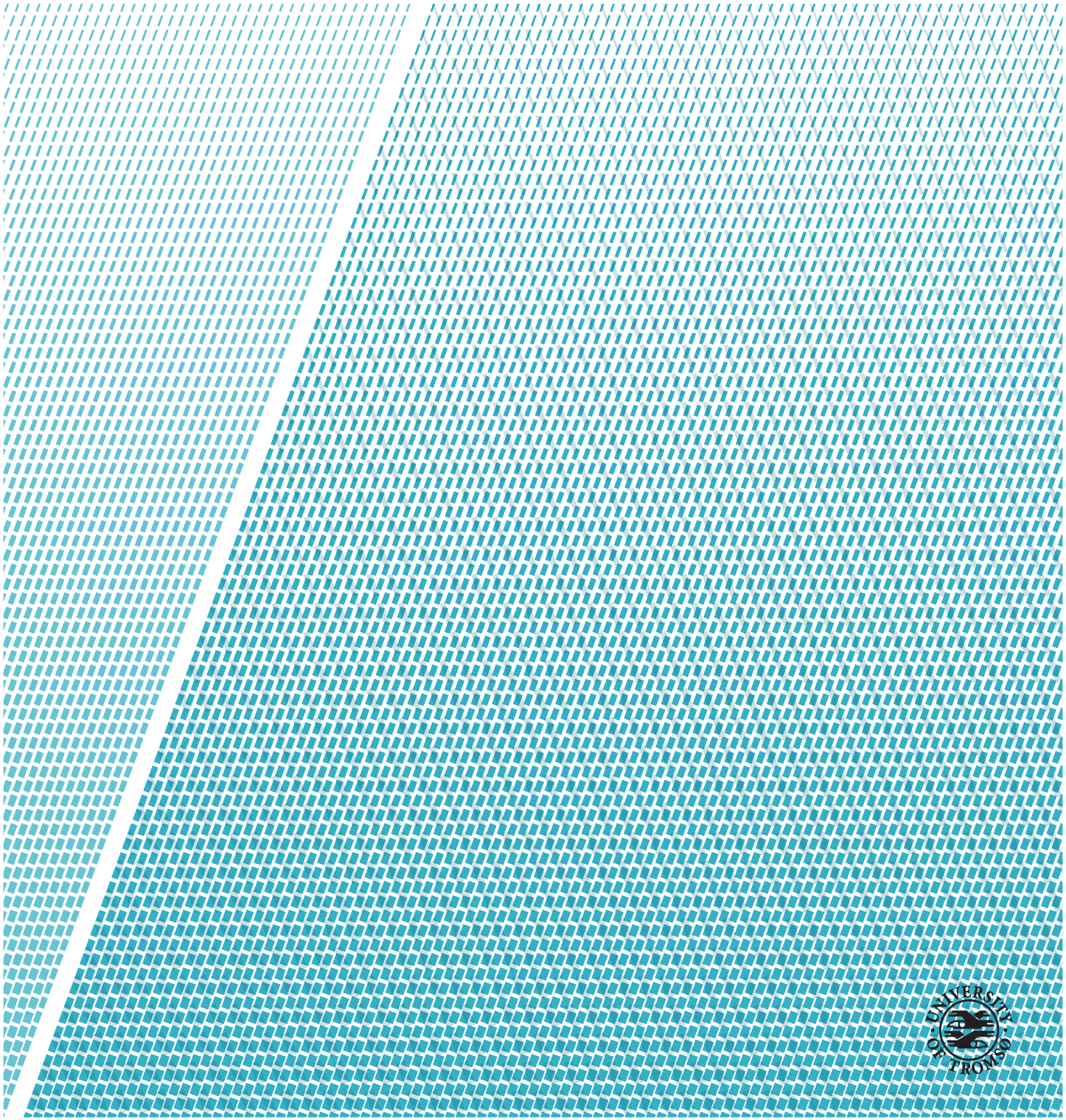
UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

Latency Optimized Microservice Architecture for Privacy Preserving Computing

Nikolai Åsen Magnussen

INF-3981: Master's Thesis in Computer Science June 2019



"Publication is the self-invasion of privacy."
–Marshall McLuhan

"Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law."
–Douglas Hofstadter

"If debugging is the process of removing bugs, then programming must be the process of putting them in."
–Edsger W. Dijkstra

Abstract

Recent developments in microservices architecture and building have lead to the advent of unikernels, a specialized operating system kernel coupled with, and executing only, a single application.

This thesis presents PPCE a distributed system utilising a microservices architecture based on unikernels, created to enable privacy-preserving computing for users, classes of users, and more importantly; doing so with acceptable latency.

We built an initial version of PPCE, and through experimentation, we measured the latency to be 63.15 ms. Then we devised two optimized approaches to reduce the user-perceived latency of the initial version.

