Hillock: an optimised and lightweight kernel to improve Qubes OS and security in cloud services.

Sergio Miguez Aparicio

4th Year Project Report
Electronics and Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

Lightweightness is a fundamental property and goal to achieve in many virtualisation environments, especially considering multi-level secure desktop OSs (like Qubes OS), serverless computing or embedded virtualisation. Focusing on multi-level secure systems application, operating systems achieve better isolation through compartmentalising processes in different virtual machines, which implies long wait times and inefficiencies. The project aims to achieve "lightweightness" (better boot up times, reduced memory and disk footprint) and reduce the high demand for resources required to run an entire OS system. We present Hillock – a highly optimised x86 monolithic kernel tailored for the Qubes OS secure desktop environment, running on top of the Xen hypervisor.

Hillock has been developed through an iterative process, where non-required configuration, modules, and drivers were removed. Afterwards, the kernel was tested against the most common alternatives in the Qubes OS environment, including the Qubes OS PVH templates, Debian minimal (PVH) or Alpine Linux (HVM). More than 650 boot ups of different configurations were recorded to check Hillock's improvement. Finally, our kernel was proven to be 1.29 times faster than the fastest alternative (Debian Minimal PVH) and 1.80 times lighter than the lightest alternative (Alpine Linux), becoming a starting point for further research.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Sergio Miguez Aparicio*)

Acknowledgements

Firstly, I would like to thank my family, Sergio, Lis, and Esther; for allowing me to reach this far, growing me up as I am and supporting my decisions.

I could not be more grateful to my dissertation supervisor, Antonio Barbalace (University of Edinburgh), who trusted me and gave me great flexibility and motivation to research and work on the project I wanted, and my external supervisor, Pierre Olivier (University of Manchester), who guided and helped me in the process.

I also want to give a special thanks to my flatmate, Rodrigo Carnero, who supported me through the development of this dissertation and pulled me up in the worst moments.

Table of Contents

Chapter 1

Introduction

Data protection and isolation have been fundamental since multiple users were allowed to work on the same machine. Research and industry have rapidly evolved, developing solutions as complicated as virtually emulating an entire system, commonly known as a virtual machine (VM), or the newer container technology [1]. Software development and other domains such as Container-as-a-Service (CaaS), Function-as-a-Service (FaaS) or serverless computing have partially shifted to the latter technology or combined it with VMs, as it solved the portability and scalability problems. These benefits have allowed significant software development over the last few decades.

Unfortunately, from a security perspective, containers are not as isolated and secure as they might seem [25]. They were designed to share and depend on the host's kernel to avoid redundancy when scaling systems. As a result, containers have lower isolation and are not as secure as VMs, which makes this infrastructure an exciting target. Through different proven attacks, malicious users can escalate privileges and take control over other containers or even the host [20] [29].

Despite these facts, a VM needs an entire operating system (OS) image with an independent kernel. Fully emulating the machine allows more robust isolation. However, this comes at a tremendous resource cost. Adding on, the code (and attack) surface of any OS is constantly increasing to improve functionality, flexibility, and hardware compatibility. Oppositely, containers focus on being fast, portable and light; all these characteristics make them an attractive solution and the reason for their widespread in the cloud environment.

There are certain domains where containers are not the best option due to their reduced security and isolation. This is the case of multi-level secure (MLS) systems which specifically focus on encapsulation and prevention of information leakage. Qubes OS is a desktop OS which applies MLS procedures to ensure maximum security in a Linux desktop distribution. It is based on the Xen hypervisor and entirely relies on virtualisation; specifically, everything is run in different VMs named by domains.

In this paper, we present Hillock, a new tailored kernel for Qubes OS. Hillock addresses some of the most critical limitations of Qubes OS, the heavyweight nature of the (traditional) VMs it uses. In addition to improving the system's responsiveness through enhanced boot times, it also focuses on attack surface reduction, memory footprint optimisation, and essential resource requirements. In this aspect, the project has been tested against multiple distributions and has shown a drastically better performance.

The urge for better isolation in cloud environments makes Qubes OS an interesting testing system. Having proven that containers are not as secure as expected, the common industry standard is to deploy containers inside VMs [27]. Therefore, improving and optimising VMs and kernels to be as fast and light as possible could lead one day to the chance of running each critical service instance on its separate VM and, speaking clearly, deploying VMs as containers.

1.1 Contribution

The project focuses on researching and developing a lightweight virtual machine which could be highly scalable for multi-secure infrastructure that heavily relies upon virtualisation.

This thesis has contributed to the following:

- Research state-of-the-art unikernels, microVMs and lightweight monolithic kernels.
- Research the methodologies to reduce and optimise an operating system.
- Develop a tailored lightweight monolithic kernel – Hillock – which improved boot time and memory (RAM) footprint for the Qubes OS environment.
- Test and benchmark the custom system against commonly used distributions, including tailored optimised systems from the Qubes OS team.

1.2 Project Outline

The report is structured as follows:

- **Chapter 2 – Background literature**: presents multi-level secure systems, the Qubes OS environment and its characteristics; the Linux kernel: processes used to tailor it and its vulnerabilities. Introduces virtualisation technologies (Containers and VMs) and their weaknesses. The Xen hypervisor and security procedures for the cloud. Finally, explains the emerging alternatives (unikernels and microVMs).
- **Chapter 3 – Development**: explains in detail the development methodology, the tools used, and the development pieces required to complete Hillock.
- **Chapter 4 – Results and Discussion**: the results achieved are presented and discussed.
- **Chapter 5 – Conclusions**: provides a conclusion to the project, a summary of the main contributions and the further research steps that should be taken.

1.3 Project Timeline

The project can be divided into different phases, which have different aims. We currently present the results of the last of these phases: Hillock.

My initial was to rethink Qubes OS in order to minimize the overhead added by the qubes domains which run the entire OS and applications. With this in mind, I intended to bring unikernel technology and unikernelize applications to run as individual qubes. As explained in 2.4.1.1, unikernels are an emerging technology which combines the lightweightness, performance and portability of a container, and the isolation of a virtual machine. Therefore, unikernels would be the best alternative to improve Qubes OS performance without putting its security philosophy at risk.

We decided to work with Lupine Linux [17], a system which achieves unikernel-like performance and outperforms other unikernel alternatives. It was developed through *specialisation* and *limiting syscalls*, and has great flexibility to run multiple applications as it is still a complete Linux system.

The initial steps followed in this dissertation were:

- Familiarizing with Qubes OS environment and the operating system architecture documentation (includes Xen Hypervisor).
- Installation of Alpine Linux OS and a script to automate operating systems installation for Qubes OS.
- Try to connect Alpine Linux to the Qubes OS environment – This would help run unikernels systems without a dedicated terminal or window manager. (Failed after exhausting all possibilities). This is fundamental because, without this communication system, the unikernel would lose most of its lightweightness benefits. This made the project not viable.

Re-evaluation of the dissertation goal. A different alternative to bringing unikernels is presented. Using a similar approach suggested in Lupine Linux of kernel *specialisation*, we decided to develop the first tailored kernel optimised for Qubes OS based on Alpine Linux, which already could run in the Qubes' environment, to achieve boot up improvements and lightweightness.

- In-depth understanding of kernel customisation, compilation, and features.
- Recognising the features required to run the VM in the Qubes OS environment.
- Development of 50 optimised iterations of the Alpine Linux kernel. The final version was named Hillock.
- Write-up of automated benchmarking scripts.
- Create an automated Hillock installation script.
- Measure 650+ boot ups of different operating systems with different configurations to test our tailored kernel.
- Organise, interpret and evaluate data.

1.4 Project Effort

This project required a broad knowledge of all systems components, from Qubes OS architecture as well as how it integrates and interacts with other operating systems to an in-depth understanding of kernel features, its customisation and compatibility with Qubes OS.

Due to the complexity of the self-proposed project, I started working on it at the beginning of 2022's summer. Until October, I was mostly reading background literature and familiarising myself with Qubes OS dedicating **2 to 3 hours a week**. It was at this moment that I started developing **one or two days a week**, scripts to automate the creation of domains and installation of different operating systems. While during November and December, the project was paused due to exams and deadlines, I decided to work full-time for Christmas, where I dedicated approximately **90 hours** during the break. Through this period I tried connecting Alpine Linux to Qubes OS. After failing to achieve this, in January I decided to redirect the project and develop Hillock to achieve results. Most of the project presented in this dissertation was full-time developed and tested from the beginning of the semester (January 16th) until approximately Flexible Learning (February 20th), where I dedicated **105+ hours**. Due to a health collapse, I had to completely stop the project and university for half a month, despite having further ideas and development pieces in progress for the project, since early March, and due to my health circumstances, I decided to focus on the write-up of the dissertation, where I dedicated approximately **150 hours**, until its deadline in April.

The development process to present the Hillock prototype required substantial compilation, debugging, testing and tuning time.

Estimate time dedicated after re-evaluation of the project:

- Research on kernel development and customisation: 20 hours.
- Development, compilation, and installation of the kernel: 30 hours.
- Install and adapt 10+ OSs in Qubes OS to test in similar conditions: 8 hours.
- Develop scripts in each OS to record timing data: 2 hours.
- Acquiring boot up and memory usage data of 8 OSs to compare results. More than 650 VMs were booted up in the process: 35 hours.
- Adapting, understanding, and expressing the data in graphs: 10h.
- Dissertation write-up: 150h.

This means that during January and February, the student dedicated **105+ hours** just to **developing the kernel** and **acquiring data**.

Finally, **the student dedicated approximately 425 hours in total to this project**.

Chapter 2

Background literature

This chapter presents previous essential research related to the project developed in this dissertation.

It has been divided into:

- Multi-level secure systems and Qubes OS – introduces MLS systems and Qubes OS as well as characteristics and requirements.
- The Linux kernel – presents the current state of kernel development, tailoring methodologies recently developed to achieve better performance and reduce attack surface while, at the same time, ensuring secure and isolated systems.
- Cloud environment and muti-tenant systems – is a section which covers the different virtualisation technologies used in scale, from containers and virtual machines to the Xen hypervisor. Various weaknesses are discussed, putting particular emphasis on security in this environment. Finally, multi-level secure architecture and the costs to achieve such protection in the cloud are presented.
- Emerging alternatives: Unikernels and microVMs – present the latest virtualisation techniques which could shape the cloud's near future (and already do) as they are faster and lighter while being better isolated and secure.

2.1 Multi-level secure systems and Qubes OS

According to [14], multi-level secure (MLS) systems are designed to prevent the leakage of highly classified information. Those systems that implement the Bell and LaPadua [12] security models. This implies individually identifying all the processes in a single system at security levels. To prevent a Trojan horse attack where information would be pushed to lower levels, a no-write-down policy is imposed, which means that higher-classified processes would be allowed to read lower levels but are not allowed to make changes to those files.

Those that keep separated data in different classifications and restrict access to different levels are also considered MLS systems. In any case, safely sharing information is a complex challenge as information must be replicated on different levels, drastically increasing the storage required.

In terms of applying multi-level secure functionality to hypervisors, there are two main ways to implement them [14]:

- Pure isolation hypervisors divide the machine resources into different isolated partitions and block any resource sharing. An excellent example of this isolation methodology is currently seen at the Z series servers from IBM.
- Sharing hypervisors: those where virtual machines can share resources (virtual or physical disk). This allows communication and connection between them. It includes the possibility of sharing files and removing replicated copies. A clear example of this methodology is KVM/370 [31].

Qubes OS is an open-source operating system which relies on the Xen hypervisor to create a secure desktop environment through heavily relying on virtualisation and virtual machines, denominated "domains". These VMs are classified into different trust levels. Overall, Qubes OS is a multi-level security system with a limited option of sharing files among the domains to optimise resources. Therefore, it could be said that it fits under the sharing hypervisor approach.

This operating system is described as "a reasonably secure Operating System" according to their official website, but it was its philosophy of encapsulating and isolating everything that caught our attention. The correct use of the distribution, advantage and distinctive characteristic is to run everything in different domains (or qubes). However, this makes the system power-hungry and frequently demands long wait times to make simple operations. All these complexities make Qubes OS less attractive for the average user but, at the same time, a perfect environment for improvement and research in virtualisation.

2.2 The Linux kernel

Operating Systems and containers rely on the kernel. As the core component which manages and interfaces processes and the computer hardware, it is fundamental flexibility and adaptable to any system. As a result, the kernel is continuously increasing to support new hardware and features, reaching 27.8 million lines of developed code in 2020 [19]. Moreover, according to the last article, fewer developers engage in the project each year, making it harder to develop and maintain, but more importantly, review and secure. A reduced number of developers implies more work per developer, making vulnerabilities harder to spot [40].

The generic distributed kernel has more than 10,000 configurable features [35] (1464 for x86 architecture [17]). The fast addition of new features due to the rapid evolution of hardware and the reluctance to remove legacy support because there are still many old systems running makes the kernel a massive project which, just through probability, might have many unknown vulnerabilities.

Considering the multi-tenant environment, all these factors make isolation a fundamental part of the System's security and a big problem when all containers depend on a single kernel. It should be considered the threat of having non-required additional privilege code running, which increases the possible attack surface of the system [18].

2.2.1 Tailored kernels to improve efficiency

The approach to improve performance and reduce the attack surface is making more specialised systems. The generally distributed kernels must boot everywhere; however, taking into consideration that the hardware and environment over which these systems would boot is optimal, there are plenty of elements that can be directly removed from the system, especially if the system is going to run on virtual machines which do not need to care about drivers more than the basic ones to accept vitalised hardware.

