only to share resources and communication channels that are explicitly stated in their security policies. This means that we enable secure sharing of information between two unikernels only when needed.

Open Design: Security of a mechanism should not depend on secrecy of the design or implementation. We intend to release the VirtusCap ACM[1] source code alongside related configuration and code additions to Xen hypervisor and the CapLib for MirageOS .

Psychological Acceptability: Security mechanisms should not make the resource more difficult to access than if security mechanisms were not present. In VirtusCap, we focus on minimising the attack surface by using unikernels and reducing the complexity of our architecture. VirtusCap is a capability-based access control mechanism that uses a minimal API to manage capabilities. In addition, we employ simple security policies that users can attach to unikernels. Keeping the balance between security and usability is of paramount importance to the research community. A system that is secure but very difficult to use will not be accepted easily by users. Furthermore, we could encounter a decrease in productivity due to a large number of user errors and misconfigurations [25].

[1]https://gitlab.com/gsfyrakis/virtuscap_project

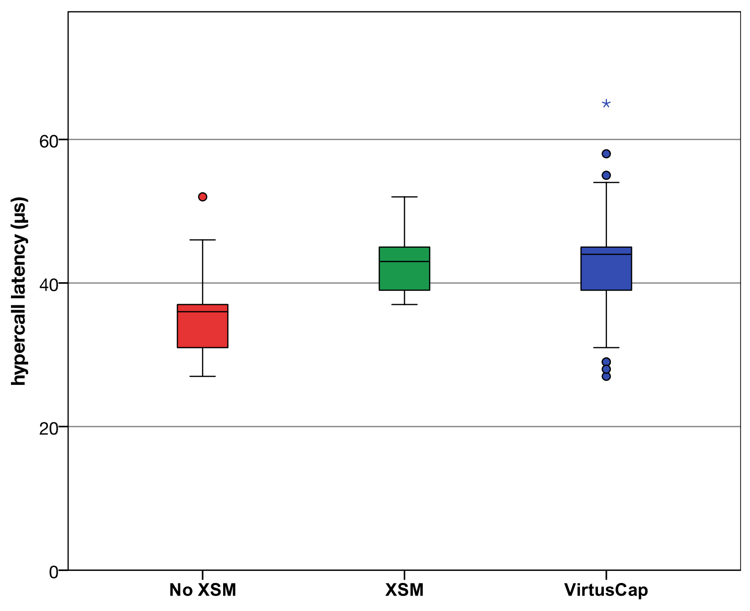
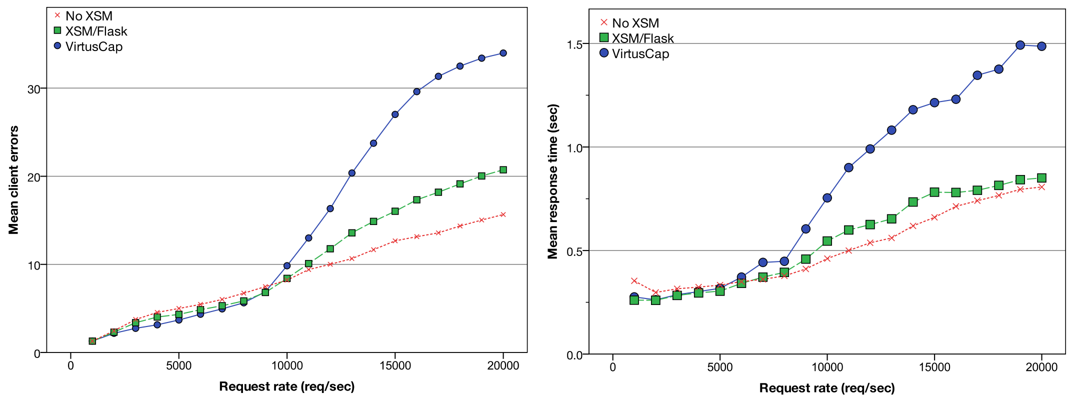


Figure 4.5: Null hypercall latency

4.6 Performance Evaluation

In this section, we evaluate the performance of our VirtusCap access control system. Our goals for this subsection are: 1) to evaluate VACM on a virtualised host using a set of performance metrics and investigate how VirtusCap behaves when we scale the architecture with hundreds of unikernels; 2) to identify performance overheads that might exist; and 3) to discuss how performance could be improved in a future version of VirtusCap. The first part of the performance evaluation discusses the performance of VirtusCap using a number of micro-benchmarks. The second part discusses the performance of VirtusCap using a case study that constructs two unikernels that communicate with each other.

We conducted experiments to measure the performance of VirtusCap using a commodity desktop computer as a virtualisation server, which consists of a 3.4GHz Intel Core i7-3770 CPU with 8GB of RAM and a 500GB hard disk. Dom0 in the virtualisation server is running Ubuntu 15.10 Linux OS with kernel version 3.19.0. Additionally, the virtualisation server uses a custom-built Xen hypervisor with version 4.6.4 and MirageOS with version 2.9.0 to build the unikernels. For evaluating the case study, we



(a) Average number of errors **(b) Average response time**

Figure 4.6: Comparison of errors and response time for a unikernel serving a single HTML page.

used MirageOS version 3.5.0, and version 4.0.0 of the ocaml-vchan library.

The unikernels in the virtualisation server are assigned IP addresses from a DHCP server running in Dom0. Dom0 has a static IP address and is assigned 1GB of RAM and 4 VCPUs. The other four VCPUs are assigned to unikernels in order to isolate their workload from Dom0. Each unikernel is assigned at least 32MB of RAM and at least 1 VCPU depending on the performance test we are conducting. In addition, a virtual network bridge (xenbr0) is connected to a physical Ethernet network link.

We evaluated the performance of unikernels using three test cases: 1) baseline (TC0), 2) XSM-Flask (TC1), and 3) VirtusCap (TC2). For the baseline configuration, we compile Xen without any access control mechanism enabled. Moreover, TC1 uses XSM-Flask as the access control mechanism under test. Finally, TC2 utilises VirtusCap ACM as the access control mechanism.

4.6.1 Micro-benchmarks

For our first micro-benchmark, we created a null hypercall to evaluate the execution time plus the time to grant access to the hypercall if any. Figure 4.5 depicts the results of the experiment. Executing the null hypercall without any access control provides the least latency with $34.37\mu s$ at 95% CI $[34.17, 34.57]$. The XSM-Flask test case has slightly more latency with $41.94\mu s$ at 95% CI $[41.75, 42, 13]$. VirtusCap has slightly more latency with $42.39\mu s$ at 95% CI $[42.19, 42.60]$.

For our second experiment, we used a unikernel as an HTTP server that hosts a simple HTML page. The test was performed from another VM in order to eliminate the local network as a bottleneck and we repeated the test 1,000 times. In this way, we could measure the response and number of client errors using Httperf [92]. Second, we used a simple unikernel that prints a message to the console and then sleeps in order to measure the boot time of a single unikernel. This test was repeated for 1,000 times. Third, we measured the boot time when we simultaneously booted 50, 100, 150, and 200 unikernels for each of our test cases. We repeated the scalability test ten times for each test case.

Figure 4.6 shows the results for both the client errors and response time for a unikernel that serves a simple HTML page. We have minimal errors for all three test cases until we reach approximately 8,000 requests/sec. An omnibus analysis of variance (ANOVA) shows that, up to that point, the difference between the mean client errors is not statistically significant, $F(2,2997) = 2.015, p = .13$. After that point we see a slight increase in error for our first two cases. VirtusCap outputs more errors while we increase the request rate. At a rate of 20,000 requests per second, a mean $0.1\%(21)$ of the requests of the XSM-based system fail compared to a mean $0.175\%(35)$ of the VirtusCap-based system. At a significance level of .05 and this request rate, the difference in proportions between successful and failed requests is not statistically significant, $p = .063 > .05$ (Fischer Exact Test).

We further investigated this issue and we determined that we would need to implement a fast cache system for the access requests in a similar fashion to the cache that the XSM-Flask uses. For response times, we experience the same behaviour. When we reach 8,000 requests/sec VirtusCap reaches a saturation point.

Figure 4.8 depicts the results from measuring the boot time of a single unikernel. Booting a unikernel without any access control provides the best overall performance with 39ms boot time, 95% CI $[38.4, 39.5]$. The XSM-Flask is slightly slower, booting at 47.4ms, at 95% CI $[47.1, 47.8]$. VirtusCap boots at 49ms, at 95% CI $[48.3, 49.7]$. The differences between the mean boot times are statistically significant, $F(2,2997) = 370.141, p < .001, \eta_p^2 = .198$.

Figure 4.7 shows the boot times for a workload in which we create a number of unikernels. When we are not using an access control mechanism, boot times are almost linear. However, when we add an access control mechanism, there is a slight overhead,

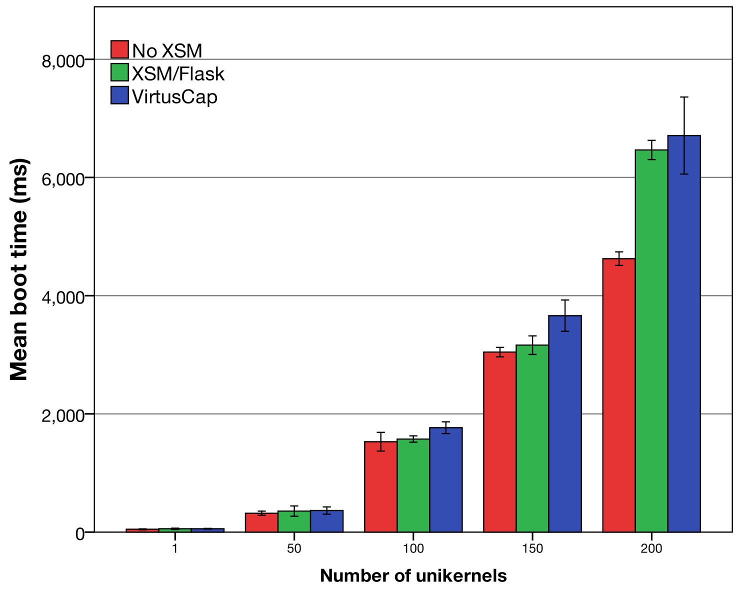


Figure 4.7: Boot times for multiple unikernels.

especially when we boot more than 150 unikernels. For VirtusCap, the boot time increases more when we boot more than 100 unikernels. Currently, a cache for access operations is not part of VirtusCap ACM.

4.6.2 Case study: unikernels communicating using the libvchan interface

In this subsection, we evaluate the performance of VirtusCap using a more realistic workload. We create inter-domain communication channels using libvchan, which we discussed in subsection 2.5.4.3. We outline the steps required for two unikernels to communicate with each other using the `libvchan` interface and discuss the performance experiment constructed for VirtusCap. Finally, we analyse the results from the performance experiment.

In this test scenario, we use the `libvchan` library that creates an inter-domain communication channel. Unikernels use the `ocaml_vchan` library to gain access to the `libvchan` interface, when using MirageOS.

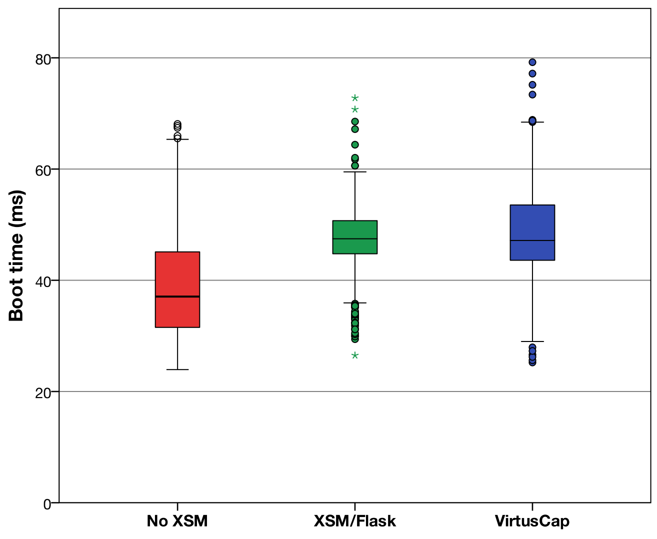


Figure 4.8: Boot times for a single unikernel.