After conducting experiments on the two optimized versions of PPCE, we discussed the applicability of the two approaches and compared them to related work. Showing that our hybrid version of PPCE is able to implement an architecture enabling privacy preserving computing with only 29.53 ms user-perceived latency.

Acknowledgements

To my supervisor Håvard D. Johansen; for his criticism, advice and guidance. Thank you for allowing me to bear the burden of responsibility and freedom, and always being available for discussions.

To all the people of the Corpore Sano lab; they have heaps of patience. We have had numerous interesting, and less interesting, discussions. You truly are an amazing bunch of people, all of which will be missed!

To my fellow master's candidates in the class of 2014; congratulations and best of luck. Almost five years ago we didn't know any programming, but look at us now. Thank you for the five amazing years, to spending even more of them together with you! I love you all!

To my friend Andreas Isnes Nilsen; for his love, hate and support. Thank you for not taking me to Dignitas, but enduring my crazy ideas and behavior, and at times encouraging it with your own craziness. Without you, this journey would not have been close to this exciting and enjoyable.

To the rest of my friends; you know who you are. Your help and presence was greatly appreciated.

List of Abbreviations

AOT Ahead of Time

API Application Programming Interface

ARP Address Resolution Protocol

BLP Bell and LaPadula

BTC Bitcoin

DDR Double Data Rate

DNS Domain Name System

GC Garbage Collector

HTTP Hypertext Transfer Protocol

HVT Hardware Virtualization Tender

IAD Information Access Disruption

IP Internet Protocol

JIT Just in Time

JSON JavaScript Object Notation

KVM Kernel-based Virtual Machine

LB Load Balancer

LXC Linux Containers

MAC Media Access Control

ML Meta Language

NVMe Non-Volatile Memory Express

OS Operating System

OSI Open Systems Interconnection

POSIX Portable Operating System Interface

PPCE Privacy-Preserving Computing Environment

PRIVATON Privacy Automaton

RAM Random Access Memory

REST Representational State Transfer

SGX Software Guard Extensions

SML Standard ML

SSD Solid State Drive

TCB Trusted Computing Base

TCP Transmission Control Protocol

TLS Transport Layer Security

UiT University of Tromsø

UUID Universally Unique Identifier

VM Virtual Machine

VMM Virtual Machine Monitor

VMMD Virtual Machine Manager Daemon

Contents

List of Figures

List of Tables

1 Introduction

The move from monolithic computing systems to microservices oriented system architectures, where each component only performs a few or a single part of the computation, has sparked new development and interest in different hosting architectures and methods. One of these new methods for hosting microservices are *unikernels*, where a library operating system in combination with the microservice application code create a small operating system only capable of performing the tasks defined in the application code. This leads to microservices with small trusted computing bases compared to conventional operating systems.

This thesis presents PPCE, a unikernel based privacy preserving computing environment. The concept privacy preserving in the context of this thesis is meant to encompass the system to preserve the privacy of its users by both access restriction and reporting of user behavior and data access. The system will be evaluated and optimized to minimise user-perceived end-to-end latency while still being scalable and practical in use.

1.1 Thesis Statement

This thesis will investigate the applicability of using the MirageOS [1] library Operating System (OS) to build a microservice architecture for privacy preserving computing. In addition, we will investigate the feasibility of using latency

hiding to optimize user-perceived end-to-end latency.

Our thesis statement is:

> *MirageOS can be used to build a microservice architecture for privacy preserving computing while providing reasonable user-perceived end-to-end latency.*

1.2 Context

The context of this thesis is in the Corpore Sano Centre[1] at University of Tromsø (UiT) with the Information Access Disruption (IAD) research group. The group focus areas are in privacy, operating systems, and employing information to empower users. All the while not violating privacy of users, and keeping systems secure.

Our work with OS architecture include Vortex [2], an implementation of the omni-kernel operating system architecture providing fine-grained scheduler control of resource allocation and pervasive monitoring. We have also developed Fireflies [3], a Byzantine fault-tolerant membership and gossip service capable of operating with the presence of malicious members. Using Fireflies, we built FirePatch [4], a secure and time-critical software patch dissemination system with the goal of preventing an adversary from delaying patches in a distributed system.

With the introduction of Intel Software Guard Extensions (SGX), we conducted extensive research and performance evaluation, leading to recommendations for application development on the architecture [5]. In the domain of privacy, we show a mechanism for enforcing privacy policies for shared data [6]. The combination of previous work, together with our partners at Pomona College and Cornell University, resulted in SGX enforcement of use-based privacy [7].

1.3 Methodology

The ACM task force on the Core of Computer Science presents a scientific framework for describing computer science and computing engineering. It defines computer science and computer engineering through three paradigms, forming the basis of scientific work within computing:

1. https://corporesano.no

Theory is roted in mathematics, where a valid theory is described through defining the objects, hypothesize possible relationships between them, proving said relationships and finally interpreting the results

Abstraction is rooted in the experimental scientific method, focusing on the investigation of phenomena. Its method involves designing experiments from collected data and performing analysis based on an initial hypothesis

Design is rooted in engineering, applicable to the construction of a system or device. These steps involve stating requirements and specifications of the system, designing and implementing it, and finally testing it.