Multiple methodologies have been developed to reduce and tailor kernels:

- Among these, Tinyx [25], an automated tool which builds minimalistic Linux virtual machines tailored to run a specific application, can be highlighted. The system created is a middle point between a unikernel and a complete image. As a result, it is more flexible and supports more applications. LightVM was created using Tinyx; more interestingly, Manco et al., to achieve way better results which were limited by the Xen bottlenecks, researchers reimplemented Xen's tool stack, proving that further improvements can be made over the Xen project and virtualisation.
- Kurmus et al. [18] proposed an automated "trimming" kernel methodology to reduce its attack surface. They suggested making an in-depth analysing of a system using the kprobes tool to detect unused kernel code and a posterior removal. As a result, using their approach over a web server, they safely achieved an impressive reduction of 88% of its kernel.
- Directly removing features from the kernel in an automated manner [33] is an effective and more straightforward alternative than detecting and removing unused code.

2.2.2 Kernel vulnerabilities and security

It is common in large development projects or systems to find vulnerabilities. The Linux kernel is no exception. Automated tools with static [8] and dynamic analysis methodologies are being used to detect issues [36]. Chen et al. [7] suggested a new framework to detect bugs effectively.

Machine Learning models have also been trained to detect flaws [6]; however, they have yet to be successful. Since then (2017), ML has drastically improved, and probably through the new large language models, the results would be way different.

To reduce the Kernel attack surface, multiple mechanisms have been suggested. Linux Seccomp facility [9] is broadly used to limit applications making unnecessary syscalls. However, this mechanism has been primarily manual and tedious. Zhan et al. [42] propose a combination of static and dynamic verification tools to limit the kernel attack surface. Another essential publication [5] tries to automate the filtering process, which finally protects against more than 61% of the kernel exploits via syscalls. However, it is evident that directly removing unnecessary features from the kernel can prevent these attacks and make the system more optimised.

2.3 Cloud environment and multi-tenant systems

2.3.1 Containers

Containers are drastically increasing in popularity as a virtualisation technique for the cloud due to their lightweight, small latency, and fastness. Docker and Kubernetes are orchestration tools which ease the development, scalation and management of extensive container infrastructure. However, the main concern about this technology is that containers share the same kernel, making the system less isolated than a VM and, therefore, less secure. This is important in a multi-tenant cloud environment where multiple containers are dedicated to the same server for different purposes.

It has been proven that containers are less secure than believed [20]. The latter paper from 2018 shows 223 effective exploits, 56.82% allowed attacks from the inside with default configurations. These will enable the attacker to escalate privileges and access to the kernel, resulting in the complete loss of isolation.

Initiatives to improve container isolation have been recently developed as concerns have been raised. LSM (Linux Security Modules) [39] was presented in 2002 as an access control framework for the Linux kernel to allow access control models to be loaded as kernel modules. Bélair et al. [4] proposed in 2019 that adapting LSM (currently aimed just for the kernel, not containers) is a possible way to improve isolation for container environments.

2.3.2 Virtual Machines

The cloud allows great flexibility and the option of dynamically allocating resources as they are required. Virtual machines (as well as containers) are a crucial element in these environments. On-demand provisioning is really attractive; however, VMs, in contrast to containers, require a long wait time to boot and be ready to be used. Extensive research [26] analysing different cloud providers as well as operating systems. The recorded long startup times show that using complete virtual machines requires planning to get fast responses and make containers the attractive alternative. Therefore, developing new optimised alternatives such as tailored light kernels or unikernels is a requirement for future cloud environment improvements.

2.3.3 Xen hypervisor

The Xen project [3] was introduced as a hypervisor that enabled the secure and efficient sharing of hardware resources without compromising performance or functionality. The Xen virtual machine monitor is widely used for cloud systems from Amazon or Rackspace [23]. Due to the extensive interest in using Xen in these environments and its flexibility, multiple projects have been developed to dynamically improve the management and scalability of allocated resources according to business demands. Malhotra et al. [24] developed a dynamic algorithm to scale and distribute load for web services into multiple virtual machines in the cloud Xen environment.

Keeping correct isolation ensuring Denial of Service attacks or time channels is fundamental to prevent information leakage in multi-level secure systems, especially considering its usage in multi-tenant environments like the cloud. Gupta et al. [10] presented new initiatives and improvements. On the other side, experiments have shown that Xen can effectively prevent illegal memory access and can enforce isolation in the cloud [21]. However, it has been mentioned that Xen, aside from the hypervisor, has many other components which should be assessed if the intention is to build a highly-assurance system that could be trusted with sensitive data [11].

2.3.4 Security in the cloud

The fundamental requirement cloud faced to expand and become a possible alternative widely was having an as good as or superior security to the one it would replace (in a business or user environment). Currently, the cloud has been drastically adopted. However, researchers [2] have proven that the technology of some popular providers, such as Amazon AWS Cloud is badly secure.

2.3.4.1 Multi-level security systems and the cloud

Due to the mixture of different elements that do not fulfil certain security standards, contemporary cloud environments could be more secure. Creating the correct architecture is mandatory. A multi-level secure system approach could be a solution for these complex issues.

To achieve the most outstanding security equivalent to the guarantee achieved in Qubes OS, the environment must be implemented as multiple, isolated, and independent clouds, a multi-level secure cloud. To increase security, each would handle data of only one security level. However, this methodology could be more practical and inexpensive. Taking into account that this implies that sharing data between them would be done through copying data from lower to higher level domains [11].

2.4 Emerging alternatives: Unikernels and microVMs

Linux is the dominant OS across nearly every imaginable type of computer. Its kernel must be continuously expanded to support a wide range of functionalities. Raza et al. [30] conclude that unikernels might be the next evolution for Linux. However, another alternative is also rapidly being implemented, microVMs.

2.4.1 Unikernels

Unikernels are a new generation of virtualised systems that can be the alternative to container technology, providing complete isolation, lower memory footprint, and even faster boot times [25]. However, they could be better regarding being flexible and general. Some projects such as OSv [15] or HermiTux [28] allow running applications without the need to modify their build but lose on specialisation opportunities. Nevertheless, due to unikernel's significant advantages, it is thought to be the technology to achieve better serverless computing services [16]. The paper by Xavier et al. [41] evaluated KVM (virtual machines), Docker containers and OSv (unikernel) in a cloud platform. It proved that unikernel outperforms the rest of the alternatives.

MirageOS [22] is another unikernel that stands out as it was developed to run in the cloud using a Xen environment. The project has been substantially expanded, adding around 100 open-source libraries, which include a wide variety of functionality [23]. Moreover, MirageOS is currently being integrated into commercial products such as Citrix XenServer [32]. Having a similar approach, the OSv unikernel was built from scratch, providing a specialised API for cloud applications [28]. Additionally, Kuo et al. [17] presented the Lupine unikernel, an optimised system that has drastically reduced its size and features to achieve outstanding performance. The base image of the systems has only 283 features. However, Lupine has about 833 features if the application-specific options are included. This is impressive, considering a generic kernel for x86 has about 1464 features.

Interestingly, the research from Williams et al. [38] proved that unikernels could achieve similar isolation as virtual machines without requiring virtual hardware abstraction if they are run as processes. As a result, hardware requirements can be further reduced, the kernel size can be reduced by half, and tools for debugging or management can be directly used over them. Additionally, it further improves the great performance unikernels achieve.

Unikernels could optimise performance and become a solution for big data processing applications. Azalea-unikernel has been recently developed to be used in the Azalea multi-kernel operating system [13]. Due to cache coherency, monolithic OSs have been proven inefficient for scalable many-core systems. Moreover, it has been proposed unikernels as a solution to increase performance, minimising context switching and TLB flushing.

2.4.1.1 Unikernel's security

Containers have been recently the solution to achieve immediate deployment and reduced costs. The main downside of this technology is that they require a complete OS underneath to be launched. As a result, in the end, it increases the attack surface of the applications. Oppositely, unikernels are entire self-contained environments with the minimum features, reducing the attack surface simultaneously in this way. The aim of reducing and making hyperlight systems has a cost. Researchers have found multiple latest version unikernel projects, among them MirageOS or IncludeOS, to be vulnerable to numerous memory attacks [34] as they lack do not have ASLR, page protection or stack canaries implemented. This can be easily solved; however, it is a reminder that reducing features can lead to vulnerabilities.

2.4.2 MicroVMs

MicroVMs, or lightweight virtual machines, are a new technology that brings together container light benefits, security, and isolation from virtual machines. Among the most popular, we can find AWS (Amazon Web Services) Firecracker or Google gVisor. The methodology used to create these microVMs is customising the Linux kernel and shrinking peripheral devices' support. Moreover, this technology boots up whole kernels per VM. There is a performance trade-off if it is compared to containers. The research by Wang [37] analyses microVMs and concludes that they could be used for IaaS and PaaS platforms. However, unikernels would be better suited for FaaS. This dissertation used a similar approach to develop Hillock; technically, it could be considered a microVM.

Virtualisation and the cloud continue evolving, moving towards lighter-weight virtual machines – microVMs. However, researchers have proven the incompatibility between the fast-booting requirements of microVMs and the methodologies used to complete a kernel address space layout randomisation (KASLR). As a result, they have suggested that a re-evaluation should be done regarding the tasks left to the VMs and the ones monitors should handle. To see further improvements in this technology might imply systems redesign in their totality [16].

Chapter 3

Development

In this chapter, the student presents different elements required in Hillock's development process. The chapter is divided into five sections which are as follows:

- Development and benchmarking hardware – the computer hardware used for the development and testing is introduced. Moreover, general settings that can affect the systems' performance or results are stated in this section.
- Preparing the Qubes OS development environment – the section gives concise guidance and explains the student's methodology to develop the project. It introduces automated scripts that facilitate basic repetitive processes required for proper testing of the systems.
- Custom kernel development – the most relevant section of Chapter 4, the detailed explanation of the kernel development process used and the steps to building a new version.
- Hillock's tailoring kernel process – the student presents the methodology used to develop Hillock and introduces an easy and automated script to install the final kernel in any VM.
- Benchmarking – the benchmarking environment and the testing processes are presented. This section focuses on measuring boot up times and memory resources used.

3.1 Development and benchmarking hardware

All the project has been developed, and results obtained in an MSI GS66 STEALTH with the following specifications:

- i7-12700H (4.70GHz, 14 cores, 20 threads and 24MB cache)
- 32GB DDR5
- 1TB NVMe SSD
- RTX 3070Ti

All benchmark results were obtained running Qubes OS over bare metal. Due to the system's high isolation philosophy and heavy reliance on virtualisation, installing Qubes OS on a virtual machine is firmly not advised.

It is essential to mention that all the tests were done with the performance setting activated and the machine plugged into the AC.

We strongly recommend using a powerful machine to run Qubes OS – especially the use of SSD and large RAM, which critically affect waiting times and user experience.

3.2 Preparing the Qubes OS development environment

To install Qubes OS, following the guidelines given on the Qubes OS documentation is hardly recommended as some configurations are different to other operating systems installation. The user must create a bootable USB drive with an OS image and install the system. Selecting the wrong settings when creating the bootable USB or during the OS installation will cause the system not to boot.

3.3 Creating domains/qubes to install operating systems

Each virtual machine at the Qubes OS environment receives the name 'qube' and runs as a domain "DomU", which is entirely isolated and managed by the main domain/system, also called dom0. Due to the project's scale and to avoid repetition, the student developed a bash script which could create a tailored qube ready to be used when run in the dom0's terminal.

3.3.1 General operating system installation process in Qubes OS

First, the system's image (.iso) must be downloaded in a qube with internet access (not all have). The "untrusted" qubes is recommended for this task and will be the "qube_source".

After this step, the user can use the Qubes OS GUI to create and tailor the qube. However, the alternative script, which uses the qubes API, automates the process.

Listing 3.1: Automate VM creation

```
#!/usr/bin/bash
qvm-create --class StandaloneVM --label red --property virt_mode=hvm /
    <qube_name>
qvm-prefs <qube_name> memory 2048
qvm-prefs <qube_name> maxmem 2048
qvm-prefs <qube_name> kernel ''
qvm-start --cdrom=<qube_source>:/home/user/<distro.iso> <qube_name>
# Manually install the ISO
```

After the installation process, the RAM and vCPUs can be further modified using the following commands:

Listing 3.2: Limitting resources of a qube

```bash
#!/usr/bin/bash
qvm-prefs <qube_name> memory <new_RAM_value>
qvm-prefs <qube_name> maxmem <new_RAM_value>
qvm-prefs <qube_name> vcpus <new_numb_vCPUs>
```

3.4 Custom kernel development

Alpine Linux is defined as a lightweight and security-oriented Linux distribution. It comes with a light kernel version by default. Due to all these characteristics, it was chosen to be the base system upon which further improvements and optimisations would be developed.

To get familiarised with kernel customisation in a friendlier environment and ease the development process, the student initially began this projects section in a Oracle VirtualBox 7.0 VM running over a Windows environment.

After installing Alpine Linux and downloading the kernel source code 3.4.2, the student found a lack of guidance to make a custom kernel in this distribution. Therefore, a generic Linux kernel customisation approach was used.