Test scenario We now outline the steps for the two unikernels to communicate with each other when we use VirtusCap. Each unikernel is granted the required capabilities for executing the libvchan hypercalls outlined in subsection 2.5.4.3:

1. The unikernel with name `vchan_server` is booted with the capabilities required for executing the libvchan related hypercalls and acts as a server.
2. The `vchan_server` registers the name of the communication channel in `xenstore`.
3. The server domain id is read from xenstore.
4. The `vchan_server` creates a port for vchan and a capability name for the port and waits to read from the client unikernel.
5. The unikernel with name `vchan_client` is booted with the required capabilities for executing the libvchan related hypercalls and acts as a client.
6. The `vchan_client` registers its name in `xenstore`.
7. The `vchan_client` reads the name and id of the server unikernel.
8. The `vchan_client` requests access to the server's event channel port.

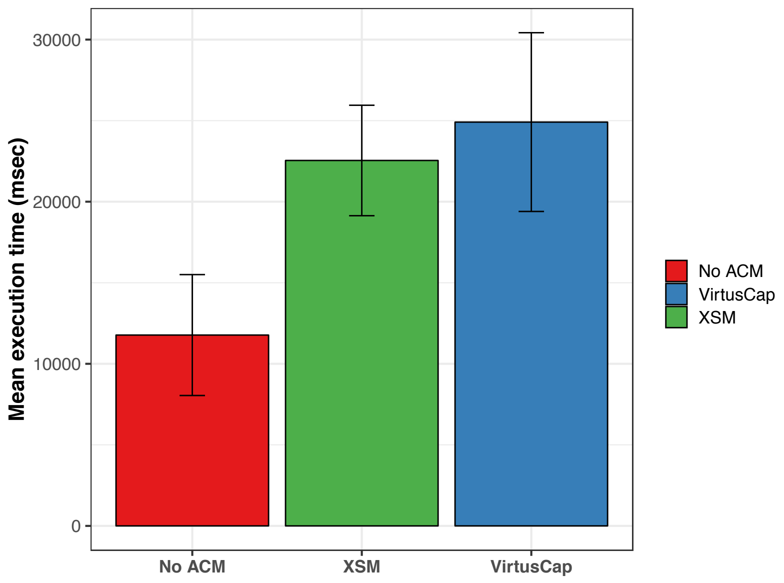


Figure 4.9: Mean execution time for macro-benchmark using libvchan library.

9. The `vchan_client` establishes a communication channel with the server unikernel by creating a port with the id of the server unikernel.

10. The `vchan_client` requests access to write a message to the shared message ring buffer and requests access to send the message.

11. The `vchan_client` writes a message to the ring buffer and sends it to the server unikernel.

12. The server unikernel reads the message send from the client unikernel and closes the connection.

First, we measure the total execution time of the above test scenario. The experiment was executed 50 times for each test case and the means for the execution time ware calculated. We measure the execution time it takes for two unikernels to establish a communication channel and send a message. The results are depicted in Figure 4.9. The results show that the test scenario that exhibits the best overall performance is when there is not any access control executing at 11773ms, at 95% CI $[8043, 15503]$. The second mean execution time happens, when we use the XSM-Flask ACM with 22543ms execution time, at 95% CI $[19398, 30425]$. When using VirtusCap there is

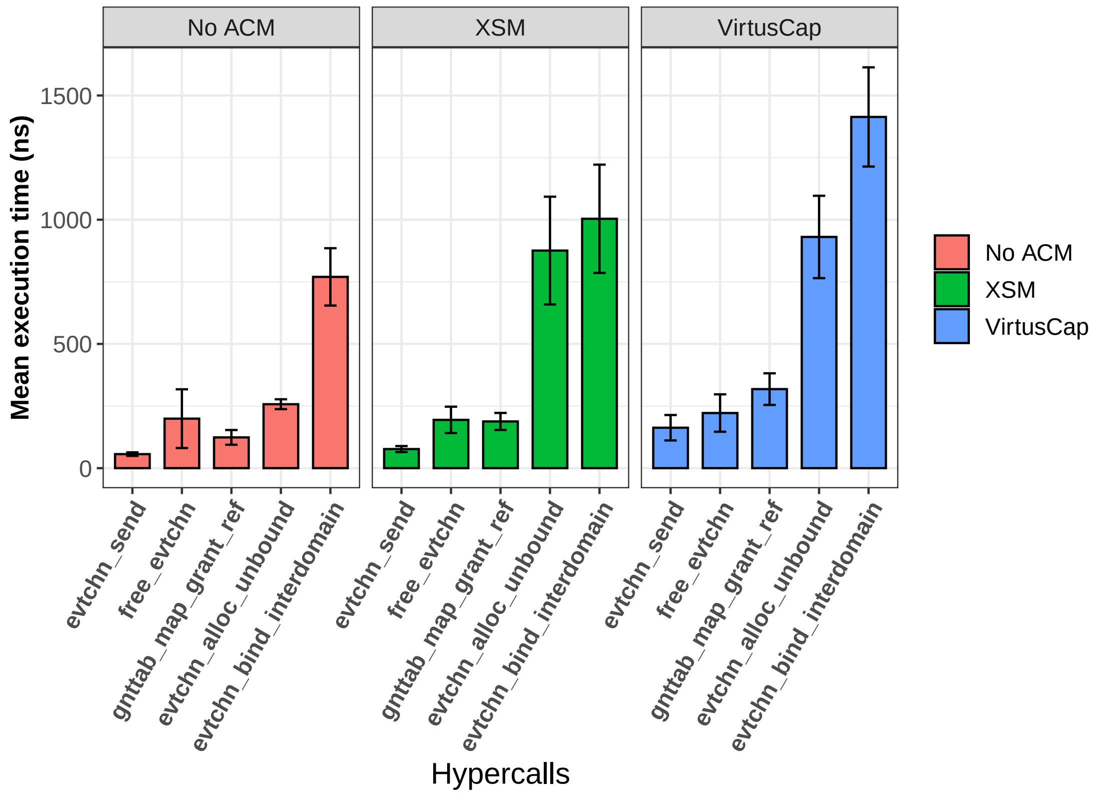


Figure 4.10: Mean execution time for hypercalls using libvchan.

a slight overhead executing at 24911ms, at 95% CI [19136, 25951], which is the behaviour we have witnessed in experiments discussed above. Second, we measure the execution time that each hypercall related to libvchan takes to perform its task. Figure 4.10 illustrates the performance results per hypercall for each test case. The result show that overall the test case with least execution time is the one without any access control mechanism. The second best test case is the one using the XSM-Flask ACM to determine if executing the hypercalls is allowed. The last test case exhibits comparable performance overhead in all hypercalls that were evaluated. We now discuss the results for each hypercall.

The `evtchn_send` hypercall executes at 56.5ns, at 95% CI [54.2, 58.7] when there is no access control mechanism used. For the second test case the hypercall executes at 76.8ns, at 95% CI [72.9, 80.6] and for the third test case the hypercall executes at 162.7ns, at 95% CI [146.5, 178.9]. The `free_evtchn` hypercall is used when the event channel needs to be closed. This hypercall executes at 199.2ns, at

95% CI [161.8, 236.6] for the test case. Regarding the second test case the hypercall executes at 194.3ns, at 95% CI [177.6, 211.1]. The final test case for this hypercall executes at 221.8ns, at 95% CI [198, 245.6].

The third hypercall we evaluated is the `gnttab_map_grant_ref`, which is used when we want to request to map the memory pointed by the grant reference. For this hypercall the execution time for the first test case is 123.8ns, at 95% CI [114.4, 133.2]. The second test case for this hypercall executes at 188ns, at 95% CI [177.2, 198.8]. The third test case that uses VirtusCap executes at 318ns, at 95% CI [298, 338.2]. The `evtchn_alloc_unbound` hypercall is used when we want to create a new event channel port. The execution time for the first test case is 257.3ns, at 95% CI [251.1, 263.7]. The second test case executes at 875.7ns, at 95% CI [807, 944.3] and the third test case at 930.5ns, at 95% CI [878.1, 983]

The final hypercall called `evtchn_bind_interdomain` is used to connect with the remote event channel. The first test case for this hypercall executes at 769.8ns, at 95% CI [733.4, 806.4]. The execution time for the second test case is 1003.7ns, at 95% CI [934.8, 1072.6] and for the final test case is 1413ns, at 95% CI [1350, 1476].

One of the main limitations of our performance evaluation was that we were unable to test the scalability of our solution beyond 200 unikernels. The reason is that our current virtualisation server cannot accommodate more than 200 unikernels due to limited memory. However, we consider it future work to test VirtusCap with thousands of unikernels.

4.7 Conclusion

In this chapter, we proposed a mechanism for controlling access to resources that reside in virtualised hosts. We introduced the first capability-based access control mechanism that focuses on unikernels. By using capabilities, we provided a mechanism with a minimal attack surface that enforces the principle of least privilege.

The benefits of VirtusCap stem from the small attack surface of MirageOS and the hosted unikernels, as well as from the robustness of the OCaml implementation. The key component is a new reference monitor, the VirtusCap ACM. VirtusCap extends XSM and creates a capability-based access control mechanism that focuses mainly on unikernels. The reference monitor is complemented by a MirageOS library that is linked into every unikernel. This library provides an API for the application layer of a unikernel to use capabilities. The benefit of this architecture is that each unikernel can be considered as a subject, and its capabilities can be explicitly declared and enforced in the configuration file of the unikernel. The main resources that are controlled include hypercalls and event channels. In effect, the new system makes it possible to tighten the security of unikernels considerably, with similar performance to the XSM-Flask. We obtain an approach that adheres to the POLP with a minimised trusted computing base.

The main limitation of this architecture is that we still include the attack surface of the Dom0 in the TCB, which includes a full OS that manages VMs. Even though the attack surface of regular VMs is kept minimal by using unikernels, there is still the attack surface of Dom0. Another limitation is that for this research prototype, we focused only on the hypercalls and event channel resources. In a production-ready version of VirtusCap, all the resources of the virtualised host would be regulated.

4.7.1 Future Work

Future work for the architecture and implementation side includes the ability to scale the architecture for thousands of unikernels and to improve the performance of VirtusCap by creating a fast cache for access requests. Another area of future work is extracting capability graphs from a set of unikernels and evaluating them against access control policies. In addition, we aim to further minimise the attack surface of Dom0 and use a privileged unikernel to manage the resources of a coalition of unikernels. Finally, another promising avenue would be to target KVM for our VirtusCap architecture using Solo5 unikernels [138]. Hence, we could create a multi-platform capability-based access control mechanism that is tailored for applications or services built using unikernels.

Chapter 5

Protecting Unikernels From an Untrusted Cloud

In this chapter, we establish a security architecture that protects unikernels from software and physical attacks. This is achieved by integrating unikernels with Intel SGX and executing the unikernel code inside a secure enclave. Whereas existing solutions for protecting applications from an untrusted cloud focus on OS-based solutions, our approach focuses on virtualisation, which yields improved isolation among applications.

5.1 Introduction

Cloud computing relies on a cloud infrastructure to deliver the required services to cloud tenants. A prominent concern among tenants is that a CSP is trusted to access a tenant's data. Privileged users such as system administrators of a cloud infrastructure can use the OS kernel to launch attacks to other software components that tenants use [27]. In addition, vulnerabilities in hypervisors provide an attack vector for malicious insiders [100] to gain access to sensitive information. Hence, there is a need to shield access to a tenant's code and data.

Even though unikernels provide a minimal attack surface without the need for a full OS, there is still the concern for the protection of a unikernel's data from a privileged user. Existing approaches have focused mainly on the OS to provide the platform for securing a user's applications using containers [9], library OSes [130], and micro-containers [113]. All of the aforementioned approaches leverage Intel SGX for achieving protection against software and hardware attacks. By using Intel SGX these approaches need to sanitise the system calls that originate from an enclave, from privileged software and vice versa. Currently, there has been little research on how to leverage Intel SGX with virtualisation technology that also incorporates a minimal attack surface to protect a tenant's code and data from an untrusted cloud. Consequently, it is our primary goal to achieve this by creating a security architecture that integrates unikernels with Intel SGX, called *UniGuard*.

5.1.1 Contribution

The goal of this chapter is to study the protection of unikernels in virtualised infrastructures from privileged software and certain hardware attacks. We aim at preventing unikernels leaking sensitive information during their execution.

We propose a novel security architecture, called *UniGuard*, for protecting unikernels deployed in virtualised infrastructures. UniGuard is capable of creating a TEE for unikernels that protects their sensitive computations. In particular, *UniGuard* uses Intel SGX to protect unikernels by executing the sensitive computations inside a secure enclave. We employ a manual partitioning approach for a unikernel. The unikernel is separated in two parts: the trusted part that executes in the enclave and the untrusted part. By partitioning the unikernel in such a way we can choose which computations to be executed in the unikernel. If the whole unikernel is added to an enclave, we wouldn't be able to optimize the execution of the unikernel. In addition, there is no limit in the number of libraries added in a unikernel, so it would be cumbersome to add the whole unikernel inside an enclave.

5.1.2 Outline

The remainder of this chapter has the following structure. We first outline the system and the threat model. Second, we explore the architecture of UniGuard and our design choices. Third, we analyse the implementation details for this security architecture. Fourth, we evaluate the security of our system using a semi-formal method and using security testing. Fifth, we evaluate the performance of *UniGuard*.

5.2 System and Threat Model

In the following subsection, we outline the system and threat model of the *UniGuard* architecture.

5.2.1 System Model

For our system model we consider a single physical host that supports a hypervisor such as KVM.
The UniGuard system includes two types of an entity. First, there are the three user entities that are actors in the system. Each of these users operate on different levels on the architecture. Second, there are software components that are created or ported for the *UniGuard* security architecture to support SGX. These components are discussed in subsection 5.3.5. The system model for UniGuard includes the following users:

- **Unikernel Developer**: This user is responsible for developing the application code and writing the configuration for the unikernels. The developer uses the resources explicitly or implicitly provided by the hypervisor and the OS, and especially the software Intel provides which is the SGX SDK, PSW, and the SGX driver. In addition, the developer can use MirageOS libraries to construct a unikernel out of a set of libraries and the application code and data.

- **Host Administrator**: This user manages the resources of a single physical host and grants access to other users such as the developer users.

- **Tenant Administrator**: The goal of this user is to administer the resources of the tenant and allocate them accordingly.

5.2.2 Threat Model

UniGuard adopts a typical threat model for software applications that use SGX. There are four components that we consider untrusted. First, the hardware that exists outside the CPU packages. Second, the system software such the OS, the hypervisor alongside other system software. Third, other applications executing in the same physical host or inside other VMs, including other enclaves. Fourth, we do not trust any software

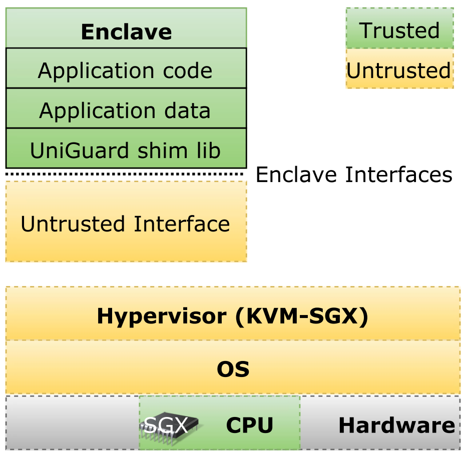


Figure 5.1: UniGuard architecture

components inside a unikernel. We only trust the CPUs and any code that is executing inside the enclave.

The Intel SGX driver is used by the hypervisor to provision the EPC memory, however we do not trust the driver. The AESMD service is a software component that must be started in order for applications to launch enclaves or enabling remote attestation.

UniGuard does not protect from side-channel attacks, denial-of-service (DoS) attacks and controlled-channel attacks [145]. These are vulnerabilities that are shared by all applications that are built on top of SGX.

5.3 System Architecture

In this section, we discuss the architecture of UniGuard. First, we present background information regarding KVM and unikernels. Second, we define the different layers of the architecture. Third, we discuss the components of the architecture. Our research focuses on creating a security architecture that integrates Intel SGX with unikernels. Figure 5.1 depicts a high-level view of the UniGuard architecture.

5.3.1 System Design Choices

Designing a security architecture based on SGX needs to account for four design choices and their tradeoffs. First, the amount of functionality to include inside the enclave. On one hand, there are works such as Haven [13] which they include a large percentage of the application code and supporting OS functionality inside the enclave. On the other hand there are works such as SCONE [9], which wraps a thin layer for the functionality inside the enclave. Even though including more code inside the enclave increases the TCB it minimises the attack surface of the interface which sits between the enclave and the OS or the hypervisor and its complexity.

Second, the complexity of the shielding support for the SGX. Although, SGX can protect an application from malicious privileged software, it cannot protect an application that requires functionality from the OS or the hypervisor. SGX does not protect from Iago attacks that can compromise an application using an unchecked system call. Therefore, a sanitising mechanism is required to verify or reject inputs from the OS or the hypervisor. The complexity of the enclave API directly affects the complexity of the shielding mechanism.

Third, the complexity of applications that support SGX. Applications range from simple services that execute a cryptographic library inside the enclave which includes a minimal shim library [10] to more complex applications which require a large number of system calls.

Fourth, how the application is partitioned. There are various ways of partitioning applications. One way is for an application to have multiple enclaves where they can communicate with each other. Another is to include more functionality outside of the enclave. For UniGuard we focus on having a one-to-one relation between a unikernel and an enclave. This is facilitated by the fact that unikernels include minimal functionality and in most situations it would require only one enclave. We consider support for multiple enclaves in unikernel as future work.

Even though, unikernels exhibit a minimal footprint and we could add the whole unikernel in an enclave we decided to use application partitioning for UniGuard. There is nothing stopping a developer to create a unikernel with a large number of libraries inside a unikernel. By partitioning the unikernel we gain the flexibility of choosing how to shape the interface between the untrusted and trusted parts of the application.

5.3.2 UniGuard Architecture

The main components of UniGuard include the *ukvm* component which we have ported to use Intel SGX according to a configuration option for a unikernel.

The architecture of UniGuard includes the following components: the KVM hypervisor, the unikernel layers, the secure enclave, and the wrapper interfaces.

We use a version of KVM and QEMU that supports Intel SGX. The main resource that SGX uses is the EPC which is a limited resource in a physical host. Section 3.2.2.3 provides background information on SGX and how EPC resources are managed. Currently, only static allocation of EPC resources is allowed. However, with the arrival of CPUs that support SGX2 dynamic memory allocation is possible. The system administrator needs to specify the size of the EPC as parameter when starting the VM.

In the following subsections, we first present information regarding the SGX support that the KVM hypervisor currently provides. This discussion we drive our argument for the importance and novelty of this contribution. Second, we discuss the support of MirageOS unikernels for the KVM hypervisor. Third, we introduce the components of the UniGuard architecture.

5.3.3 KVM-SGX

KVM is a Type II hypervisor where VMs can be deployed. For more general background information regarding KVM we refer the reader to subsection 2.5.5. We have chosen the KVM hypervisor over other hypervisors to develop UniGuard mainly because of the KVM's support for unikernels that use hardware-assisted virtualisation rather than paravirtualisation. The MirageOS unikernels that support Xen only support paravirtualisation which cannot be easily integrated with the SGX architecture. SGX requires the supervisor instructions to be executed in ring 0, which is not supported when using paravirtualisation, which places the guest OS kernel in ring 1 as seen in Figure 2.6. If an SGX supervisor instruction is executing in such a configuration, then a general exception is triggered to the hypervisor. Although, it would be possible to catch the exception in the hypervisor and perform the instruction on behalf of the paravirtualized guest, it would require to reengineer the Xen hypervisor to support all the supervisor instructions. That is the main reason that the Xen hypervisor only supports hardware-assisted virtualisation for SGX.

Another way this could be solved, is to add support for hardware-assisted virtualisation in unikernels that are deployed in the Xen hypervisor. This can be achieved by extending the mini-os unikernel base to support hardware-assisted virtualisation. Recently, there is a proposal for the *Unicore* [65] project that aims to include support for hardware-assisted virtualisation and provide a holistic unikernel base.

We chose to use the KVM hypervisor to implement UniGuard, since it has already support for unikernels that use hardware-assisted virtualisation. The security mechanism introduced with UniGuard is based on a version of KVM that supports SGX. We will refer to this version of KVM as KVM-SGX for the remainder of this thesis.

The way that KVM-SGX works is that it follows a unified model where the SGX driver manages all the EPC pages. The main objective of KVM-SGX is to call the driver's API to manage SGX information such as allocating or freeing EPC pages. The only difference is that KVM-SGX is not able to call the driver's API directly, since that hosts without SGX-enabled CPUs won't load an SGX driver in the OS. Hence, if the APIs are called directly, KVM-SGX won't be loaded. This is achieved by using the *symbol_get* function to get the SGX driver's APIs during runtime. KVM-SGX needs to trust the SGX driver as it delegates SGX features and the EPC resource detection to the driver itself.

KVM-SGX introduces three data structures to store required information for SGX. First, a new data structure is added called *kvm_sgx* which stores per-VM SGX data. This data include the guest's CPUID and the EPC slot information. The EPC is a private memory slot in KVM-SGX. Second, the *kvm_epc* structure represents the EPC slot information for a particular VM. Third, the *kvm_epc_page* data structure tracks the EPC page status of the VM. This data structure stores all physical EPC pages that are allocated to the VM. When the VM is destroyed, KVM-SGX uses information of the data structure to free all pages that have been allocated to the VM.

KVM cooperates with QEMU to boot VMs. There are two parameters that KVM requires to expose SGX to VMs and in extension to unikernels. Both parameters are passed to the QEMU version that supports SGX. The first parameter is the *epc* which specifies the unikernel's EPC size using 1MB increments to the 128MB max size. The second parameter is *lehash* which creates a unikernel with 256-bit hash of a third party's RSA public key.

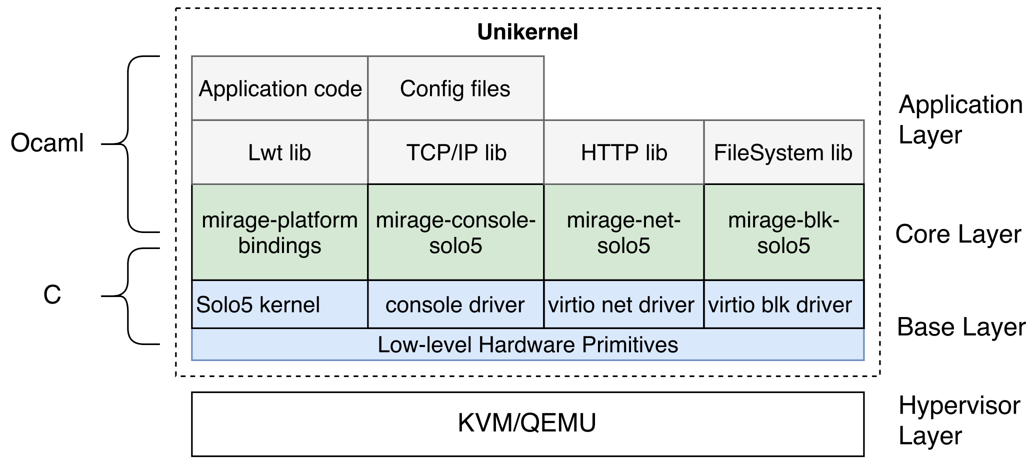


Figure 5.2: Unikernel support for the KVM hypervisor

5.3.4 KVM Support for MirageOS Unikernels

MirageOS unikernels provide support for the KVM hypervisor using a different unikernel base than mini-os. It uses the Solo5 unikernel base to integrate with the KVM hypervisor. Figure 5.2 illustrates the changes in each software layer needed for the unikernels to support the KVM hypervisor.

The development of the Solo5 unikernel base for KVM requires three elements. First, it requires to setup the kernel hardware initialisation that loads data into specified memory regions and starts the processor in the correct mode. Second, the integration with virtio devices which are supported in KVM hypervisor. Virtio provides a paravirtualized standard for the KVM hypervisor and other hypervisors. Third, the *mirage-platform* package includes the functionality for the bindings between the unikernel base and the higher level code used inside the unikernel. The Solo5 unikernel base first supports the KVM hypervisor and QEMU and then it includes support for a specialised unikernel monitor called *ukvm*. This monitor is a user space application that monitors the execution of unikernels and creates a minimal interface with only the required capabilities.