3.4.1 Required packages

The following packages must be installed to start the development:

- alpine-sdk
- ncurses-dev
- bison
- flex
- bc
- perl
- libelf
- elfutils-dev
- findutils
- linux-headers
- pkgconfig
- installkernel
- xz

3.4.2 Acquiring a base kernel

The raw kernel source code must be downloaded to develop the custom kernel. In this case, the latest kernel version available at that time (Linux-5.15.91-lts) in a tar-compressed format from the official Kernel Website was used.

Listing 3.3: Getting the 5.15.91 kernel version

```
#!/bin/bash
KERNEL_VERSION=linux-5.15.91
cd
wget https://cdn.kernel.org/pub/linux/kernel/v5.x/$KERNEL_VERSION.tar.xz
tar -xvf $KERNEL_VERSION.tar.xz
cd $KERNEL_VERSION
make mrproper # clean in case there are any previous compilation files
```

3.4.3 Compiling the kernel

To compile the kernel, ‘make’ must be used. All the ‘make’ options can be found in the make README. The essential options of ‘make’ that should be taken into consideration at the time of building the kernel are:

- clean: which removes most generated files but keeps the config file.
- mrproper: which removes everything.
- menuconfig: used to customise the config file through a menu-based program.
- defconfig: generates a default config file.

Further examples of how to use these commands can be found in the given scripts 3.4 and 3.5

3.4.4 Making changes to the kernel

To start the development, the following commands are recommended:

Listing 3.4: Start kernel customization

```
#!/bin/bash
make ARCH=x86 defconfig # replace x86 with your specific CPU architecture
make menuconfig
```

It must be highlighted again that this tailored kernel is being done for the x86 architecture. More information about other architecture files can be found in the make README.

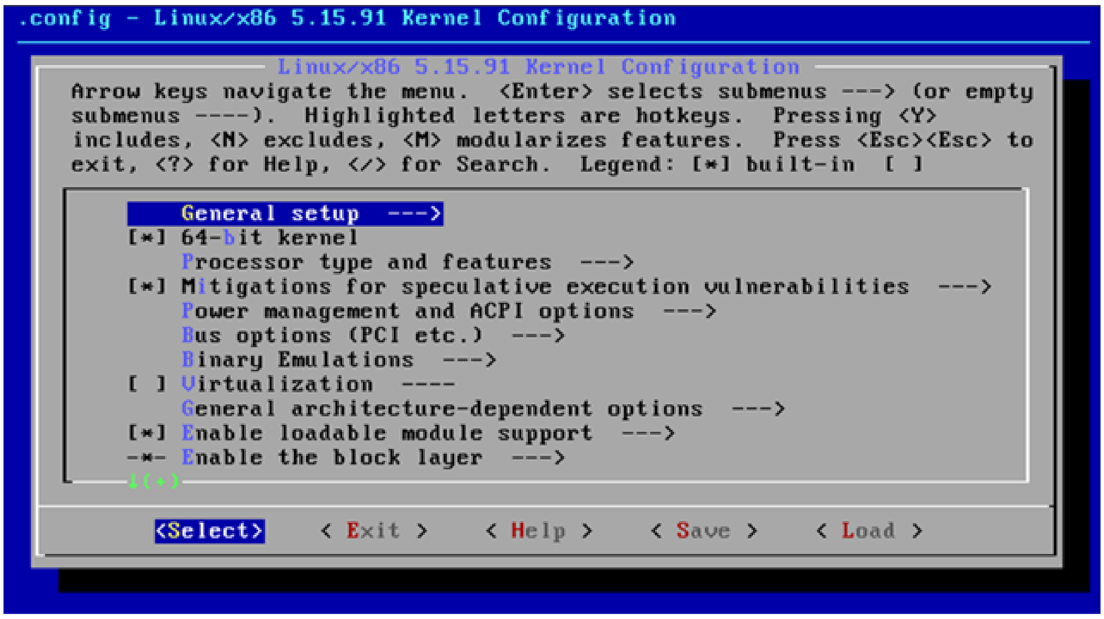
To build a correct kernel, ‘make’ uses a ‘.config’ file. As explained in [33], the kernel config specifies which kernel functions must be enabled in the compilation, determining which parts of the source code will be compiled. To make changes to it, the menuconfig command must be used. It will display a ”user-friendly” menu where different capabilities can be added or removed from the kernel.

The different kernel configuration alternatives can be:

- Added to the kernel.
- Removed from the kernel.
- Added as a Module.

These will be selected using the keys Y (add), N (remove), and occasionally, M (in case it can be added as a module). Once selected, code pieces can be directly built into the kernel or as a module. Kernel modules give great flexibility and reduce the amount of code constantly running as part of the kernel, as it loads and unloads code upon demand. However, if some code – a module – is not always required, for an optimised system is not vital and, therefore, just a waste of resources in most situations. More about it in 3.5.

Figure 3.1 shows the general menu displayed when menuconfig has finished compiling.



```
.config - Linux/x86 5.15.91 Kernel Configuration
              Linux/x86 5.15.91 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

          General setup  --->
      [*] 64-bit kernel
          Processor type and features  --->
      [*] Mitigations for speculative execution vulnerabilities  --->
          Power management and ACPI options  --->
          Bus options (PCI etc.)  --->
          Binary Emulations  --->
      [ ] Virtualization  ----
          General architecture-dependent options  --->
      [*] Enable loadable module support  --->
      -*- Enable the block layer  --->
      ↓(+)

       <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 3.1: menuconfig screenshot - kernel customisation menu

Once the changes are done, the kernel must be compiled and installed:

Listing 3.5: Kernel compilation

```
#!/bin/bash
make -j $(nproc)
make modules_install -j $(nproc)
make install -j $(nproc)
```

The ”$(nproc)” option uses all the available processing units. It can be replaced with the number of threads dedicated to the compiling task. The more, the faster it will compile.

Using the highest value possible is highly recommended, as the kernel can take a few minutes to several hours, depending on the allocated hardware. In these experiments, 14 vCPUs allowed complete kernel compilations in approximately 2 to 4 minutes.

Something fundamental to keep in mind is that whenever changes are done over the kernel configuration file, the system must be fully recompiled. Therefore, making the correct changes and the right amount is necessary. This is required to optimise the number of compilation cycles.

A final side note from the previous script 3.5 - the third line is only required if the kernel configuration file has modules to be installed.

3.4.5 Updating the bootloader

To complete the new kernel installation, the bootloader must be updated. Alpine uses Syslinux as the default bootloader. It has caused issues in completing this task. Therefore, a commonly known bootloader – grub – was used, and installing the tailored kernel in Alpine Linux is recommended. Moreover, the other operating systems used to test Hillock's improvement use grub as their bootloader. Making this change helped normalise the initial start-up elements.

Listing 3.6: Setting up grub bootloader

```bash
#!/bin/bash
doas apk del syslinux
doas apk add grub
doas apk add grub-bios
grub-install 'DISK'
grub-mkconfig -o /boot/grub/grub.cfg
```

The 'DISK' parameter should be substituted with the partition that contains the boot information. In the case of the default installation of Alpine Linux in Qubes OS, it will be "/dev/xvda". In a Windows VM with Virtual Box, it will be "/dev/sda".

Every time a new kernel is compiled, it must be added to grub through the grub-mkconfig command.

3.5 Hillock's tailoring kernel process

Plenty of things can be customised in the Linux kernel config file. menuconfig divides all the possible configuration options into categories, and it is crucial to mention that setting the incorrect configuration will make the system unbootable, unsupported by the hardware, or unfunctional.

The development process of the custom kernel was done iteratively. In each iteration, multiple settings were changed and checked that the system could still boot. As seen in the graph 3.2, it is in the initial iterations where the changes made a big difference in the kernel size.

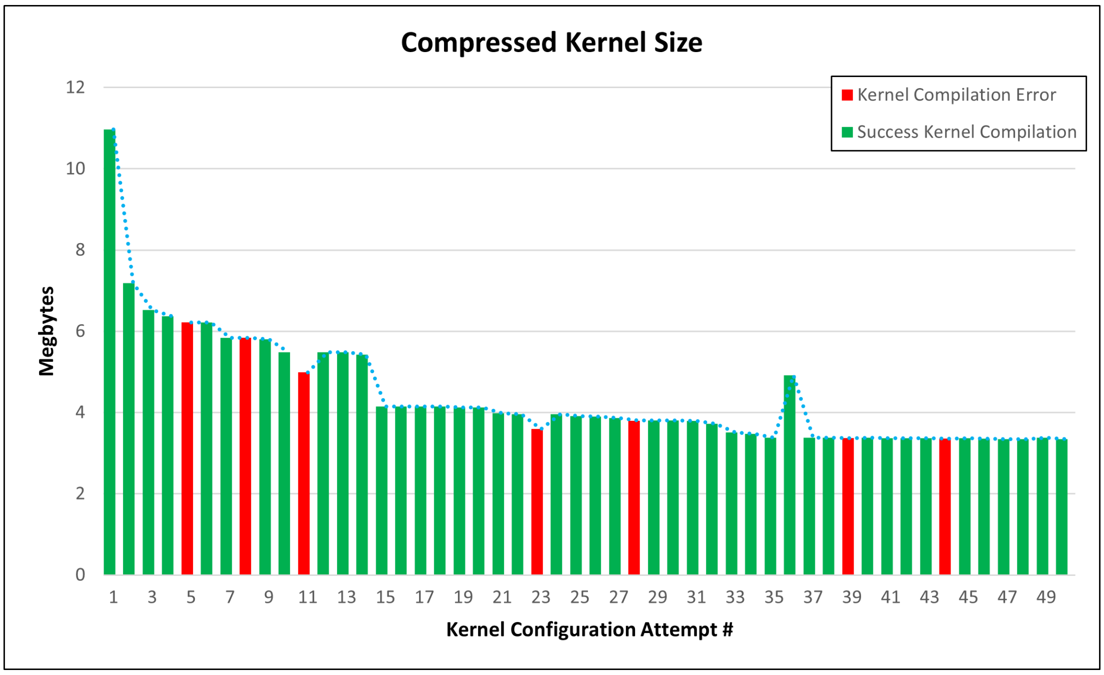


Figure 3.2: Hillock's kernel size per iteration.

As mentioned earlier, the kernel has many configurations and options to give great flexibility and ensure the boot up of the system wherever it is being deployed. This means that the default kernel comes with an uncountable number of lines of code which are entirely unnecessary. The first initial steps taken in the development of Hillock were removing all the non-x86 drivers that were preloaded. The kernel is custom-made for an Intel processor. Therefore, all AMD drivers can immediately be removed. The system is running on the latest hardware from 2022, which meant that all options to support legacy systems were removed.

It is evident that all those settings and drivers related to Xen or Intel always remained untouched.

All the configurations were classified depending on their description, dependencies, and impact on the system. Options can be divided into configurations which:

- ”Is safe to remove”.
- “If unsure, it can be removed”.
- “If unsure, better to be left”.
- “If not left, could break the system”.
- “If not left, break the system”.

Most of the information needed to classify each option in this list is given by the description shown when the configuration is selected.

During the initial iterations, the student removed most features from the first two categories. All these changes together already brought much improvement to the system. In most cases, elements from these categories are modules and drivers, which clearly do not relate to the hardware being used. It was mentioned in the Linux kernel configuration description what modules are. It is evident that in a large kernel with no restrictions, modules are a great solution to minimise the amount of code running with privileges. Conversely, having less code is fundamental for a small kernel that tries to optimise all its aspects. This comes as a cost, being more restrictive or tailored. In most cases, modules, as stated in their definition, are pieces of code loaded on demand. Therefore, it is a way of reducing unused code. For our use case, most of the modules could be removed.

Afterwards, those configurations which suppose a higher risk was removed in small chunks to keep track of the required elements. A list which records all the removed features per iteration can be found in appendix A.

Among the removed elements, it can be highlighted networking, IO (except keyboard and graphics), PCI, I2C, hardware monitoring support, virtualisation (except Xen), alternative filesystems, legacy compatibility, unnecessary libraries, kernel debugging and testing. Moreover, no initial RAM filesystem is used during boot time. Through this last option, decreasing the required dedicated memory mentioned tested in 3.6.3 was possible.

3.5.1 Automatic installation of Hillock

Due to the necessity of testing the system in multiple qubes, the student elaborated a script which allows the automatic installation of Hillock's kernel in a few simple steps.

To get Hillock, clone the GitHub repository (https://github.com/SergioMiguez/Hillock) and run the 'kernel_install.sh' file inside Custom Kernel Development with root privileges.

The script will:

- Ask for the base kernel version to be downloaded.
- Install all the required packages.
- Download and decompress the base kernel.
- Acquire Hillock's config files.
- Compile the Hillock.
- Install Hillock.
- Update the bootloader.
- Clean all the generated files.
- Reboot.

At each step, the script prompts the user questions to adapt the installation or to skip specific steps.

We strongly recommend running this script in an empty/cloned VM with all files backed up first. In the case of failure due to missing drivers or running it in an incompatible system, the VM might not boot. Run this at your own risk!

3.6 Benchmarking

Two main factors are being tested in this project: boot up time and memory (RAM) resources used. It is essential to mention that all benchmarks are always done in VMs with two dedicated vCPUs. The same performance profile is selected, and the laptop is plugged into AC and static dedicated RAM to ensure the same conditions are shared among all the systems and measurements.

3.6.1 Benchmarkign environment setup

To compare the performance, boot time, disk and memory footprints, the student installed multiple operating systems and progressively reduced the resources given to each machine until they were not bootable.

The following OSs were tested:

- Hillock (tailored kernel developed for Qubes OS) – HVM
- Alpine Linux 3.17 Standard - HVM
- Debian 11 - HVM
- Fedora CoreOS - HVM
- Debian 11 - PVH Qubes OS Template
- Debian Minimal - PVH Qubes OS Template
- Fedora 36 - PVH Qubes OS Template

All the previously mentioned OSs did not have a desktop environment, just a terminal to improve lightness and boot up speeds.

3.6.2 Boot up benchmarking

The boot up benchmarking was done using an automated process, and three different phases can be recognised at the time of launching a qube:

- Time from Xen boot-up command until Xen hypervisor starts the machine, time used to dedicate resources.
- Time since VM is started until the Xen hypervisor launches the VM's window.
- Time since VM's window is launched until OS fully boots up (user login screen).

To get more insightful figures, each OS was tested in an iterative process reducing its VM resources until the system could not boot. Then, **to compare boot times properly**

and fairly, the OS was tested with its empirically found minimum resources and the minimum specifications of the other operating systems.

Distribution	Dedicated RAM (MB)	Boot
Alpine LTS	400	Stable (+20 correct boot ups)
Alpine LTS	200	Stable (+20 correct boot ups)
Alpine LTS	100	Error (fail to boot up)
Alpine LTS	150	Unstable (4/20 correct boot ups)
Alpine LTS	175	Unstable (9/20 correct boot ups)
Alpine LTS	185	Unstable (12/20 correct boot ups)
Alpine LTS	190	Stable (+20 correct boot ups)

Table 3.1: Alpine LTS empirical minimum RAM to boot up

The following bash script uses the qubes API to measure in an automated way the time it takes for the Xen hypervisor to launch a machine screen to boot. The process starts and shuts down the VMs ten times, automatically printing the time used in each phase in the terminal.

Algorithm 1 Benchmarking boot time 10 times.

```
1: qube ← qube's name
2: goto Benchmark.
3: function BENCHMARK:
4:     shutdown all qubes
5:     for i=0 to 10, step 1 do
6:         sleep 15 seconds
7:         goto InnerTime
8: function INNERTIME:
9:     start qube
10:    print time to start qube
11:    if PVH then
12:        launch terminal
13:    shutdown qube
```

The test slightly differs depending on whether the OS runs in HVM or PVH mode.

- **HVM**: Qubes OS runs generic operating systems, fully emulating the machine. If the system is not directly compatible or has the correct infrastructure of Qubes, the VM will not be able to communicate with Qubes OS. Therefore, in these cases, an inner script is usually required to report the boot up time (third phase).

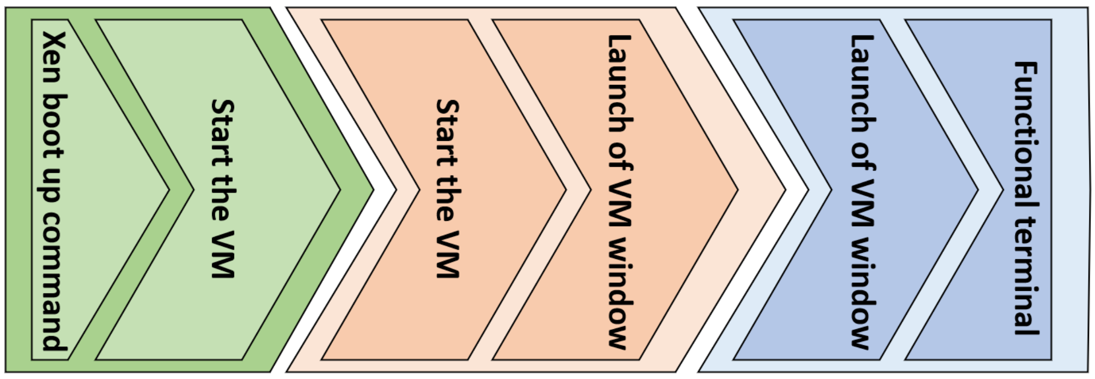


Figure 3.3: Benchmarking boot up phases of HVM virtual machines

- **PVH**: Qubes OS communicates and does not launch a window for the VM until the user requests it. The script is adapted to launch and report the boot up time. In this situation, the recorded times are Xen boot up command until the VM has started and the time since it has begun till the machine launches a functional terminal. This method adapts the time measurements to compare them similarly to an HVM VM.

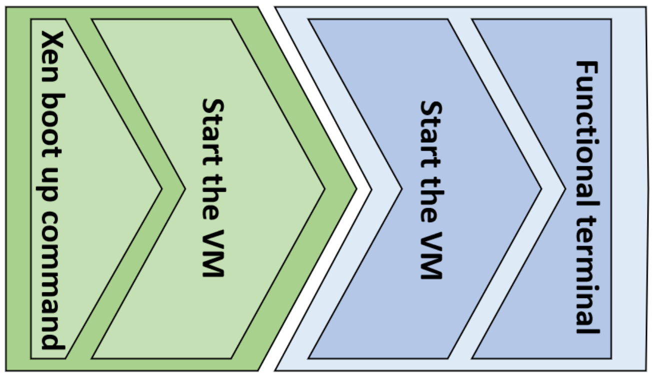


Figure 3.4: Benchmarking boot up phases of PVH virtual machines

To sum up, these sets of tests try to analyse the time different operating systems take to boot up. Each system was tested with the same VM resources. To get a better understanding, the dedicated RAM was increased progressively, and for each memory configuration, the systems were booted ten times, making a total of over 650 recorded boot ups. Results and discussion of time improvements can be found in sections 4.1, 4.5 and 4.6.

3.6.3 Memory (RAM) benchmarking

Memory usage is crucial in the cloud and multi-level secure infrastructure. Since these environments run many VMs instances simultaneously, memory resources are a determining factor for scalability. When a VM is launched, the dom0 or the main OS of the system assigns some RAM for it. In Qubes OS, this memory can dynamically increase to a maximum or reclaim and decrease to a minimum determined in the qube's configuration. Therefore, taking a big-scale infrastructure, optimising RAM usage per VM is fundamental to make them cloud and multi-level secure friendly.

All qubes had the same memory settings to ensure fairness, and no dynamic allocation was allowed.

Due to the memory allocation setting, Qubes OS then sets that memory for the VM and assumes it is being used. As a result, if we use the Xen infrastructure to check memory usage through ***xl top***, it shows that the RAM used is constant to the qube's memory allocating in its settings, regardless of the OS running in the VM. The '***free -k***' command inside each VM is used to get accurate results for our benchmarking. This Linux command reports the current memory used in kilobytes. Multiple samples taken after the entire boot of the qube give an approximated idea of the base memory requirement.

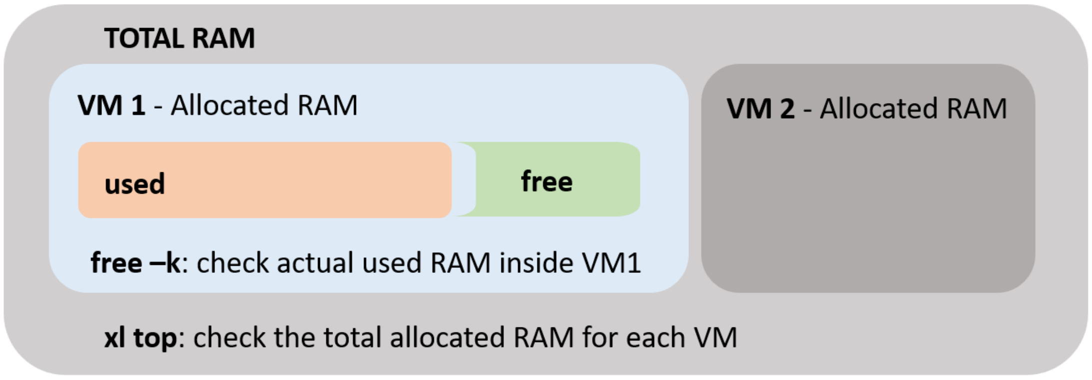

TOTAL RAM
VM 1 - Allocated RAM
used
free
free –k: check actual used RAM inside VM1
VM 2 - Allocated RAM
xl top: check the total allocated RAM for each VM

Figure 3.5: Benchmarking RAM in HVM virtual machines in Xen environment

It is commonly assumed that the system will be running over hardware far more capable than minimum resources; therefore, a common approach is to load the compressed kernel into RAM and decompress and boot it up from there, as it is much faster than any other disk. As a result, in the beginning, the system requires a minimum of dedicated memory to boot up, freed once the system booted up. For this reason, when cutting the dedicated memory to the minimum, the system might run into a memory error and kill the boot up.

However, at the same time, this means that more dedicated memory is required to start the system as the compressed and decompressed kernel must fit in memory. Nevertheless, there is a possibility to optimise this issue. Excepting tailored kernels which intentionally do not use this methodology, all operating systems rely on RAM to decompress the kernel.

As shown in table 3.1, Alpine can run with far fewer resources. However, the system becomes unstable, and the less resources are dedicated, the higher chance the system will fail. The following table shows the minimum resources required to run Debian 11 HVM:

Distribution	Dedicated RAM (MB)	Boot
Debian 11	400	Stable (+20 correct boot ups)
Debian 11	200	Error
Debian 11	250	Error
Debian 11	275	Error
Debian 11	285	Error
Debian 11	295	Unstable (14/20 correct boot ups)
Debian 11	300	Stable (+20 correct boot ups)

Table 3.2: Debian 11 empirical minimum RAM to boot up

Reducing the general kernel size is essential as it affects other system characteristics, which can determine the actual performance and viability in the environment where it should be run.

- Decompression time – with a smaller kernel, fewer files must be decompressed, gaining optimising the overall time. The best alternatives to improve this time is either choosing a fast-decompressing algorithm, a light decompression or, in the best case, not decompressing the system.
- Decompressed size – a smaller kernel implies reducing the actual boot up time as less code must be run.
- Amount of RAM used during and after the boot. – an optimised kernel will require less total RAM while booting up, reducing the minimum necessary resources to boot up. Moreover, after booting up, the RAM demand will be lower as fewer kernel functionalities must be loaded constantly.

To conclude, memory (RAM) benchmarking is a set of tests used to determine the base kernel RAM requirements in order to boot and measure its decompressed size. Further information about results can be found in sections 4.3 and 4.4.

Chapter 4

Results and Discussion

Indeed, the initial aim of substituting VMs in Qubes OS with unikernels to improve boot up times still needs to be accomplished. After the project got stuck, it was re-evaluated, and the goal was changed: making the first tailored kernel – Hillock – aimed for Qubes OS and proving that it is one of the best alternatives to this date.

In this chapter, the results of the development process are presented. It is divided into different tests and benchmarks performed:

- Average boot up times (of only the OS) 4.1 – is the section where Hillock boot up is tested directly against other operating systems. These tests do not consider delays caused by Qubes OS or the Xen environment.
- Compressed kernel size 4.2 – section where kernels from different OSs are compared. This only considers compressed sizes.
- Required RAM to boot up 4.3 – presents the empirical data which determines the minimum RAM necessary dedicated by different OSs to a VM to boot up correctly.
- RAM used by the kernel 4.4 – in this section, each kernel's amount of used memory is compared. This is assumed to be equivalent to a decompressed kernel size.
- Impact of increasing RAM 4.5 – here, boot up times for each system are presented while increasing dedicated RAM.