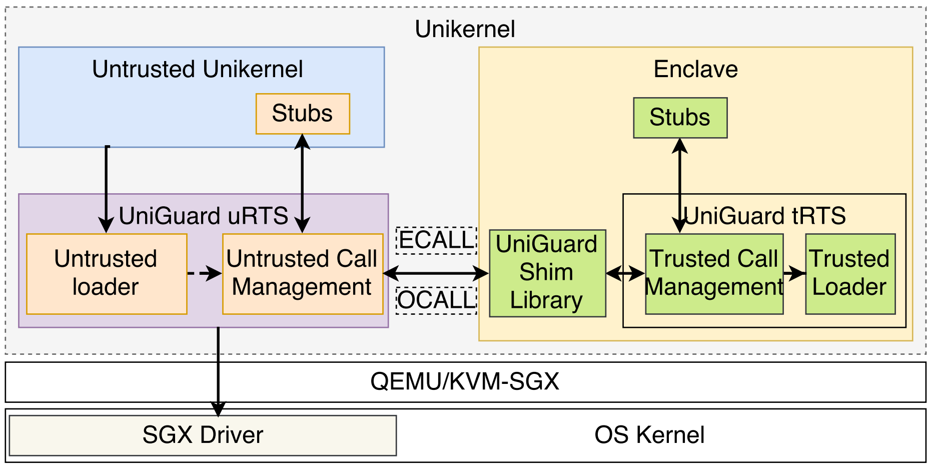


Figure 5.3: UniGuard components

5.3.5 UniGuard Components

Figure 5.3 depicts the components of the UniGuard architecture. All the software components inside the enclave are considered trusted. We have ported the trusted Run Time Service (tRTS) so that it can be used in UniGuard by using only the code necessary for the unikernels. For instance, we haven't ported the code for the Intel VTune library that provides profiling information and any code related to the simulation mode of the SGX. The tRTS contains the trusted loader and the Trusted Call Management inside the enclave. The Trusted Call Management provides an interface where ECALLs can be received and OCALLs send to the untrusted hypervisor to execute a hypercall. The stubs provide an interface inside the enclave for the application code executed inside the enclave.

On the untrusted part of the unikernel includes the untrusted stubs code which interface with the untrusted Call Management which delegates any OCALLS to the untrusted hypervisor and sends an ECALL to the trusted part of the unikernel. Even though, porting libraries from the SGX SDK and PSW to unikernels it still adds more functionality in the unikernel which could be avoided if these libraries where implemented in OCaml. This is the same approach that the Rust SGX SDK [37] project has taken. We consider this as potential future work for using unikernels and SGX.

5.4 UniGuard Implementation

The implementation of UniGuard involves the modification of MirageOS unikernels that enable the execution of a unikernel's trusted code and data inside a secure enclave. We have modified the MirageOS platform libraries that provide support for KVM and the Solo5 unikernel base that provides hardware virtualisation.

5.4.1 UniGuard Library

UniGuard creates an interface in unikernels where it can accept an enclave as an extra library and load the enclave after the execution of the unikernel. This is achieved by using all the required items needed by SGX to correctly load the enclave such as the enclave signature, the enclave library and the enclave token. During the creation of the enclave, if the enclave token is not present, then it is created by calling the *aesmd* service that is included with the Intel SGX SDK.

We have created a number of generic calls that follow the required partition for SGX. There are untrusted function calls for executing outside the enclave and trusted functions that can be executed inside the enclave. This generic API creates an enclave shim that it is also responsible to sanitise each calls parameters. Hence, we are creating a shield for the enclave untrusted calls. In addition, the virtualisation mechanism also ensures that the unikernel is isolated from other processes or unikernels in the host. The aim of UniGuard is not only to create enclaves but to also shield the calls to the OS and create defence in depth mechanisms.

The QEMU part of the KVM hypervisor includes as parameter the size of the EPC. The unikernel includes the path to the actually *enclave.so*, the enclave signature. During the execution of the unikernel, the unikernel loading code calls the SGX driver to initialise the enclave. The main elements for initialising the enclave is the actual enclave file, the number of threads, the enclave token, the enclave sigstruct, the enclave SECS, the enclave entry address, and the enclave size. Currently, SGX supports only enclaves which have the size of a power of two. After the initialisation step the create enclave call (ECREATE) is sent to the SGX driver with the required parameters.

5.4.2 UniGuard Shim Library

The UniGuard shim library sits in between the untrusted part of the unikernel and the enclave. Essentially, this library provides an interface that verifies and sanitises input or output information. The aim of this library is to maintain the confidentiality and integrity guarantees of the enclave calls. The shim library provides three different mechanisms that verify and sanitise parameters, check pointers, and verify integrity and o order of ecalls.

The first mechanism checks the parameters for each ecall according to certain criteria such as size of a string, or length of an array. If the criteria do not much the parameters then the parameters are sanitised or if they cannot be sanitised the ecall is rejected by the shim library. The second mechanism validates that the pointer should access only trusted memory inside the enclave, or only untrusted memory inside the unikernel. Therefore, mixing up pointers is not allowed by the shim library. If the request cannot be accepted by the shim library it is rejected. The third mechanism focuses on keeping the correct order of ecalls and providing a freshness counter to mitigate replay attacks.

5.4.3 Changes in ukvm

Apart from using the QEMU with support to SGX, we can use the ukvm monitor to provide a specialised hypervisor for unikernels. The ukvm monitor is modified to provide experimental support for SGX in a unikernel. The main modifications are aimed at providing access to an SGX driver API from inside the unikernel to the enclave. Hence, the unikernel part provides the untrusted part of an enclave application. Even though, the unikernel part is untrusted it is isolated using hardware virtualisation and protecting the unikernel from colluding processes if it was constructed as regular OS process. The unikernel includes the port of the SGX related functionality needed to interface with the host's kernel SGX driver. This is architected in this way because the enclave needs to be executed in ring 3. Hence, we also need to create a user mode inside the unikernel and then create the enclave inside the unikernel.

The enclave requires a subset of memory from the memory that the unikernel is allocated. If the enclave memory is more than the available unikernel memory then an exception is triggered and the enclave stops its execution and displays an error message to the console. Subsequently, the unikernel stops execution and the hypervisor destroys the unikernel. If the memory is enough then the enclave is started and the execution starts inside the enclave, performs the computations assigned to the enclave with confidentiality and integrity and returns control to the unikernel untrusted part to continue execution. The enclave cannot execute any OS calls from inside the enclave but can delegate the call to the untrusted part of the unikernel which in turn executes the call for the enclave. The unikernel then returns the results to the enclave after first verifying and sanitising the result. The main idea here is that the unikernel part of the application can perform its computations kernel mode and the return the result to the enclave.

One advantage of using the ukvm monitor is that we have a specialised monitor that supports SGX. Even though, the ukvm monitor is a user space process it could be secured by using an enclave to hold the main interactions with the unikernel. Using the approach a ukvm monitor is protected from any tampering attempts. We consider this approach as future work for UniGuard.

5.4.4 Changes in Solo5 libraries

We have introduced a number of changes for the Solo5 unikernel base and related libraries that provide support for the KVM hypervisor in MirageOS. Our first change in the *solo5* package that creates the *solo5-kernel-virtio* library is to introduce another module in the package that provides support for SGX.

5.5 Security Evaluation

We evaluate the security of UniGuard in two ways. First, a security analysis argues that the privileged software cannot compromise the confidentiality and integrity of a security-sensitive computation inside an enclave that uses the UniGuard security architecture. Second, we evaluate the UniGuard shim library empirically to demonstrate that the security requirements are fulfilled.

5.5.1 Security Analysis

The security analysis considers that UniGuard is a security architecture that provides trusted execution for unikernels deployed on the KVM hypervisor using Intel SGX. We establish three lemmas that demonstrate that UniGuard can shield sensitive computations from an untrusted cloud by using enclaves.

5.5.1.1 Assumptions

The attacks that are out of scope are discussed in subsection 5.2.2. The security analysis is built on the following explicit assumptions, which form the basis for the security argument.

Limited Access [access]: We assume that a malicious user cannot manipulate the CPU package and compromise it in any way. If this assumption does not hold then Intel SGX is not secure.

Trusted SDK and PSW [trustedsdk]: We consider the SGX SDK and PSW that is implemented without any vulnerabilities that would compromise the security of the enclave.

Intel SGX driver [sgxdriver]: The SGX driver is considered to not be trusted.

Trusted OCaml and GCC compiler [trustedcompiler]: We consider the OCaml compiler for building unikernels and the GCC compiler for building enclaves to be trustworthy and not compromised in any way.

Trusted microcode for SGX [trustedmicrocode]: The microcode used in the CPU to implement SGX is considered to be working correctly and is not tampered with.

5.5.1.2 Security Requirements

We establish the following security requirements that UniGuard needs to fulfil. First, we require that all parameters passed into and out of ecalls and ocalls have not been tampered with and are properly checked. Second, we require that a pointer may only point to untrusted memory and a pointer may only point to enclave memory. Third, we establish the integrity requirement for the ecalls that the enclave receives.

5.5.1.3 Security Proofs

Lemma 5.5.1 (**Ecall parameters passed in enclave**). *If the enclave has used the parameters passed in via an ecall, then the ecall parameters must have been checked by the UniGuard shim library.*

Proof. As an enclave function is executed without compromising the security guarantees of the enclave following that *[access]*, *[trustedsdk]*, *[trustedcompiler]*, and *[sgx-driver]* assumptions hold. The parameters must have been passed to the enclave function, only if they have been checked by the UniGuard shim library. The UniGuard shim library must have received the ecall parameters, only if there is a successful context switch to the enclave code by the *[trustedmicrocode]* assumption. The ecall must have been sent by a privileged process with a number of parameters as input. □

Lemma 5.5.2 (**Ocall parameters passed out of enclave**). *If the enclave has send the parameters passed in via an enclave function to an ocall, then the ocall parameters must have been sanitised by the UniGuard shim library.*

Proof. As an ocall is executed for requesting privileged or I/O operations, it must have successfully switched context from the trusted code in the enclave code to the untrusted part of the unikernel by the *[trustedmicrocode]* assumption. The ocall parameters must have been checked by the UniGuard shim library and the enclave function must have initiated the ocall without compromising the security inside the enclave by the *[access]*, *[trustedsdk]*, *[trustedcompiler]*, and *[sgxdriver]* assumptions. □

Lemma 5.5.3 (**Pointers may only point to enclave memory**). *If a pointer to an untrusted memory has been passed to the enclave via an ecall and the memory referenced is allocated inside the enclave, then the UniGuard shim library has previously checked the pointer.*

Proof. Case (in): As the memory referenced by the pointer is copied from the untrusted memory to the enclave memory. The enclave memory must have been correctly allocated by the trusted bridge following the *[trustedsdk]* assumption. The pointer to untrusted memory with attribute *in* must have been checked by the UniGuard shim library.

Case (out): As the contents of the trusted buffer are copied to untrusted memory. The contents must have been checked by the UniGuard shim library. The trusted function must have been passed a pointer to the trusted buffer and have cleared the contents of the buffer. The buffer in trusted memory must have been allocated correctly by the *[trustedsdk]* assumption. The pointer to untrusted memory with the *out* attribute must have been passed to the enclave and checked by the UniGuard shim library. □

Lemma 5.5.4 (**Ecalls integrity for enclave**). *If the enclave has received an ecall with correct order and freshness, it has been checked by the UniGuard shim library.*

Proof. For any ecall received by the enclave, it holds that the ecall is in order and is not repeated. We obtain the order of the ecall from the ecall number that is checked by the UniGuard shim library. The particular order of a set of ecalls required by the enclave function is defined in a data structure inside the enclave. The freshness property is obtained by a freshness counter stored in the enclave for each data item, which is validated by the UniGuard shim library. □

5.5.2 Security Testing

For each security requirement we devised a different test, which was executed in order to demonstrate empirically that the requirement is fulfilled when we use UniGuard. In our first test, we demonstrate that parameters passed into the enclave using an ecall are checked and then passed to the corresponding enclave function that requires them. We executed the test for checking the size of a parameter entering the enclave interfaces. We chose a variety of sizes to evaluate, if the UniGuard shim library would accept the request to the enclave. This security test was executed one hundred times. The results show that the UniGuard shim library accepts the requests that have parameters with acceptable size for the corresponding enclave interface.