- Including Xen in boot up time benchmarks 4.6 – Finally, all the systems' boot up times are compared, considering the actual Qubes OS delays.

Finally, a last section 4.7 is added with further discussion of the results and the methodology used to achieve them.

4.1 Average boot up times (of only the OS)

Hillock shows an outstanding improvement in boot up times. The following table presents the average results (only for HVM systems) obtained using the boot up benchmarking methodology explained in section 3.6.2:

Distribution	Time (s)	Hillock's Percentage Improvement (%)
Hillock	3.818	-
Alpine LTS	12.745	-70.04%
Debian 11	6.437	-40.69%
Fedora CoreOS	20.570	-81.44%

Table 4.1: Average boot up times

Something essential to highlight is that these average times are the ones with each system's minimum stable boot up resources. As shown later in section 4.3, Hillock is booting up with much fewer resources and still being the fastest.

As can be seen, Hillock boots up on average in 70% less time than Alpine, its original system. It has been noticed that Alpine takes a long time to start the chronyd service, which is fundamental to synchronising the systems clock with NTP servers; and starting the VM network system. Besides not having drivers or unnecessary modules, Hillock's lightest version removes the network capabilities compiled into the kernel. This can be useful for running desktop executables/programs that do not require a network connection, increasing the system's security. Iteration 36 in the kernel development in figure 4.2 was developed to include networking capabilities to get an approximated idea.

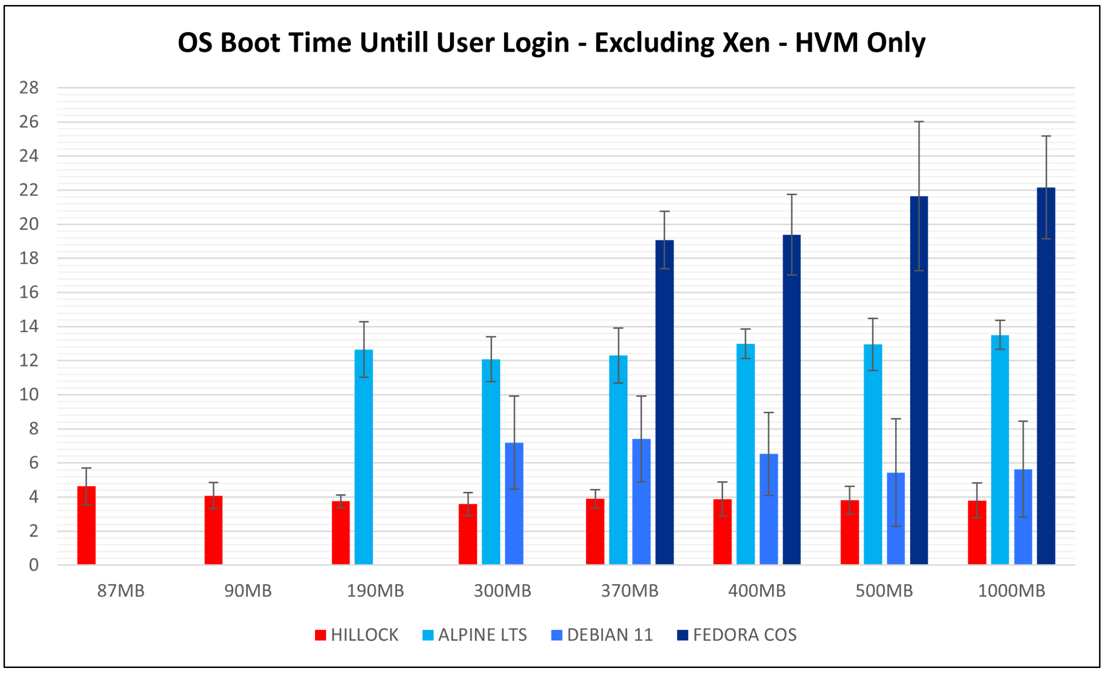


Figure 4.1: Comparing boot up times HVM

Other factors that significantly impact are the kernel size, which translates into more code to be run while booting up, or the type of kernel compression used. As shown in the next section, Hillock is the smallest kernel. Interestingly, Debian 11 boots on average faster than Alpine, probably due to the previously commented services, Debian probably runs more code as modules, which translates into kernel code just running when it is required and not during boot up; or running some services in the background, slowing the boot up time, but all these is uncertain and difficult to check.

4.2 Compressed kernel size

Through the iterative process to develop Hillock, unnecessary elements were removed. As a result, the kernel shrank drastically, becoming the lightest among the compared distributions. The following results 4.2 were obtained by checking each kernel file size in their respective /boot folder:

Distribution	Kernel Size (MB)	Hillock's Percentage Improvement (%)
Hillock	3.348	-
Default x86 kernel	10.577	-68.34%
Alpine LTS	8.225	-59.29%
Debian 11	7.701	-56.52%
Fedora CoreOS	13.000	-74.24%
Debian 11 PVH	7.019	-52.30%
Fedora 36 PVH	12.732	-73.70%

Table 4.2: Kernel size and Hillock improvement percentage

The following graph (4.2) shows the evolution per iteration of Hillock's size.

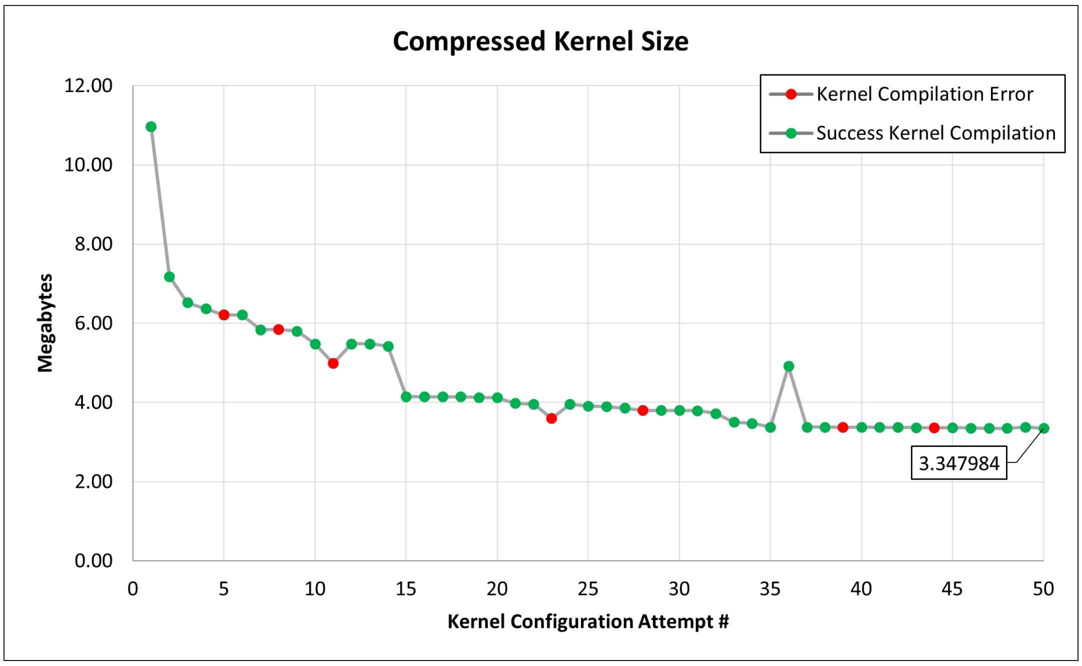


Figure 4.2: Compressed kernel size progress

We reduced the initial default Linux kernel from 10.577MB to 3.348MB, representing a 68.34% reduction. Moreover, figure 4.3 puts in perspective the actual Hillock kernel's size compared to other distributions. As previously mentioned, the kernel size slightly affects the decompression time and the memory required to boot the system.

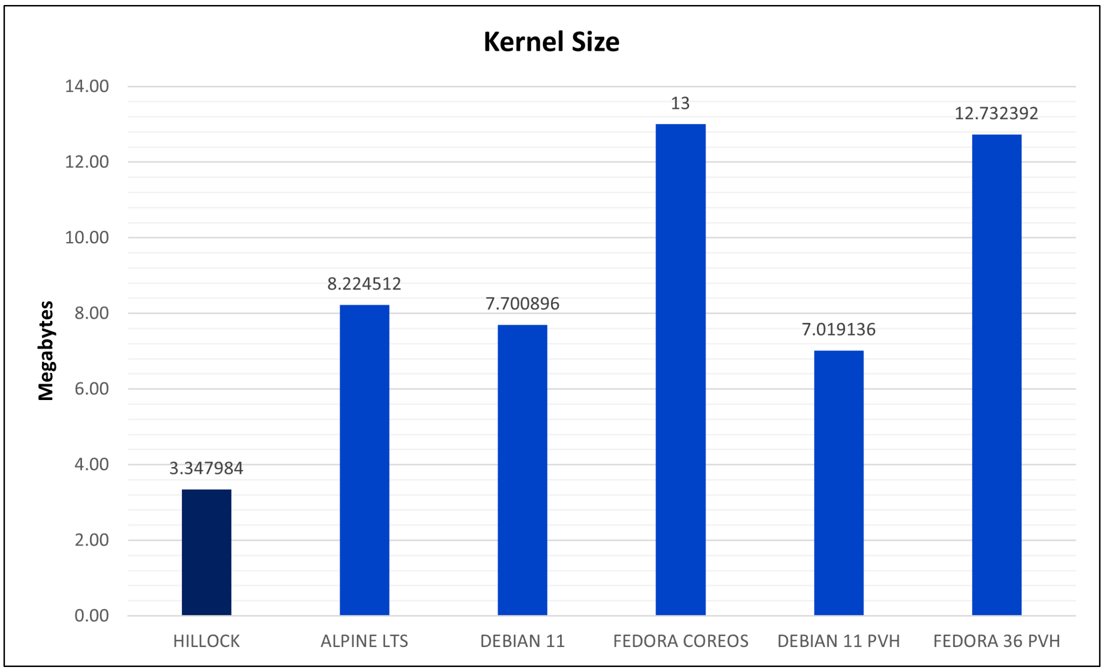


Figure 4.3: Comparing compressed kernel size

4.3 Required RAM to boot up

Each distribution was tested in an iterative process reducing the RAM dedicated until the VM was not bootable. The following table shows the minimum required RAM to achieve a stable boot up (without time-to-time memory errors) for each system. The values displayed were obtained through the iterative reducing resources procedure explained in section 3.6.2 and table examples 3.1 and 3.2.

Distribution	Memory (MB)	Hillock's Percentage Improvement (%)
Hillock	87	-
Alpine LTS	190	-54.21%
Debian 11	300	-71.00%
Fedora CoreOS	370	-76.48%
Debian 11 PVH	195	-55.38%
Fedora 36 PVH	210	-58.57%
Debian Minimal PVH	195	-55.38%

Table 4.3: Required minimum RAM memory to boot up

Hillock allows a stable boot up with the least resources among all the options. The fundamental improvement is against Alpine lts, the second lightest option. Hillock reduces and frees more than half of the resources (-54.21% or 103MB). This achievement is critical in a system where everything runs over VMs, as it allows more than double the number of parallel VMs running in the same system.

As mentioned previously, Hillock boots up, on average, the fastest and, at the same time, uses the least resources.

Interestingly, the Debian Minimal PVH requires the same resources as the full PVH version. Conversely, the Qubes OS developers have adapted both versions to optimise the communication and compatibility with the Qubes OS environment.

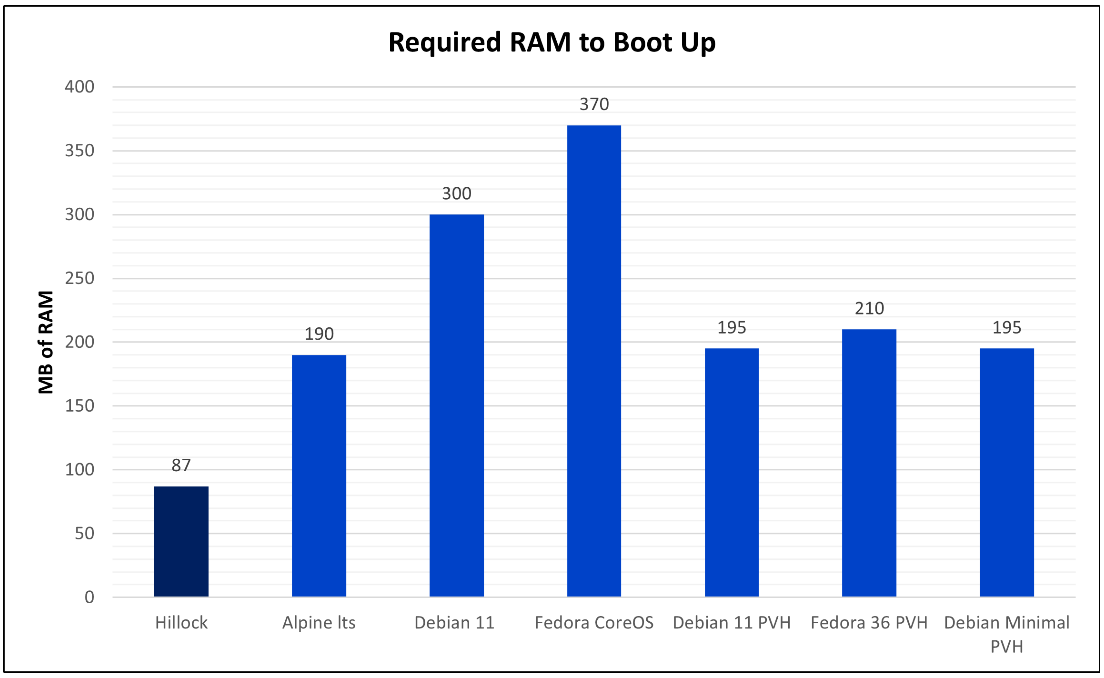


Figure 4.4: Comparing required RAM to boot

4.4 RAM used by the kernel

Aside from the required RAM to boot up the VM, the best way to compare the system's kernels is to measure the kernel's total size and the actual RAM footprint. The following table shows the obtained results through the RAM benchmarking procedure explained in section 3.6.3

Distribution	Memory (MB)	Hillock's Percentage Improvement (%)
Hillock	33.324	-
Alpine LTS	60.1	-44.55%
Debian 11	75.789	-56.03%
Debian Minimal PVH	145.148	-77.04%
Debian 11 PVH	272.536	-87.77%
Fedora 36 PVH	279.564	-88.08%

Table 4.4: Decompressed kernel size in RAM

The values remain almost constant despite the increase or reduction of dedicated RAM. These values include the swap used to store part of the kernel when the dedicated RAM is less than the one the kernel requires. That is why the PVH systems are allowed to boot with less dedicated RAM (shown in the previous section), despite their more significant kernel size.

Putting the figures into perspective, dedicating 300MB of RAM to all systems, the following graph shows how much RAM is just used by the kernel:

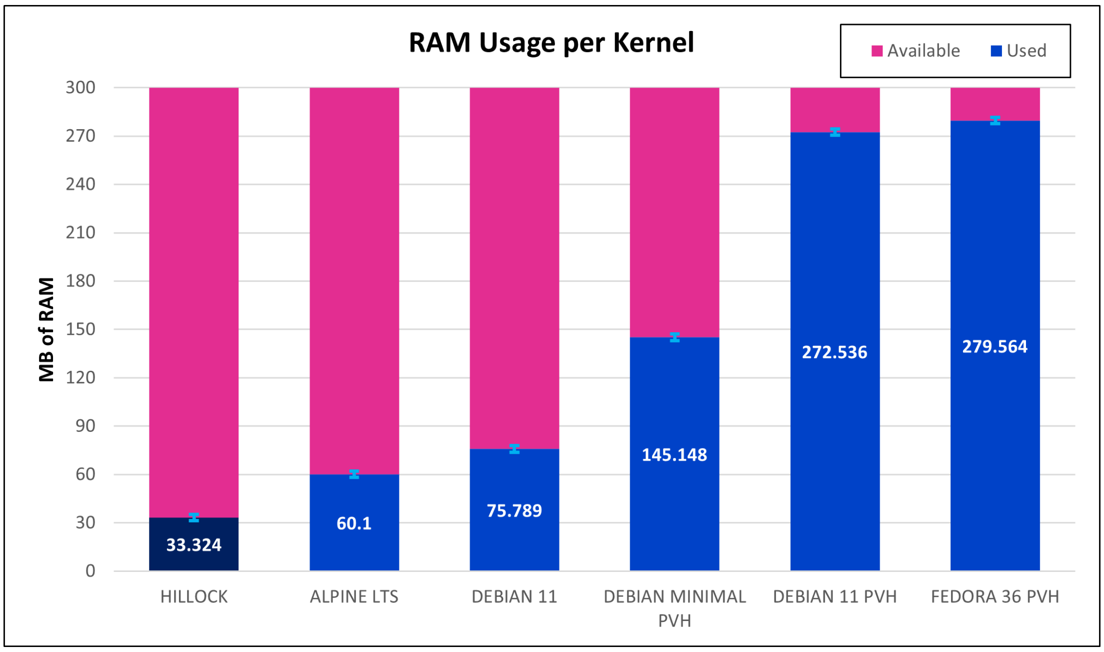


Figure 4.5: Comparing used RAM/decompressed kernel

Here it can be seen that the Debian Minimal use fewer resources than its full version. On the other side, PVH systems have a much larger memory footprint. This is probably due to loading all the modules required to work in the Qubes OS environment, as all these systems can be controlled from dom0. However, focusing on the HVM systems, Hillock's size is almost half Alpine's and further less than Debian's. Moreover, Hillock has drastically reduced the attack surface, and finding and fixing vulnerabilities in a properly maintained system would be easier.

4.5 Impact of increasing RAM

Through the automatic testing and using the previously mentioned average boot up time results and the required resources information, the graph 4.6 was developed

This graph only compares the time since the Xen hypervisor started the VM until there is a working terminal. As explained in the boot up benchmarking subsection, each system corresponds to 10 consecutive deployments of the OS. Moreover, it should be mentioned that the qube's memory is disposed of after each turn off, which is why, through the

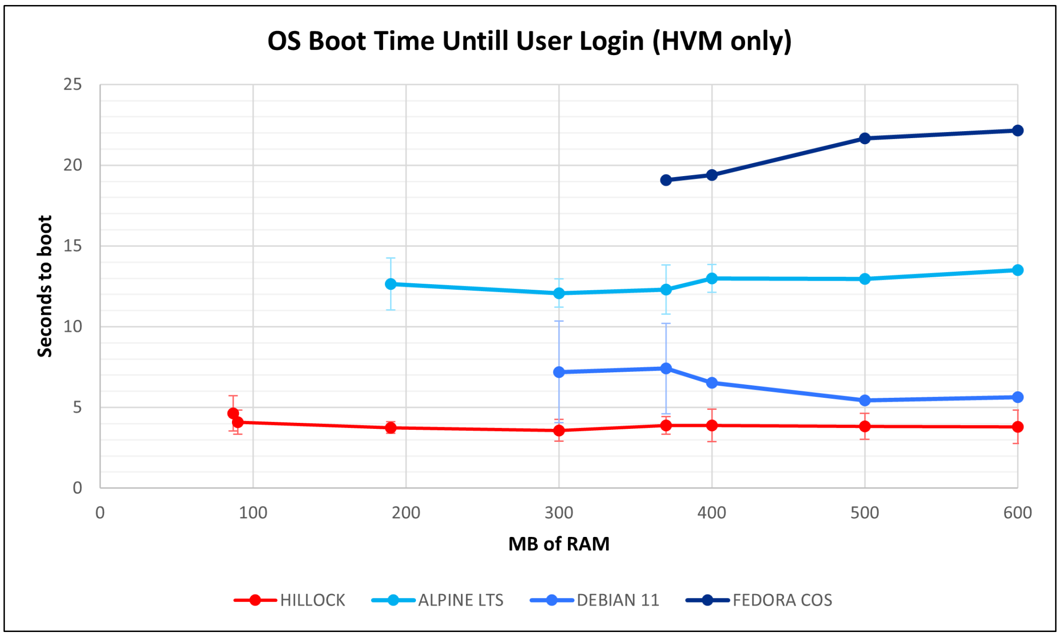


Figure 4.6: Average times to boot while increasing RAM

boot up iterations (and having loaded the OS into RAM), there is no improvement in that sense, and time remains constant.

The graph shows that Hillock outperforms the rest of the distributions, allowing the VM to boot up with only 87MB dedicated. It is followed by its predecessor, Alpine LTS, which requires 190MB; Debian 11, which needs 300MB; and Fedora CoreOS, which demands 370MB. Moreover, it not only requires an outstanding -54.21% memory resources than Alpine, or -71% than Debian 11; Hillock can boot up faster than all the other systems, taking 70.04% less than Alpine and 40.69% less than Debian.

4.6 Including Xen in boot up time benchmarks

As previously mentioned in the boot up benchmarking section (3.6.2), there are multiple phases that do not entirely rely on the VM's system but more on the resources dedicated to the machine. Table 4.5 presents the average measured boot up times obtained using the procedures explained in the boot up benchmarking section (3.6.2)

Distribution	Time (s)	Hillock's Percentage Improvement (%)
Hillock	17.80	-
Debian Minimal PVH	23.08	-22.88%
Fedora 36 PVH	28.06	-36.56%
Debian 11 PVH	28.90	-38.41%
Alpine LTS	31.82	-44.06%
Debian 11	36.80	-51.63%
Fedora CoreOS	51.84	-65.66%

Table 4.5: Average time to boot a VM in Qubes OS

Hillock boots up on average 22.88% faster (or 5.28 seconds) than the fastest available option in the Qubes OS environment, the Debian Minimal PVH system. Moreover, compared to its parent and predecessor project, Alpine lts, it has achieved an average improvement of 44.06% (or 14.02 seconds).

Comparing the boot up times of the different systems, the 4.7 graph shows the average time to boot a full VM in the Qubes OS environment.

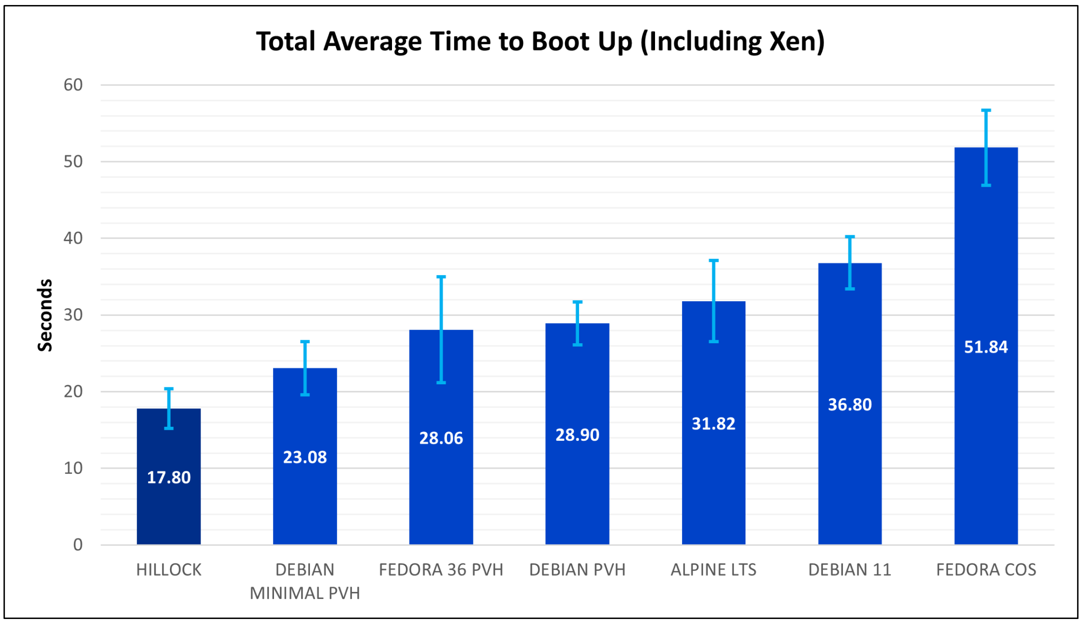


Figure 4.7: Total average time to boot including Xen environment requirements

Figure 4.8 shows a more in detail comparison between Hillock, Alpine lts and Debina Minimal (PVH).

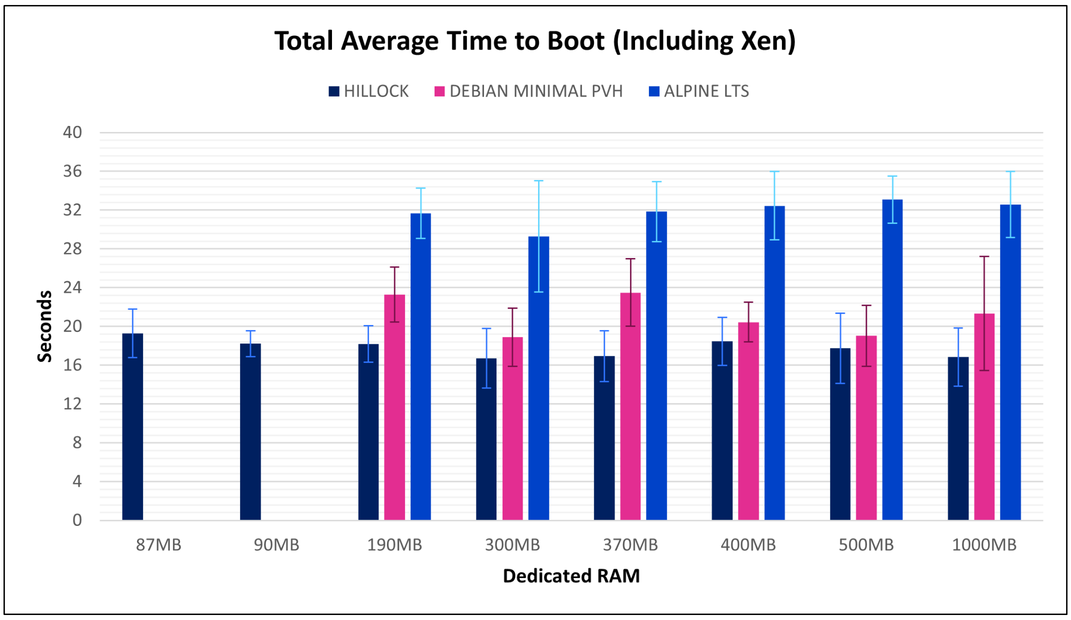


Figure 4.8: Comparing the boot up time of the key distributions

Hillock again takes the lead, being the fastest overall, even faster than the optimised PVH versions developed by the Qubes OS team. Taking into account that Hillock is running with strong isolation and has not been developed by the Qubes OS team, being lighter, less resource-hungry and faster than all the available alternatives; it is an outstanding achievement.

Focusing just on Hillock boot up benchmarks and dividing them by the phases explained in section 3.6.2 and figure 3.3, 4.9 shows the total boot up time averages, including the Xen infrastructure for Hillock:

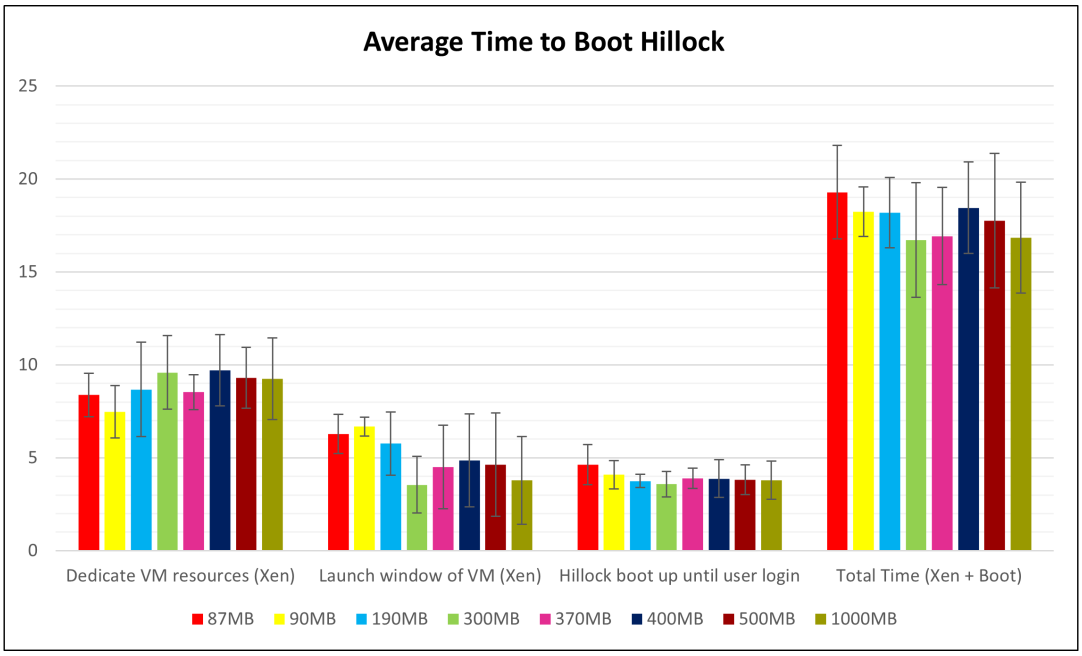


Figure 4.9: Average times of the phases while booting Hillock

As can be seen, the time the Xen Hypervisor takes to allocate the VM resources is independent and approximately constant, regardless of the dedicated VM memory. However, it is clearly different when launching the VM window and starting the machine, where there is a clear decrease tendency. Moreover, Hillock takes a similar time to boot, as seen in the third set of columns.

Finally, figure 4.10 presents all the boot up measurements done for this dissertation per distribution, and per RAM configuration.

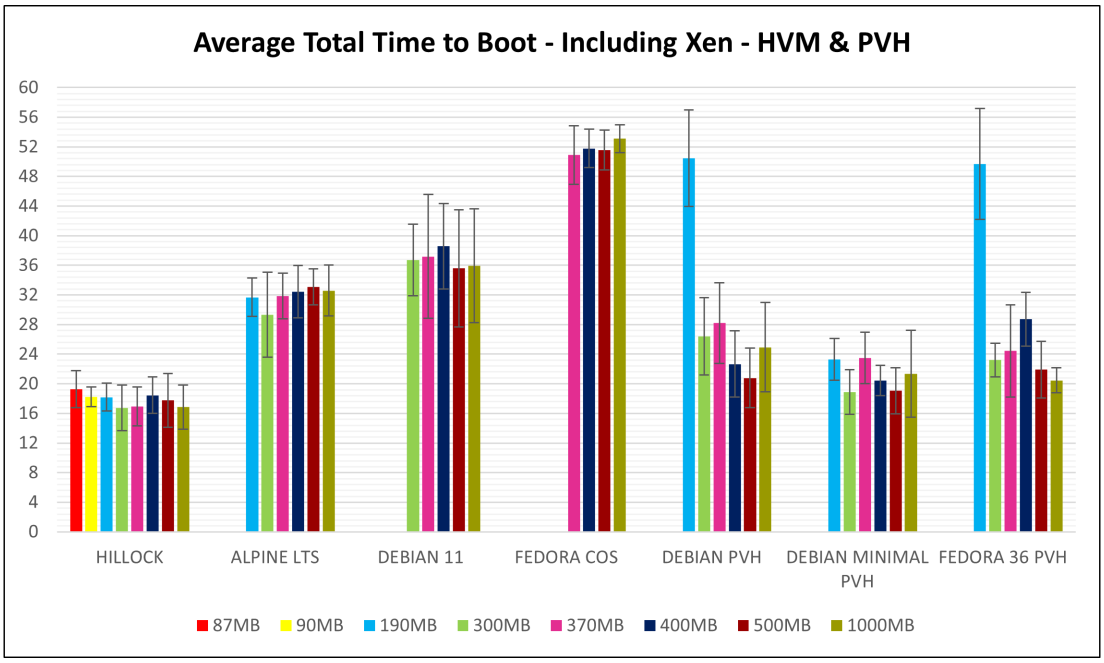


Figure 4.10: Complete - average times to boot while increasing RAM

4.7 Further discussion

The results place Hillock as the best current alternative among the standard options for Qubes OS. However, it should be considered that the other options are running entire systems without being optimised for the Qubes OS environment, especially those running in HVM. The PVH systems might have gone through an adaptation process to be able to communicate with Qubes OS, and only Debian minimal might be close to having a similar kernel customisation approach to the one presented in this dissertation. Despite having a simple tailoring approach through a manually and iterative selected configuration, Hillock still achieved better results.

The main reasons we believe these significant improvements were achieved are:

- No properly adapted and highly specialised systems have been previously developed to compare with. The systems are not optimised as they are designed to run in any environment and hardware. Making them larger and including thousands of lines of code in drivers and features that might be later not used.
- Hillock is highly specialised: removing all possible drivers and features (which do not break the system) and focusing on optimising its functionality for the Qubes OS environment and hardware used in its development.

Chapter 5

Conclusions

In conclusion, a new tailored kernel for the multi-level secure operating system Qubes OS was created during this project. Hillock – the kernel developed – is tailored for this specific situation to improve performance and lightness. However, the kernel can still be used in any other virtualisation system based on x86 architecture, including the cloud. The entire project is released under an open-source license at Hillock's GitHub repository (https://github.com/SergioMiguez/Hillock), where the source code, all kernel configuration files per iteration, benchmarks and results can be found.

A significant problem solved in this project is the bad user experience due to wait times in Qubes OS. Every time the user wants to launch a program in an open VM, it might take a couple of seconds, but if the qube is off and assuming it is only a using PVH, the user might need to wait an average of 26.68 seconds to start launching the program. Qubes OS aims to encapsulate different tasks into different VMs, making the waiting time a common situation for any operation. If the HVM alternatives are also included, the average wait time is 40.15 seconds. The kernel developed – Hillock – allows this wait time to be reduced to 17.80 seconds, decreasing on average 33.28% (or 8.88 seconds) of the wait time of the PVH systems and 55.67% (or 22.35 seconds) of HVM alternatives.

Hillock has been proven to be a great option in the Qubes OS environment. It is faster, lighter and requires fewer resources, beating the rest of the alternatives and meeting by far the initial aim of this research project. Compared to the other OS, Hillock achieved a 22.88% improvement in boot up compared to the fastest alternative and 40.69% if the system uses the same virtualisation technics (HVM). However, the most relevant thing is that these results have been achieved simultaneously by cutting down by a minimum of 54.21% of the required memory resources that must be allocated to allow the boot up. At the same time, Hillock has the smallest compressed and decompressed kernel size, improving by 52.30% and 44.55% to their respective best alternatives.