For our second requirement, we devised a test to evaluate if the pointers only access trusted or untrusted memory but not both. The test included random pointers to both untrusted and trusted memory. The UniGuard shim library managed to accept a number of pointers correctly and deny the pointers that were trying to access both types of memory.

The third security test involved the testing of the correct order of ecalls. We then devised a test case that involved a number of correct order of ecalls and a number of

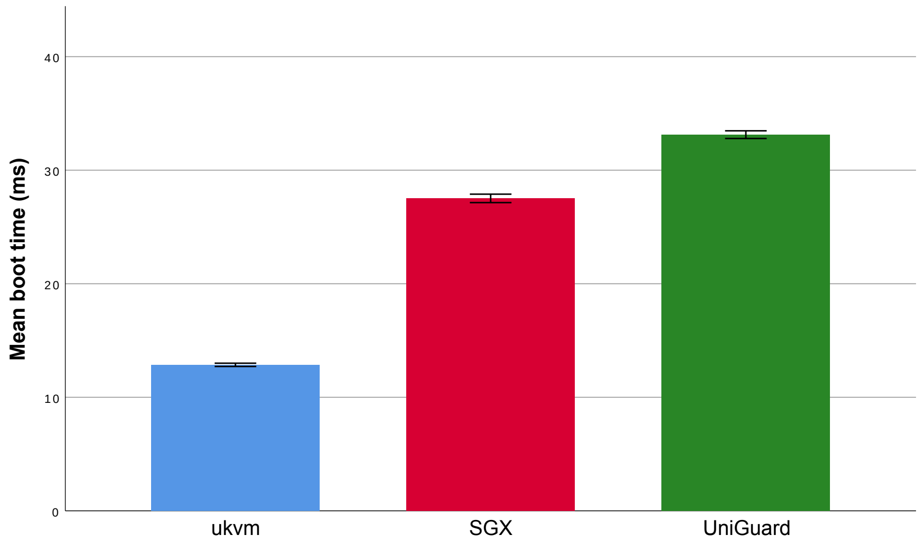


Figure 5.4: Boot times for a single unikernel, an enclave and UniGuard

ecalls with the wrong order. The UniGuard shim library managed to correctly validate the correct order of ecalls and deny the rest of ecalls, which are not in the correct order.

5.6 Performance Evaluation

In this section, we evaluate the performance of our UniGuard security architecture. Our goals for this subsection are (1) to evaluate UniGuard on a virtualised host using a set of performance metrics and investigate how unikernels perform when using UniGuard or not, (2) to identify performance overheads that might exist and (3) to discuss how performance could be improved in a future version of UniGuard.

We conducted our performance evaluation of UniGuard using an Intel NUC kit as a virtualisation server for our test experiments. Our testbed uses an Intel Core i5–6260U CPU with support for SGX version 1, 16 GB of RAM and a 500 GB SSD hard disk. The virtualisation server is running on top of Ubuntu 15.10 Linux OS with kernel version 4.11.0. Additionally, the virtualisation server uses a custom build of the KVM hypervisor, which supports SGX and MirageOS 3.0.5 for creating unikernels.

We evaluated the performance of unikernels and enclaves using three test cases: 1)

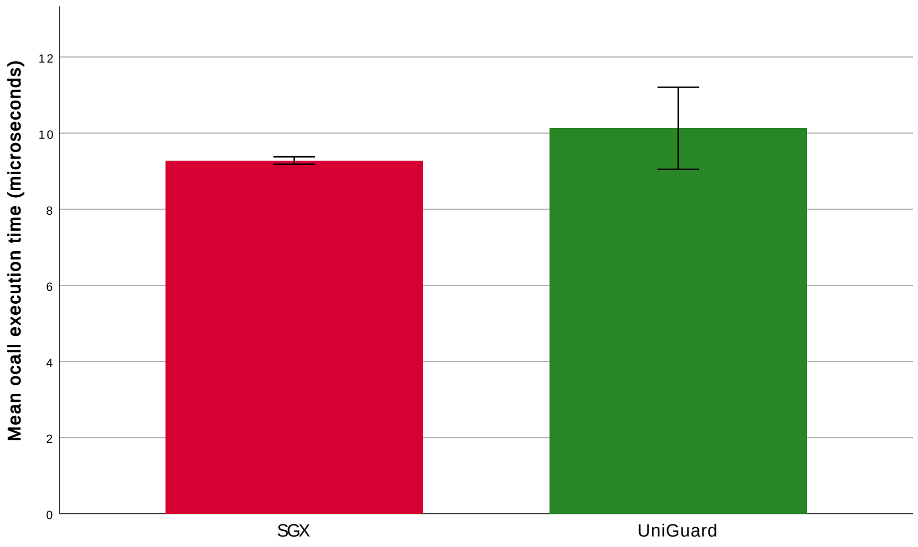


Figure 5.5: Mean execution time for ocalls

baseline (TC0), 2) SGX (TC1), 3) UniGuard (TC2). For the baseline configuration, we create a unikernel using *solo5* libraries and the *ukvm* monitor. The second test case creates an enclave using Intel SGX. The third test case creates a unikernel using UniGuard.

Figure 5.4 depicts the results from measuring the boot time of a single unikernel using ukvm, an enclave, and a unikernel using UniGuard. This performance test was executed one thousand times and afterwards the mean was calculated. Results show that booting a unikernel with *ukvm* provides the best overall performance with 11 ms boot time, 95% CI $[11.1, 11.5]$. The enclave is slightly slower, booting at 27.5 ms, at 95% CI $[27.1, 27.8]$. The *TC2* test case results show that the unikernel with the enclave boots at 33.1 ms, at 95% CI $[32.7, 33.4]$. Thus, UniGuard adds a moderate 20% overhead regarding boot time over the booting time of a single enclave.

Figure 5.5 depicts the results from measuring the execution time when an ocall is executed from an enclave and when an ocall is executed when the UniGuard shim library checks the contents of the ocalls. Results show that executing an ocall without the shim library provides the best performance, with mean execution time at 9.28 μs , 95% CI $[9.18, 9.37]$. The enclave using UniGuard is slightly slower, with mean execution time at 10.1 μs , at 95% CI $[9, 11.2]$. Thus, UniGuard adds a moderate 10%

overhead regarding average ocall execution time over the execution time of a single enclave. The reason for this overhead is the checking of the ocalls that the UniGuard shim library performs.

5.7 Conclusion

In this chapter we presented **UniGuard**, a security architecture that leverages Intel SGX to protect computations in unikernels from privileged software and physical attacks. The system partitions the unikernel in an untrusted and trusted part. The former part executes as an untrusted process inside a unikernel and the latter executes the sensitive code inside an enclave. We implemented the research prototype of **UniGuard** for the KVM hypervisor and our performance evaluation shows a moderate 20% overhead for executing the same computation inside an enclave in a Linux OS and an enclave inside a unikernel using UniGuard.

5.7.1 Limitations

A limitation of the UniGuard security architecture is that we only support a one-to-one relationship between the unikernel and the enclave. The reason for this approach is that a MirageOS unikernel currently supports only one thread for executing a computation in a unikernel. Therefore, the development of a research prototype is not as complex to build if we only support this relationship. Support for multiple enclaves would require more complex logic that manages the multiple enclaves and its resources and the context switching between the enclaves and the unikernel.

The focus of this research work was to investigate how the level of security of the unikernels can be increased rather that focusing on improving the performance of the unikernels. This shows during the performance evaluation. However, the performance of UniGuard can be improved if the context switching from trusted to untrusted code is minimised.

5.7.2 Future Work

Future work for the UniGuard system includes the ability to create multiple enclaves from a unikernel. Currently, UniGuard only supports a one to one relation between the unikernel the number of enclaves. Another area of future work is improving the performance of the system and decreasing the number of transitions from enclave code to unikernel code. This issue can be solved by the second version of SGX where we can dynamically adjust the size of the EPC resource.

Chapter 6

Conclusions

This research project was conducted in two main stages which involved investigating how to design, implement and evaluate security mechanisms for unikernels. These security mechanisms were designed following two perspectives. The first is that a malicious unikernel can exploit vulnerabilities in the virtualisation layer. Consequently, we need an efficient security mechanism that prevents privilege escalation and VM escapes. The second perspective is to protect the code and data of a unikernel during execution in an untrusted cloud.

During the first stage of this research project we designed, implemented and evaluated a novel security mechanism that provides access control to unikernels leveraging capabilities. In this stage, we added a new access control module in the hypervisor. We modified the unikernel base layer to provide access to capabilities and created a new library that the application layer in a unikernel can use. We obtain an approach that adheres to the POLP with a minimised trusted computing base.

The second stage of this research project was to create a security architecture to protect unikernels from software and certain hardware attacks. In this stage, we integrated unikernels with Intel SGX. We chose as a platform for the development of this security architecture a Type II hypervisor. We modified the way MirageOS works to create unikernels that can spawn secure enclaves and protect the unikernel's computations. In essence, a unikernel acts as a starting point for a secure enclave and provides the shielding functionality that regulates any calls to the OS and its replies and vice versa.

The remainder of this chapter has the following structure. Section 6.1 discusses our main contributions and how they differ from previous work. Finally, section 6.2 discusses threads of research related to this thesis that could be pursued in the future.

6.1 Summary of Contributions

This thesis includes three main contributions: a literature review, design and implementation of a novel access control mechanism for unikernels, and design and implementation of a security mechanism that protects unikernel from software and hardware attacks.

The literature review presented in this thesis provides a view on current virtualisation and hardware based solutions that provide access control to resources and protection from software or hardware attacks. It analyses if any of the solutions on one hand prevent unikernels from exhibiting malicious behaviour and on the other hand to protect the confidentiality and integrity of code and data of a unikernel during execution in an untrusted cloud.

The second contribution of this research project includes an access control mechanism for unikernels. We designed, developed and evaluated our security mechanism to regulate access to resources of a cloud infrastructure for unikernels. The novel contribution is that it is the only access control mechanism that focuses explicitly on unikernels and creates unikernels that follow the POLP using capabilities. This design prevents the privilege escalation and VM escape attacks that burden current cloud infrastructures.

Our final contribution protects unikernels from an untrusted cloud. For this contribution, we designed, implemented and evaluated a security architecture that protects unikernels from privileged software and hardware attacks leveraging Intel SGX. We propose how to integrate unikernels with Intel SGX using a Type II hypervisor. The novelty of this contribution is that it uses unikernels with a minimal attack surface to create secure enclaves. The secure enclaves execution guarantee the confidentiality and integrity of sensitive code and data.

6.2 Future Work

This section provides a discussion for a number of potential future works that stem from the research presented in this thesis.

Future work for the access control mechanism includes the ability to scale the architecture for thousands of unikernels and to improve the performance of VirtusCap by creating a fast cache for access requests. In addition, further minimising the attack surface of Dom0 and use bare metal rump kernels alongside unikernels and MirageOS. In this situation, we might have to adjust some aspects of our architecture in order to cater to the rump kernels device drivers and adjust how to secure them using our access control system.

Another research thread would be to port our security mechanism to Intel SGX 2 and take advantage of the dynamic allocation of memory which would ultimately improve performance. In addition, supporting the memory oversubscription model proposed by Intel how this can it be integrated with unikernels [26]. Automatic partitioning of unikernels to trusted and untrusted parts using annotations in the source code would be another promising thread of research. In this way, a unikernel developer can specify the security-sensitive computations and create an enclave protecting them.

For both our main contributions, future research could include further reduction in the TCB that deals with the management in the virtualisation layer. For instance, the KVM hypervisor requires a full blown OS, which has a large attack surface. The challenge here is to create an OS system or a minimal hypervisor that is specialised for unikernels. A similar approach is used with a specialised OS called CoreOS [32] for containers. Future research that would benefit both main contributions is the creation of a formal model that verifies rigorously the security properties of each contribution. For instance, a formal model of a capability-based access control mechanism could be created for unikernels and proved in a theorem prover such as Isabelle [97]. The same approach could be used for formally verifying the security properties of enclaves in unikernels.

Bibliography

[1] Amazon. Amazon Elastic Compute Cloud: User Guide. Technical report, Amazon, Feb. 2014.

[2] Amazon. *Amazon Web Services: Overview of Security Processes*, June 2014.