The kernel developed, despite not being a unikernel, clearly fulfils the general purpose of this dissertation: researching and creating an alternative system for a multi-level secure system which relies heavily on virtualisation and proving that improvement can be made in the area. It also proves that with further research, better-isolated alternatives to containers could be brought to complex environments which depend on efficiency and resources, such as the cloud.

It is intention of the author and the supervisors to summarize the results of this project into a scientific publication to be submitted.

5.1 Main contributions – summary

- Research on kernel customisation and optimisation.
- Properly explain the methodology and requirements to develop a custom kernel for Alpine Linux.
- Develop over 50 kernel iterations to make a final compatible and optimised version for the Qubes OS environment. This last iteration has been presented under the name of Hillock.
- Empirically search the resource limits required to boot up the most common operating systems used in Qubes OS.
- Develop benchmarking scrips for each system and make more than 650 VM boot up tests to acquire enough data.
- Evaluate Hillock's performance compared to its alternatives.
- Develop an automated script to install Hillock in any system.

5.2 Future work

Bringing unikernels to Qubes OS is still a big challenge and something to be done. It might require a team of researchers to adapt one system to this complex environment appropriately.

Conversely, other non-tested alternatives could further explore this project when developing a custom kernel. It would be interesting to create a decompressed kernel version. This methodology could improve the boot up, as no wasted time is dedicated to decompressing the kernel.

Furthermore, other complex methodologies have been researched to reduce the kernel source code through *kprobes* [18] or automated kernel reduction methodologies like [33]. Applying them to Hillock to decrease the kernel further would be interesting.

It would be interesting to do further research to turn containers into VMs or use an optimised and reduced kernel for each container. Hillock's short boot up time (not counting the delay caused by Qubes OS) – 3.81 seconds on average – and the possibility of further optimising this time while having an entire operating system is just a start. This would allow complete isolation in a complex container environment where security is paramount.

This project proves that there is still an improvement margin in the virtualisation sector and that the development of new technology will drastically affect industry and software services. Each program could run in a separate sandbox with an initial 2 or 3 seconds delay to launch and ensure excellent isolation to the point that malware could be drastically reduced.

Bibliography

[1] Charles Anderson. Docker [software engineering]. 32(3):102–c3. Conference Name: IEEE Software.

[2] Eman Al Awadhi, Khaled Salah, and Thomas Martin. Assessing the security of the cloud environment. In *2013 7th IEEE GCC Conference and Exhibition (GCC)*, pages 251–256.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177. Association for Computing Machinery.

[4] Maxime Bélair, Sylvie Laniepce, and Jean-Marc Menaud. Leveraging kernel security mechanisms to improve container security: a survey. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, pages 1–6. Association for Computing Machinery.

[5] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating seccomp filter generation for linux applications. In *Proceedings of the 2021 on Cloud Computing Security Workshop*, CCSW '21, pages 139–151. Association for Computing Machinery.

[6] Timothy Chappelly, Cristina Cifuentes, Padmanabhan Krishnan, and Shlomo Gevay. Machine learning for finding bugs: An initial report. In *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pages 21–26.

[7] Bo Chen, Zhenkun Yang, Li Lei, Kai Cong, and Fei Xie. Automated bug detection and replay for COTS linux kernel modules with concolic execution. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 172–183. ISSN: 1534-5351.

[8] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. 18(1):4:1–4:32.

[9] Jake Edger. A seccomp overview [LWN.net].

[10] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. In Maarten van Steen and

Michi Henning, editors, *Middleware 2006*, Lecture Notes in Computer Science, pages 342–362. Springer.

[11] Mark R. Heckman, Roger R. Schell, and Edwards E. Reed. A multi-level secure file sharing server and its application to a multi-level secure cloud. In *MILCOM 2015 - 2015 IEEE Military Communications Conference*, pages 1224–1229.

[12] Leonard J. LaPadua and David Elliot Bell. Computer security model: Unified exposition and multics interpretation.

[13] Seung Hyub Jeon, Seung-Jun Cha, Ramneek, Yeon Jeong Jeong, Jin Mee Kim, and Sungin Jung. Azalea-unikernel: Unikernel into multi-kernel operating system for manycore systems. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1096–1099. ISSN: 2162-1233.

[14] P.A. Karger. Multi-level security requirements for hypervisors. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 9 pp.–275. ISSN: 1063-9527.

[15] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv - optimizing the operating system for virtual machines. pages 61–72.

[16] Ricardo Koller and Dan Williams. Will serverless end the dominance of linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 169–173. Association for Computing Machinery.

[17] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, pages 1–15. Association for Computing Machinery.

[18] Anil Kurmus, Alessandro Sorniotti, and Rüdiger Kapitza. Attack surface reduction for commodity OS kernels: trimmed garden plants may attract less bugs. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, pages 1–6. Association for Computing Machinery.

[19] Michael Larabel. The linux kernel enters 2020 at 27.8 million lines in git but with less developers for 2019.

[20] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 418–429. Association for Computing Machinery.

[21] Yan Liu. Design and implement a safe method for isolating memory based on xen cloud environment. In *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 804–807. ISSN: 2327-0594.

[22] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft.

Unikernels: library operating systems for the cloud. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 461–472. Association for Computing Machinery.

[23] Anil Madhavapeddy and David J. Scott. Unikernels: the rise of the virtual library operating system. 57(1):61–69.

[24] Mallika Malhotra, Sanya Kapoor, and Prakash Kumar. Dynamic scaling of web services for xen based virtual cloud environment. In *2014 Seventh International Conference on Contemporary Computing (IC3)*, pages 212–217.

[25] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 218–233. Association for Computing Machinery.

[26] Ming Mao and Marty Humphrey. A performance study on the VM startup time in the cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 423–430. ISSN: 2159-6190.

[27] Ilias Mavridis and Helen Karatza. Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. 94:674–696.

[28] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, pages 59–73. Association for Computing Machinery.