[3] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, 2013.

[4] J. Anderson. A Comparison of Unix Sandboxing Techniques. *FreeBSD Journal*, pages 16–25, 2017.

[5] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, USAF Electronic Systems Division, Oct. 1972.

[6] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. *IEEE Symposium on Security and Privacy*, pages 65–71, 1997.

[7] ARM. ARM Security Technology Building a Secure System using TrustZone Technology, Sept. 2009.

[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[9] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers,

R. Kapitza, P. R. Pietzuch, and C. Fetzer. SCONE - Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[10] P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch. TaLoS: Secure and transparent TLS termination inside SGX enclaves. *Imperial College London, Tech. Rep*, 5, 2017.

[11] A. M. Azab, P. Ning, and X. Zhang. SICE - a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2011.

[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, New York, USA, 2003. ACM Press.

[13] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the eleventh USENIX Symposium on Operating Systems Design and Implementation*. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 14, 2014.

[14] D. E. Bell and L. J. La Padula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report MITRE MTR-2997, The MITRE Corporation, Mar. 1976.

[15] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical report, DTIC Document, 1973.

[16] S. Berger, R. Cáceres, K. Goldman, D. Pendarakis, R. Perez, J. R. Rao, E. Rom, R. Sailer, W. Schildhauer, D. Srinivasan, S. Tal, and E. Valdez. Security for the Cloud Infrastructure: Trusted Virtual Data Center Implementation. *IBM Journal of Research and Development*, 53(4):6:1–6:12, 2009.

[17] S. Berger, R. Cáceres, D. Pendarakis, and R. Sailer. TVDc: Managing Security in the Trusted Virtual Datacenter. *ACM SIGOPS Operating Systems Review*, 2008.

[18] K. J. Biba, M. C. B. MA, and U. S. A. F. S. C. E. S. Division. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, DTIC Document, Apr. 1977.

[19] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, 2003.

[20] M. Blanc, A. Bousquet, J. Briffaut, L. Clevy, D. Gros, A. Lefray, J. Rouzaud-Cornabas, C. Toinard, and B. Venelle. Mandatory Access Protection Within Cloud Systems. In *Security, Privacy and Trust in Cloud Systems*, pages 145–173. Springer Berlin Heidelberg, Berlin, Heidelberg, Sept. 2013.

[21] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. *NIST SPECIAL PUBLICATION SP*, pages A–10, 1989.

[22] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In *CLOUDCOM '15: Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE Computer Society, Nov. 2015.

[23] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. *Proceedings. 1989 IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[24] E. Brickell, J. Camenisch, and L. Chen. Direct Anonymous Attestation. In *Proceedings of the 11th ACM conference on Computer and Communications Security*, pages 132–145. ACM, 2004.

[25] S. Brostoff and M. A. Sasse. Safe and Sound - a Safety-Critical Approach to Security. In *Proceedings of the 2001 Workshop on New Security Paradigms*, 2001.

[26] S. Chakrabarti, R. Leslie-Hurd, M. Vij, F. McKeen, C. Rozas, D. Caspi, I. Alexandrovich, and I. Anati. Intel® Software Guard Extensions (Intel® SGX) Architecture for Oversubscription of Secure Memory in a Virtualized Environment. In *the Hardware and Architectural Support for Security and Privacy*, pages 1–8, New York, New York, USA, 2017. ACM Press.

[27] S. Checkoway, H. Shacham, S. Checkoway, and S. Checkoway. Iago attacks: why the system call API is a bad untrusted RPC interface. In *ACM SIGARCH Computer Architecture News*, pages 253–264. ACM, Apr. 2013.

[28] W. R. Claycomb and A. Nicoll. Insider Threats to Cloud Computing: Directions for New Research Challenges. In *2012 IEEE 36th Annual Computer Software and Applications Conference - COMPSAC 2012*, pages 387–394. IEEE, Sept. 2017.

[29] Cloud Security Alliance. The Treacherous 12 - Top Threats to Cloud Computing + Industry Insights, 2017.

[30] Cloudozer. Erlang on Xen: at the heart of super-elastic clouds. https://github.com/cloudozer/ling, 2014. [Accessed April 4, 2017].

[31] G. Coker. Xen Security Modules (XSM). In *Xen Summit*, pages 1–14, Apr. 2007.

[32] CoreOS. CoreOS. https://coreos.com. [Accessed October 22, 2018].

[33] V. Costan and S. Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016.

[34] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum - Minimal Hardware Extensions for Strong Software Isolation. *USENIX Security Symposium*, 2016.

[35] Cybersecurity Insiders. Cloud Security Spotlight. https://www.alertlogic.com/assets/industry-reports/Cloud-Security-Spotlight-2018.pdf, 2018. [Accessed April 22, 2019].

[36] J. B. Dennis and E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 1966.

[37] R. Duan, L. Li, S. Jia, Y. Ding, L. Wei, and T. Chen. Rust SGX SDK. https://github.com/baidu/rust-sgx-sdk, 2017. [Accessed September 9, 2017].

[38] A. J. Duncan, S. Creese, and M. Goldsmith. Insider Attacks in Cloud Computing. *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, 1:857–862, 2012.

[39] G. Dunlap. The Intel SYSRET privilege escalation, June 2012.

[40] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified Protection Model of the seL4 Microkernel. *VSTTE*, 5295(Chapter 11):99–114, 2008.

[41] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, New York, 1995. ACM Press.

[42] D. F. Ferraiolo, J. Cugini, and D. R. Kuhn. Role-Based Access Control (RBAC): Features and Motivations. In *Proceedings of 11th Annual Computer Security Application Conference (ACSAC)*, 1995.

[43] Galois. Haskell Lightweight Virtual Machine. https://github.com/GaloisInc/HaLVM. [Accessed October 22, 2018].

[44] J. Geffner. VENOM - Virtualized Environment Neglected Operations Manipulation. http://venom.crowdstrike.com, Apr. 2015. [Accessed April 4, 2017].

[45] GlobalPlatform, Inc. TEE Client API Specification. Technical report, GlobalPlatform Inc., Mar. 2013.

[46] GlobalPlatform, Inc. TEE Internal Core API Specification v1.1.1. Technical report, GlobalPlatform Inc., June 2016.

[47] GlobalPlatform, Inc. TEE Management Framework v1.0. Technical report, GlobalPlatform Inc., Nov. 2016.

[48] GlobalPlatform, Inc. TEE System Architecture v1.1. Technical report, GlobalPlatform Inc., Jan. 2017.

[49] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[50] Google. App Engine - Build Scalable Web & Mobile Backends in Any Language — Google Cloud Platform. https://cloud.google.com/appengine/, 2017. [Accessed October 22, 2018].

[51] T. Groß and I. Sfyrakis. Specification of the Graph Signature Cryptographic Library and the PRISMACLOUD Topology Certification Version 0.9.2. Technical Report CS-TR-1523, Newcastle University, July 2018.

[52] S. Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive*, 2016.

[53] S. Gueron. Memory Encryption for General-Purpose Processors. *IEEE Security and Privacy*, 2016.

[54] G. Hofemeier. *Intel Digital Random Number Generator (DRNG) software implementation guide*. Intel, May 2014.

[55] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vTZ: Virtualizing ARM TrustZone. In *USENIX Security 2017*, 2017.

[56] M. S. Inci, B. Gülmezoglu, G. I. Apecechea, T. Eisenbarth, and B. Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. *IACR Cryptology ePrint Archive*, 2015.

[57] Intel Corporation. Intel Software Guard Extensions Programming Reference. Technical Report 329298-002US, Intel, Oct. 2014.

[58] Intel Corporation. *Intel Trusted Execution Technology (Intel TXT)*, July 2015.

[59] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. Kang. PrivateZone: Providing a Private Execution Environment using ARM TrustZone. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2016.

[60] A. Kantee. *Flexible Operating System Internal: The Design and Implementation of the Anykernel and Rump Kernels* . PhD thesis, Aalto University, 2012.

[61] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. OSv - Optimizing the Operating System for Virtual Machines. *USENIX Annual Technical Conference*, pages 61–72, 2014.

[62] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2–70, 2014.

[63] 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 *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. NICTA, ACM, Oct. 2009.

[64] R. Ko and S. S. G. Lee. Cloud Computing Vulnerability Incidents: A Statistical Overview. Technical report, Cloud Security Alliance, Nov. 2013.

[65] S. Kuenzer, F. Huici, and F. Schmidt. Unicore. https://lists.xenproject.org/archives/html/mirageos-devel/2017-09/pdfu1nBH1Z1NF.pdf, Sept. 2017. [Accessed April 4, 2017].

[66] KVM. Kernel Virtual Machine. https://www.linux-kvm.org/, 2017. [Accessed June 8, 2017].

[67] S.-M. Lee, S.-B. Suh, B. Jeong, and S. Mo. A Multi-Layer Mandatory Access Control Mechanism for Mobile Devices Based on Virtualization. In *2008 5th IEEE Consumer Communications and Networking Conference*, pages 251–256. IEEE, 2008.

[68] T. Leonard and M. Preston. MirageOS Security Advisory 02 - grant unshare vulnerability in mirage-xen. https://mirage.io/blog/MSA02, Apr. 2019. [Accessed April 27, 2019].

[69] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[70] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. J. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie. Jitsu: Just-In-Time Summoning of Unikernels. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 559–573, 2015.

[71] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels. In *the eighteenth international conference*, pages 461–472, New York, New York, USA, 2013. ACM Press.

[72] A. Madhavapeddy and D. J. Scott. Unikernels: the Rise of the Virtual Library Operating System. *Communications of the ACM*, 57(1):61–69, 2014.

[73] P. K. Manadhata. *An Attack Surface Metric*. PhD thesis, Carnegie Mellon University, Pittsburgh, Dec. 2008.

[74] F. Manco, J. Martins, K. Yasukata, J. Mendes, S. Kuenzer, and F. Huici. The Case for the Superfluid Cloud. In *HotCloud'15: Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing*. USENIX Association, 2015.

[75] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 459–473. USENIX Association, 2014.

[76] J. M. McCune, T. Jaeger, S. Berger, R. Cáceres, and R. Sailer. Shamon: A System for Distributed Mandatory Access Control. In *22nd Annual Computer Security Applications Conference (ACSAC)*, pages 23–32. IEEE, 2006.

[77] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. *EuroSys*, 42(4):315–328, 2008.

[78] H. Mehnert. MirageOS Security Advisory 00 - memory disclosure in mirage-net-xen. https://mirage.io/blog/MSA00, May 2016. [Accessed April 23, 2019].

[79] P. Mell and T. Grance. The NIST definition of cloud computing. *National Institute of Standards and Technology*, 2009.

[80] S. Michaels and J. Dileo. Assessing Unikernel Security. Technical report, NCC group, Apr 2019.

[81] Microsoft. Next-Generation Secure Computing Base (NGSCB). https://www.microsoft.com/resources/ngscb/default.mspx, 2003. [Accessed April 4, 2017].

[82] MITRE. The Intel SYSRET privilege escalation - CVE-2012-0217, June 2012.

[83] MITRE. libvchan failure handling malicious ring indexes - CVE-2014-1896, Feb. 2014.

[84] MITRE. Remote exploit vulnerability in bash - CVE-2014-6271, Oct. 2014.

[85] MITRE. FreeBSD: qemu – Heap overflow in QEMU PCNET controller, allowing guest to host escape - CVE-2015-3209, June 2015.

[86] MITRE. VENOM: QEMU vulnerability - CVE-2015-3456, May 2015.

[87] C. Molina-Jiménez, I. Sfyrakis, E. Solaiman, I. Ng, M. W. Wong, A. Chun, and J. Crowcroft. Implementation of smart contracts using hybrid architectures with on- and off-blockchain components. *arXiv:1808.00093 [cs.SE]*, 2018.

[88] C. Molina-Jiménez, I. Sfyrakis, E. Solaiman, I. Ng, M. W. Wong, A. Chun, and J. Crowcroft. Implementation of smart contracts using hybrid architectures with on- and off-blockchain components. *8th IEEE International Symposium on Cloud and Services Computing (IEEE SC2)*, 2018.

[89] C. Molina-Jiménez, E. Solaiman, I. Sfyrakis, I. Ng, and J. Crowcroft. On and Off-Blockchain Enforcement Of Smart Contracts. In *International Workshop on Future Perspective of Decentralized Applications (FPDAPP 2018)*, 2018.