[29] James Pope, Francesco Raimondo, Vijay Kumar, Ryan McConville, Rob Piechocki, George Oikonomou, Thomas Pasquier, Bo Luo, Dan Howarth, Ioannis Mavromatis, Pietro Carnelli, Adrian Sanchez-Mompo, Theodoros Spyridopoulos, and Aftab Khan. Container escape detection for edge devices. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, SenSys '21, pages 532–536. Association for Computing Machinery.

[30] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. Unikernels: The next stage of linux's dominance. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 7–13. Association for Computing Machinery.

[31] Marvin Schaefer, Barry Gold, Richard Linde, and John Scheid. Program confinement in KVM/370. In *Proceedings of the 1977 annual conference on - ACM '77*, pages 404–410. ACM Press.

[32] David Scott, Richard Sharp, Thomas Gazagnaire, and Anil Madhavapeddy. Using functional programming within an industrial product group: perspectives and perceptions. 45(9):87–92.

[33] Takuya Shizukuishi and Katsuya Matsubara. An efficient tinification of the linux kernel for minimizing resource consumption. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, pages 1228–1237. Association for Computing Machinery.

[34] Joshua Talbot, Przemek Pikula, Craig Sweetmore, Samuel Rowe, Hanan Hindy, Christos Tachtatzis, Robert Atkinson, and Xavier Bellekens. A security perspective on unikernels. In *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, pages 1–7.

[35] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem. In *Proceedings of the sixth conference on Computer systems*, pages 47–60. ACM.

[36] P.A. Teplyuk, A.G. Yakunin, and E.V. Sharlaev. Study of security flaws in the linux kernel by fuzzing. In *2020 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*, pages 1–5.

[37] Zicheng Wang. Can "micro VM" become the next generation computing platform?: Performance comparison between light weight virtual machine, container, and traditional virtual machine. In *2021 IEEE International Conference on Computer Science, Artificial Intelligence and Electronic Engineering (CSAIEE)*, pages 29–34.

[38] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 199–211. Association for Computing Machinery.

[39] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel.

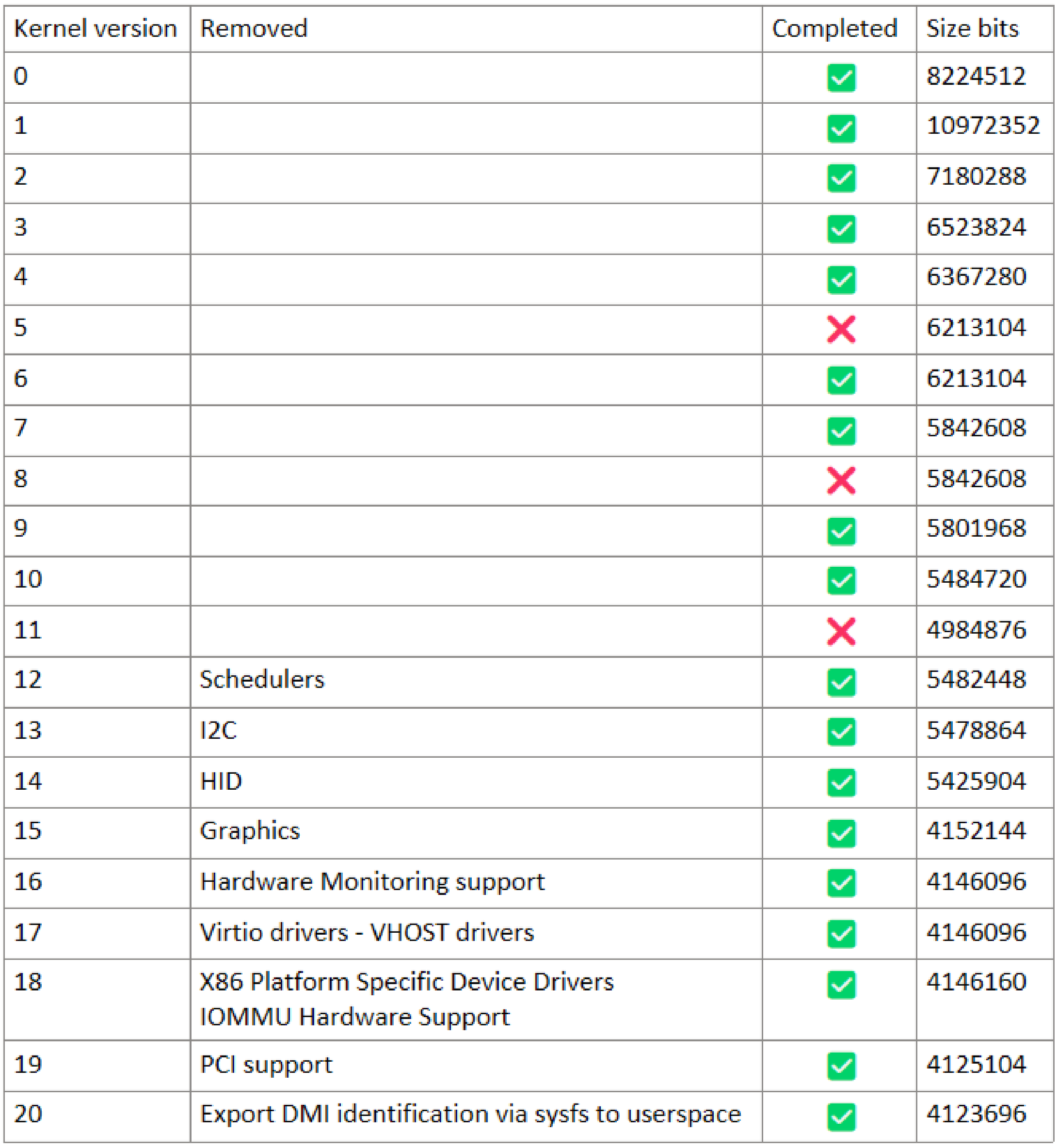
[40] Qiushi Wu and Kangjie Lu. On the feasibility of stealthily introducing vulnerabilities in open-source software via hypocrite commits.

[41] Bruno Xavier, Tiago Ferreto, and Luis Jersak. Time provisioning evaluation of KVM, docker and unikernels in a cloud platform. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 277–280.

[42] Dongyang Zhan, Zhaofeng Yu, Xiangzhan Yu, Hongli Zhang, and Lin Ye. Shrinking the kernel attack surface through static and dynamic syscall limitation. pages 1–1. Conference Name: IEEE Transactions on Services Computing.

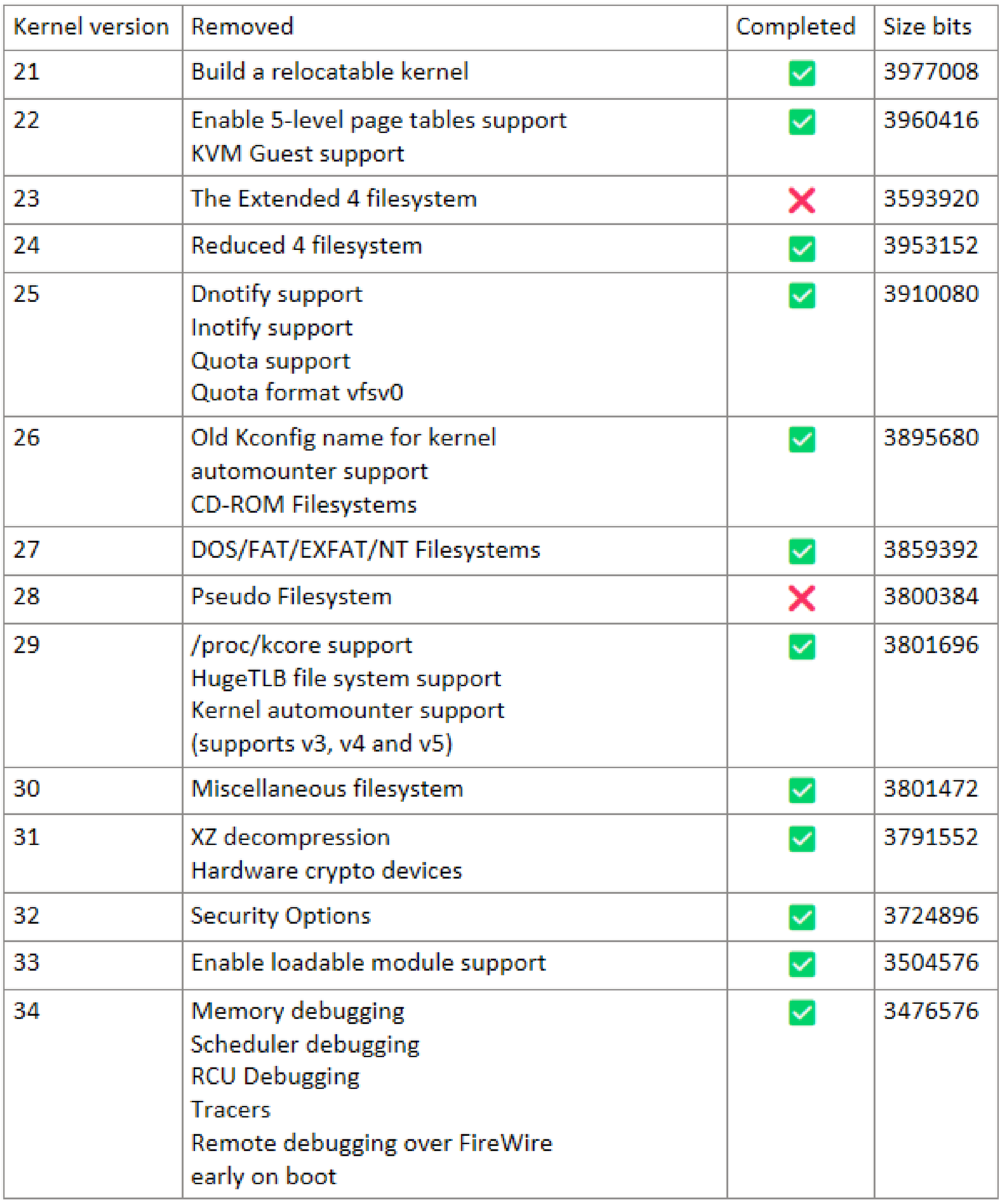
Appendix A

Hillock's iteration table



Kernel version	Removed	Completed	Size bits
0		✓	8224512
1		✓	10972352
2		✓	7180288
3		✓	6523824
4		✓	6367280
5		✗	6213104
6		✓	6213104
7		✓	5842608
8		✗	5842608
9		✓	5801968
10		✓	5484720
11		✗	4984876
12	Schedulers	✓	5482448
13	I2C	✓	5478864
14	HID	✓	5425904
15	Graphics	✓	4152144
16	Hardware Monitoring support	✓	4146096
17	Virtio drivers - VHOST drivers	✓	4146096
18	X86 Platform Specific Device Drivers IOMMU Hardware Support	✓	4146160
19	PCI support	✓	4125104
20	Export DMI identification via sysfs to userspace	✓	4123696

Figure A.1



Kernel version	Removed	Completed	Size bits
21	Build a relocatable kernel	✓	3977008
22	Enable 5-level page tables support KVM Guest support	✓	3960416
23	The Extended 4 filesystem	✗	3593920
24	Reduced 4 filesystem	✓	3953152
25	Dnotify support Inotify support Quota support Quota format vfsv0	✓	3910080
26	Old Kconfig name for kernel automounter support CD-ROM Filesystems	✓	3895680
27	DOS/FAT/EXFAT/NT Filesystems	✓	3859392
28	Pseudo Filesystem	✗	3800384
29	/proc/kcore support HugeTLB file system support Kernel automounter support (supports v3, v4 and v5)	✓	3801696
30	Miscellaneous filesystem	✓	3801472
31	XZ decompression Hardware crypto devices	✓	3791552
32	Security Options	✓	3724896
33	Enable loadable module support	✓	3504576
34	Memory debugging Scheduler debugging RCU Debugging Tracers Remote debugging over FireWire early on boot	✓	3476576

Figure A.2

Kernel version	Removed	Completed	Size bits
35	Include all symbols and kallsyms Kernel->user space relay support (formerly relayfs)	✓	3379616
36	Adding network support again	✓	4916512
37	Freezer Controller (check if using cgroup2) Include legacy /proc/pid/cpuset file	✓	3377280
38	Printk and dmesg options Kernel Testing and Coverage	✓	3375232
39	Kernel support for ELF binaries Write ELF core dumps with partial Segments	✗	3368896
40	Write ELF core dumps with partial Segments	✓	3375232
41	Support memoryless force-feedback devices	✓	3372096
42	Sparse keymap support library	✓	3371712
43	Event interface	✓	3365760
44	AT Keyboard	✗	3358560
45	Serial port line discipline	✓	3365696
46	Initial RAM filesystem and RAM disk initramfs/initrd Support Initial Ramdisk/ramfs compressed using gzip	✓	3353312
47	X86 Debugging	✓	3348016
48	Kernel debugging	✓	3349168
49	Adding 17, 18	✓	3379184
50	Final version - 47 Recompiled	✓	3347984

Figure A.3

Appendix B

Repository architecture

This section presents and describes the organisation and files from Hillock's repository - **https://github.com/SergioMiguez/Hillock**.

- Custom Kernel Development – Folder – Contains all the files related to the Hillock kernel development
  - Kernel Compiled Size Reports – Folder – Contains all the information for the kernel file sizes.
  - Kernel Iterations – Folder – Includes all the development configuration files for each kernel iteration created. The files can be moved to a folder with the Linux kernel source code and be compiled, installing any of the versions developed and available from this project.
  - Testing Performance – Folder – Contains the scripts to output the uptime value of any VM and a Results folder with all the raw data and measurements. The key file in the folder is "TestData.xlsx", which includes all the graphs used in the dissertation and measurements done.
  - .config_v50_final – Linux kernel configuration file – The final iteration developed in this project, also named Hillock. The file can be moved to a folder with the Linux kernel source code and compiled, installing Hillock in the machine.
  - .defconfig_x86 – Linux kernel configuration file – The initial basic configuration file. This file is just used as a reference and is automatically generated using the correct "make" commands.
  - README.md – a README file which explains step-by-step how to install the Hillock kernel.
  - kernel_install.sh – Script – Automates all the Hillock kernel download, installation and clean-up to be used anywhere.
- dom0_managin_scripts – Folder – Contains the automated scripts to create, clone and remove VMs in Qubes OS

- update_repo.sh – Script – Used to automatically move the development files from dom0 (without network access) to another working VM with the GitHub repository to be committed and pushed.