[90] C. Molina-Jiménez, E. Solaiman, I. Sfyrakis, I. Ng, and J. Crowcroft. On and off-blockchain enforcement of smart contracts. *arXiv:1805.00626 [cs.CY]*, 2018.

[91] J. Morris. sVirt: Hardening Linux Virtualization with Mandatory Access Control. In *Linux.conf.au Conference*, 2009.

[92] D. Mosberger and T. Jin. httperf - A Tool for Measuring Web Server Performance. Technical Report HPL-98-61, Hewlett Packard, Apr. 1998.

[93] T. Moses. *Extensible Access Control Markup Language (XACML) version 2.0*. Oasis Standard, 2005.

[94] Y. Naik. Xen-Cap: A Capability Framework for Xen. Technical report, University of Utah, Salt Lake City, 2013.

[95] National Institute of Standards and Technology. An Introduction to Computer Security: The NIST Handbook. Technical report, NIST, Washington, 1995.

[96] M.-D. Nguyen, N.-T. Chau, S. Jung, and S. Jung. A Demonstration of Malicious Insider Attacks inside Cloud IaaS Vendor. *International Journal of Information and Education Technology*, 4:483–486, 2014.

[97] L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[98] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 291, New York, Mar. 2011. ACM Press.

[99] M. Preston. MirageOS Security Advisory 01 - memory disclosure in mirage-net-xen. https://mirage.io/blog/MSA01, Mar. 2019. [Accessed April 23, 2019].

[100] F. Rocha, T. Gross, and A. van Moorsel. Defense-in-Depth Against Malicious Insiders in the Cloud. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 88–97. IEEE, 2013.

[101] A.-R. Sadeghi. Trusted Execution Environments Intel SGX. https://bit.ly/2GEM9f9, Jun 2014. [Accessed April 24, 2019].

[102] R. Sailer, T. Jaeger, E. Valdez, R. Cáceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 276–285. IEEE, 2005.

[103] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. *Techn. Rep. RC23511*, 2005.

[104] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[105] H. Scherzer, R. Canetti, P. A. Karger, H. Krawczyk, T. Rabin, and D. C. Toll. Authenticating Mandatory Access Controls and Preserving Privacy for a High-Assurance Smart Card. In *Computer Security – ESORICS 2003*, pages 181–200. Springer Berlin Heidelberg, Berlin, Heidelberg, Oct. 2003.

[106] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.

[107] I. Sfyrakis and T. Groß. UniGuard: Protecting Unikernels using Intel SGX. Technical Report CS-TR, Newcastle University, September 2017.

[108] I. Sfyrakis and T. Groß. VirtusCap: Capability-based Access Control for Unikernels. *IEEE Cloud Engineering Conference (2017)*, 2017.

[109] I. Sfyrakis and T. Groß. UniGuard: Protecting Unikernels using Intel SGX. *IEEE Cloud Engineering Conference (2018)*, 2018.

[110] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: A fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 170–185, New York, NY, USA, 1999. ACM.

[111] W. Shi, C. Lu, and H.-H. S. Lee. Memory-Centric Security Architecture. *HiPEAC*, 3793(Chapter 11):153–168, 2005.

[112] W. Shi, C. Lu, and H.-H. S. Lee. Memory-centric security architecture. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 153–168. Springer, 2005.

[113] S. Shinde, D. Le Tien, S. Tople, and P. Saxen. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2017)*, Reston, VA, 2017. Internet Society.

[114] A. Shriraman, S. Dwarkadas, A. Shriraman, and S. Dwarkadas. Sentry: lightweight auxiliary memory access control. In *ACM SIGARCH Computer Architecture News*, pages 407–418. ACM, June 2010.

[115] E. Solaiman, I. Sfyrakis, and C. Molina-Jiménez. Dynamic Testing and Deployment of a Contract Monitoring Service. In *5th International Conference on Cloud Computing and Services Science (CLOSER 2015)*, pages 463–474. SCITEPRESS, 2015.

[116] E. Solaiman, I. Sfyrakis, and C. Molina-Jiménez. Dynamic Testing and Deployment of a Contract Monitoring Service. Technical Report CS-TR-1460, Newcastle University, January 2015.

[117] E. Solaiman, I. Sfyrakis, and C. Molina-Jiménez. A State Aware Model and Architecture for the Monitoring and Enforcement of Electronic Contracts. *18th IEEE Conference on Business Informatics*, 2016.

[118] E. Solaiman, I. Sfyrakis, and C. Molina-Jiménez. High Level Model Checker Based Testing Of Electronic Contracts. In *Cloud Computing and Services Science*, volume 581, pages 193–215. Springer International Publishing, 2016.

[119] E. Solaiman, I. Sfyrakis, and C. Molina-Jiménez. High Level Model Checker Based Testing Of Electronic Contracts. Technical Report CS-TR-1490, Newcastle University, January 2016.

[120] E. R. Sparks. A Security Assessment of Trusted Platform Modules Computer Science. Technical Report TR2007-597, Dartmouth College, 2007.

[121] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies . In *SSYM'99: Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*. Secure Computing Corporation, USENIX Association, Aug. 1999.

[122] U. Steinberg and B. Kauer. NOVA: a Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems*, page 209, New York, Apr. 2010. ACM Request Permissions.

[123] K. Stengel, F. Schmaus, and R. Kapitza. EsseOS: Haskell-based tailored services for the cloud. *ARM@Middleware :1-4:6*, pages 4–6, 2013.

[124] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. *AEGIS: architecture for tamper-evident and tamper-resistant processing*. architecture for tamper-evident and tamper-resistant processing. ACM, New York, New York, USA, June 2003.

[125] J. Szefer and R. B. Lee. Hardware-Enhanced Security for Cloud Computing. In *Secure Cloud Computing*, pages 57–76. Springer New York, New York, NY, Dec. 2013.

[126] TCG. Trusted Platform Module Library Part 1: Architecture. Technical report, The Trusted Computing Group, Nov. 2014.

[127] H. Tian, Y. Zhang, C. Xing, and S. Yan. SGXKernel: A Library Operating System Optimized for Intel SGX. In *Proceedings of the Computing Frontiers Conference*, pages 35–44, New York, NY, USA, 2017. ACM.

[128] Trusted Computing Group. TPM Main Specification Level 2, part 1-3, Version 1.2, Revision 116. http://www.trustedcomputinggroup.org/tpm-main-specification/. [Accessed June 4, 2017].

[129] Trusted Computing Group. Trusted Platform Module Library Specification, part 1-4, Family "2.0", Level 00, Revision 01.38. http://www.trustedcomputinggroup.org/tpm-library-specification. [Accessed June 4, 2017].

[130] C. C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: a Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2017.

[131] S. Türpe, A. Poller, J. Steffan, J.-P. Stotz, and J. Trukenmüller. Attacking the BitLocker Boot Process. *TRUST*, 2009.

[132] P. Varanasi and G. Heiser. Hardware-supported virtualization on ARM. *APSys*, page 11, 2011.

[133] A. Vasudevan, J. M. McCune, N. Qu, L. van Doorn, and A. Perrig. Requirements for an Integrity-Protected Hypervisor on the x86 Hardware Virtualized Architecture. *TRUST*, 6101(Chapter 10):141–165, 2010.

[134] VMWare. Understanding Full Virtualization, Paravirtualization, and Hardware Assist. Technical report, VMWare, Oct. 2007.

[135] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. *IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[136] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Oct. 2016.

[137] M. V. Wilkes and R. M. Needham. *The Cambridge CAP computer and its operating system*. North-Holland, 1979.

[138] D. Williams and R. Koller. Unikernel Monitors - Extending Minimalism Outside of the Box. *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[139] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. *Architectural Support for Programming Languages and Operating Systems, ASPLOS*, page 304, 2002.

[140] Wojtczuk, Rafal and De Graaf, Daniel. Vchan Xen Library. https://github.com/mirage/xen/tree/master/tools/libvchan, 2010. [Accessed April 23, 2019].

[141] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI Capability

Model: Revisiting RISC in an Age of Risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, Apr. 2014.

[142] World Economic Forum. The Future of Jobs across Industries. http://reports.weforum.org/future-of-jobs-2018/the-future-of-jobs-across-industries/, 2018. [Accessed April 23, 2019].

[143] Xen. Xen Security Modules : XSM-FLASK. http://wiki.xen.org/wiki/Xen_Security_Modules_:_XSM-FLASK. [Accessed April 23, 2019].

[144] Xen. Xen Project Software Overview. http://wiki.xen.org/wiki/Xen_Project_Software_Overview, Apr. 2015. [Accessed April 23, 2019].

[145] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks - Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[146] L. T. Yang, W. Qiang, H. Jin, S. Wang, D. Zou, and L. Shi. CloudAC: a cloud-oriented multilayer access control system for logic virtual domain. *IET Information Security*, 7(1):51–59, Mar. 2013.

[147] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 203–216, 2011.

[148] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, pages 305–316, New York, 2012. ACM Press.

Appendix A

Partial code listings for VirtusCap

The full source code for VirtusCap can be found in `https://gitlab.com/gsfyrakis/virtuscap_project`.

```
/xen/include/xsm/xsm.h
...
struct xsm_operations {
void (*security_domaininfo) (struct domain *d,
                                    struct xen_domctl_getdomaininfo *info);
    int (*domain_create) (struct domain *d, u32 ssidref);
    int (*getdomaininfo) (struct domain *d);
    int (*domctl_scheduler_op) (struct domain *d, int op);
    int (*sysctl_scheduler_op) (int op);
    int (*set_target) (struct domain *d, struct domain *e);
    int (*domctl) (struct domain *d, int cmd);
    int (*sysctl) (int cmd);
    int (*evtchn_unbound) (struct domain *d, struct evtchn *chn, domid_t id2);
    int (*evtchn_interdomain) (struct domain *d1, struct evtchn *chn1,
                                    struct domain *d2, struct evtchn *chn2);
    void (*evtchn_close_post) (struct evtchn *chn);
    int (*evtchn_send) (struct domain *d, struct evtchn *chn);
    int (*evtchn_status) (struct domain *d, struct evtchn *chn);
    int (*evtchn_reset) (struct domain *d1, struct domain *d2);

    int (*grant_mapref) (struct domain *d1, struct domain *d2, uint32_t flags);
    int (*grant_unmapref) (struct domain *d1, struct domain *d2);
    int (*grant_setup) (struct domain *d1, struct domain *d2);
    int (*grant_transfer) (struct domain *d1, struct domain *d2);
    int (*grant_copy) (struct domain *d1, struct domain *d2);
    int (*grant_query_size) (struct domain *d1, struct domain *d2);
...
}

/xen/xsm/virtuscap/hooks.c
```

```
...
static struct xsm_operations virtuscap_ops = {
       .security_domaininfo = virtuscap_security_domaininfo,
       .domain_create = virtuscap_domain_create,
       .getdomaininfo = virtuscap_getdomaininfo,
       .domctl_scheduler_op = virtuscap_domctl_scheduler_op,
       .sysctl_scheduler_op = virtuscap_sysctl_scheduler_op,
       .set_target = virtuscap_set_target,
       .domctl = virtuscap_domctl,
       .sysctl = virtuscap_sysctl,
       .readconsole = virtuscap_readconsole,

       .evtchn_unbound = virtuscap_evtchn_unbound,
       .evtchn_interdomain = virtuscap_evtchn_interdomain,
       .evtchn_close_post = virtuscap_evtchn_close_post,
       .evtchn_send = virtuscap_evtchn_send,
       .evtchn_status = virtuscap_evtchn_status,
       .evtchn_reset = virtuscap_evtchn_reset,

       .grant_mapref = virtuscap_grant_mapref,
       .grant_unmapref = virtuscap_grant_unmapref,
       .grant_setup = virtuscap_grant_setup,
       .grant_transfer = virtuscap_grant_transfer,
       .grant_copy = virtuscap_grant_copy,
       .grant_query_size = virtuscap_grant_query_size,
...
}
```

Listing A.1: Partial list of supported XSM and VirtusCap operations

A.1 VirtusCap code additions to xen.h

```
/xen/include/public/xen.h

...
#define __HYPERVISOR_tmem_op 38
#define __HYPERVISOR_xc_reserved_op 39
#define __HYPERVISOR_xenpmu_op 40

#define __HYPERVISOR_virtuscap_create_op 41
#define __HYPERVISOR_virtuscap_grant_op 42
#define __HYPERVISOR_virtuscap_query_op 43
#define __HYPERVISOR_virtuscap_revoke_op 44
...
```

Listing A.2: VirtusCap code additions to xen.h file

A.2 VirtusCap hypercalls additions in Xen

```
/xen/arch/x86/x86_64/entry.S
...
ENTRY(hypercall_table)
...
    .quad do_virtuscap_create_op /* virtuscap hypercall 41 */
    .quad do_virtuscap_grant_op /* virtuscap hypercall 42 */
    .quad do_virtuscap_query_op /* virtuscap hypercall 43 */
    .quad do_virtuscap_revoke_op /* virtuscap hypercall 44 */
...
ENTRY(hypercall_args_table)
...
    .byte 1 /* do_virtuscap_create_op */ /* 41 */
    .byte 1 /* do_virtuscap_grant_op */ /* 42 */
    .byte 1 /* do_virtuscap_query_op */ /* 43 */
    .byte 1 /* do_virtuscap_revoke_op */ /* 44 */
...
```

Listing A.3: Add VirtusCap hypercalls in Xen hypercall table

A.3 Add prototypes for VirtusCap hypercalls

```
/xen/include/xen/hypercall.h
...
extern int do_virtuscap_create_op(struct vcap, struct domain *d);
extern int do_virtuscap_grant_op(struct vcap, struct domain *d);
extern int do_virtuscap_query_op(int cap, struct domain *d);
extern int do_virtuscap_revoke_op(int cap, struct domain *d);
...
```

Listing A.4: Add prototypes for VirtusCap hypercalls

A.4 Function for VirtusCap hypercalls

```
/tools/libxc/xc_private.h
...
static inline int do_virtuscap_query_op(xc_interface *xch, int cap, domid_t
    domain_id)
{
    int ret;
    DECLARE_HYPERCALL;
    DPRINTF("**do_virtuscap_query_op_hypercall_buffer* START\n");
    hypercall.op = __HYPERVISOR_virtuscap_query_op;
    hypercall.arg[0] = cap;
    hypercall.arg[1] = domain_id;
    ret = do_xen_hypercall(xch, &hypercall);
    DPRINTF("return do_virtudcap_query_op xen hypercall: %d\n", ret);
    DPRINTF("**do_virtuscap_query_op_buffer STOP\n");
    return ret;
}
...
/stubdom/libxcx86_32/xc_private.c
...
int xc_virtuscap_query_op(xc_interface *xch, int cap, domid_t domain_id)
{
    return do_virtuscap_query_op(xch, cap, domain_id);
}
...
```

Listing A.5: Add function for VirtusCap hypercalls

A.5 VirtusCap query hypercall

```
/xen/common/domctl.c
...

/* virtuscap hypercall */
int do_virtuscap_query_op(int cap, struct domain *d) {
    int rc = 0;
    d = rcu_lock_domain_by_any_id(domain_id);

    if (d == NULL) {
        printk("d is null");
        return -ESRCH;
    }

    rc = xsm_virtuscap_query_op(XSM_HOOK, cap, d);
    printk("result from xsm_virtcap_op: %d\n", rc);
    rcu_unlock_domain(d);

    if (rc < 0) {
       return -EPERM;
     } else {
      return rc;
    }
}
...
```

Listing A.6: Implementation of VirtusCap query hypercall

A.6 Implementation of VirtusCap security hooks

```
/xen/xsm/virtuscap/hooks.c
...

static int virtuscap_query_op(int cap, struct domain *d) {
     int i=0;
    struct vc_list *vl_t;
    struct vcap vcap_t;
    struct virtuscap_rights *vrights;
    vl_t = xzalloc(struct vc_list);
    vl_t->d = d;
    int len = vl_t->lenth;
    for (i=0; i<len; i++) {
        vcap_t=vl_t->c_list[i];
        if (vcap_t.cap_id == (unsigned long) cap) {
            return 0;
        }
    }

    return -1;
}
...
```

Listing A.7: Implementation of VirtusCap security hooks

A.7 VirtusCap OCaml interface

```
virtuscap.mli
...
external virtuscap_create: unit -> unit = "stub_virtuscap_create"
external virtuscap_grant: unit -> unit = "stub_virtuscap_grant"
external virtuscap_query: unit -> unit = "stub_virtuscap_query"
external virtuscap_revoke: unit -> unit = "stub_virtuscap_revoke"

...
```

Listing A.8: VirtusCap OCaml interface in mirage-platform library

```
virtuscap_stub.c
...
CAMLprim value stub_virtuscap_query(value unit)
{
    CAMLparam1(unit);
    CAMLlocal1(resource);
    printk("virtuscap query \n");
    virtuscap_query(resource);
    resource = Val_unit;
    CAMLreturn(resource);
}
    ...
```

Listing A.9: VirtusCap stub interface in mirage-platform library

Acronyms

AESMD Architectural Enclave Service Manager Daemon 120

ASR Address Space Randomization 22

AVC Access Vector Cache 37

CPU Central Processing Unit 119

CSP Cloud Service Provider v, 1, 85, 117

DHCP Dynamic Host Configuration Protocol 109

ecall Enclave call 129, 130

EPC Enclave Page Cache 50, 135

HTML Hypertext Markup Language 22, 110

HTTP Hypertext Transfer Protocol 22, 91, 110

IaaS Infrastructure as a Service 27

IP Internet Protocol 109

KVM Kernel-based Virtual Machine 41, 96, 116, 119, 120, 129, 132, 134

MEE Memory Encryption Engine 50, 53

MLS Multi-Level Security 34, 101

ocall Outside call 129, 130

OS Operating System v, 2–4, 15–20, 22, 27–32, 43, 47, 49, 59, 61–63, 67, 72, 78, 86, 87, 91, 92, 97, 116, 117, 119, 134

PaaS Platform as a Service 26

PDP Policy Decision Point 88, 105, 106

PEP Policy Enforcement Point 89, 105, 106

POLP Principle of Least Privilege 4, 7, 13, 67, 90, 116, 136, 137

PRM Processor Reserved Memory 50

QEMU Quick Emulator 42

RBAC Role-based Access Control 34, 101

RTM Root of Trust for Measurement 45

RTR Root of Trust for Reporting 45

RTS Root of Trust for Storage 45

SaaS Software as a Service 26

TCB Trusted Computing Base 14, 86

TCP/IP Transmission Control Protocol/Internet Protocol 91

TE Type Enforcement 34, 101

TEE Trusted Execution Environment 4, 11, 43, 83, 118

TPM Trusted Platform Module 11, 44, 104

VENOM Virtualized Environment Neglected Operations 1, 86

VM Virtual Machine 1, 61, 63, 65, 66, 86, 92

VMM Virtual Machine Monitor 28, 31

XSM Xen Security Module 6, 95, 96, 115

Glossary

Access Vector Cache A cache used for storing access control decisions when Xen hypervisor is configured to use the Flask security module 37

Address Space Randomization A computer security technique that prevents the exploitation of memory corruption vulnerabilities by randomly arranging the address space positions of the key data areas of a big process 22

ARM TrustZone A hardware security extension, which provides a secure execution environment by dividing computer resources between two compartments, the normal world where regular software is executed and the secure world where the security sensitive software is executed in an isolated environment 58

Capability An unforgeable token that designates access to a resource, according to a set of rights xiii

Capability list A data structure that includes the capabilities of a particular subject, which designate the access granted for objects 15

Cloud Operating System A library operating system that is optimised for cloud computing and packages a single application in a virtual machine by removing extraneous software components and executing in the same address space 19

Cloud Service Provider A third-party corporation that offers computing, networking and storage resources to multiple customers using a pay-per-use model v, 88

Enclave A protected execution environment embedded in a process that is used in Intel SGX and protects the confidentiality and integrity of code and data that reside in an enclave 50

Flux Advanced Security Kernel A security architecture for regulating access of resources with flexible security policies by controlling the propagation and the enforcement of access rights, and supporting the revocation of previously granted access rights 96

Hypervisor v, *see* Virtual Machine Monitor

Infrastructure as a Service A low-level service layer in cloud computing that provides the tenant with the list of computing resources that are available and can select the appropriate processing, storage, and network resources according to her requirements, which are under her control. The cloud provider manages the physical and virtualised infrastructure 27

Intel Software Guard Extensions A set of CPU instruction codes that provide integrity and confidentiality guarantees to security- sensitive computations performed on a computer where all the privileged software is potentially compromised 4

Kernel-based Virtual Machine A Type II hypervisor that provides full virtualisation for Linux on x86 hardware that supports virtualisation extensions and consists of a linux kernel module and a processor specific module 41

Library Operating System A library or set of libraries that provide as much of the operating system as it is possible at the user level with the goal to minimise the overhead from context switching and improve the flexibility of processes 18

Libvchan A library in Xen hypervisor that creates a bidirectional communication channel between two VMs based on the grant table mechanism and event channels 40

Mandatory Access Control A type of access control where only the system administrator can specify usage and access policies to resources, which the users cannot alter 34

Memory Encryption Engine A hardware component of the Intel SGX architecture that provides cryptographic protection of memory 50

Multi-Level Security A security policy that enables you to classify objects and users based on a system of hierarchical security levels and a system of non-hierarchical security categories 34

Platform as a Service A medium-level service layer in cloud computing that gives tenants the ability to deploy applications they have developed to the cloud infrastructure using software and tools the cloud provider offers 26

Policy Decision Point A component in an access control system that decides if it will authorize or not the request of a user or process, taking into account the available information and relevant security policies 88

Principle of Least Privilege A design principle for assigning an application only the resources and information that are necessary to fulfil its purpose 4, 86

Privilege escalation The exploitation of a bug, design flaw or misconfiguration in operating systems or software components, which aims to elevate access to resources that are normally protected from an application or user. The application or user gains privileges that were not intended by the application developer or system administrator and performs unauthorised actions 1

Quick Emulator A hosted virtual machine monitor that emulates the processor using dynamic binary translation and executes a number of guest operating systems using different hardware and device models 42

Role-based Access Control An access control mechanism that regulates access to resources in a computer system that depend on the user or group of users role in an organisation 34

Root of Trust for Measurement A tamper proof component that boots at the beginning of the boot process and measures all other elements booted after it 45

Root of Trust for Reporting A trusted component that provides reports of any integrity measurements that have been make, while attesting to the platform configuration 45

Root of Trust for Storage A trusted component that stores either data or keys and supports confidentiality and integrity protection 45

Software as a Service A higher-level software service layer for cloud computing, in which the tenant can control only a limited set of application configuration settings and can only use applications and services the cloud provider offers 26

Trusted Computing Base Contains the set of all the hardware, software, and firmware components that are critical to establishing and maintaining the security of the computer system 14

Trusted Execution Environment An environment that resides inside a main processor, executes code at the same time as the operating system in an isolated environment and guarantees that the code and data inside the Trusted Execution Environment are protected in terms of confidentiality and integrity 4, 58

Trusted Platform Module A microcontroller that provides storage and generation of encryption keys, remote attestation, binding and sealing of data used to authenticate hardware devices 11, 58

Type Enforcement Access to resources in a mandatory access control system is governed using a subject-access-object set of rules 34

Unikernel A single-address-space executable image that considers application code and device drivers as libraries and compiles them with a kernel and only with the required operating system libraries into a single binary file v

Virtual Machine An image that virtualises the software and hardware of a real computer machine v, 1

Virtual Machine escape Is the process of breaking out of a virtual machine and interacting with the host hypervisor in order to gain access to the other virtual machines that the hypervisor manages 1

Virtual Machine Monitor A software component in virtualisation environments that creates, manages and governs virtual machines, while virtualising and sharing the resources that the virtual machines require to fulfil their purpose 28,

Virtualized Environment Neglected Operations A security vulnerability that resides in the virtual floppy driver code used in virtualisation platforms. It enables an attacker to perform a Virtual Machine escape to access the host system and a potential privilege escalation to access the local network and adjacent computer systems 1

Xen Security Module A security framework for Xen hypervisor that enables a system administrator to specify permissible interactions between virtual machines, the hypervisor, and resources 6, 89, 161

Index