This thesis is rooted in the areas of systems engineering and the scientific method, consequently using methods from both the design and abstraction paradigms. We will specify our requirements regarding end-users and system features. PPCE will be implemented through an iterative process, taking into account the original requirements and from experimentation and results, we will change the system to better suit end users, demonstrating the system's capabilities and applicability.

1.4 Outline

The rest of the thesis is outlined as follows:

Chapter 2 introduces required background information, and other prerequisites pertaining to system components, theoretical concepts such as information flow and the lattice model, the high-level concept of unikernels, and the more specialized MirageOS which is employed in this thesis.

Chapter 3 describes the overall system architecture with design choices and requirements, and implementation details of the components composing PPCE.

Chapter 4 first describes the experimental setup, then goes into detailed description of the experiments conducted, with the results from the different experiments, and discussing the experimental results.

Chapter 5 discusses PPCE in terms of security and privacy, service lifetimes and its shortcomings.

Chapter 6 concludes the thesis, compares PPCE against other related work,

and outlines future work.

2 Background

2.1 Information Flow

In the field of information theory, information flow is a well researched subject, with Denning's seminal paper from 1976 [8] introducing the lattice model for secure information flow. Information flow focuses on the flow of information between objects through operations [9], rather than the transfer of access and access to larger objects such as processes or files, such as the original Bell and LaPadula (BLP) model [10].

The lattice model is a mathematical framework for formulating requirements for secure information flow between security classes. As the name suggests, the central component is a lattice structure, which is derived from a set of security classes. Secure information flow is defined to be the property that no unauthorized flow of information is possible [8]. In practice, information flow policies are not necessarily safety properties [11].

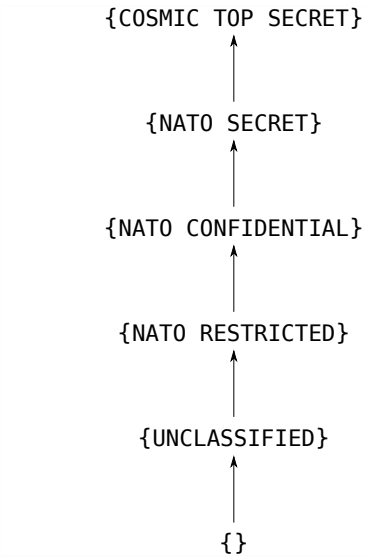


Figure 2.1: Linear lattice based on mutually exclusive security classes

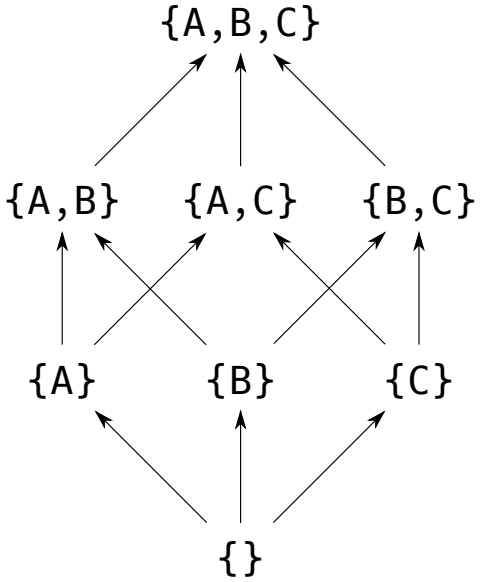


Figure 2.2: Subset lattice based on a subset of the security classes {A, B, C}

The lattice structure is derived from a set of security classes, and either be a linearly ordered lattice, or a lattice of subsets. As Figure 2.1 illustrates; the lattice derived from the security classes {UNCLASSIFIED, NATO RESTRICTED, NATO CONFIDENTIAL, NATO SECRET, COSMIC TOP SECRET} is linear. On the other hand, Figure 2.2 illustrates the more complex lattice derived from the security classes {A, B, C}. It is also possible to construct even more complex lattices, with the only constraint being that the constructed structure is a *universally bounded lattice* [8].

More formally, the following notation is defined:

SC is the set of disjoint security classes of information.
$\oplus$ is the class-combining operator specifying the output security class when any binary operator is applied to a pair of operands.
$\longrightarrow$ is the flow relation on pairs of security classes.

We write $A \longrightarrow B$ if and only if information from class A is permitted to flow into class B. Using the security classes from Figure 2.1, we can write NATO CONFIDENTIAL $\longrightarrow$ COSMIC TOP SECRET, but not NATO RESTRICTED $\longrightarrow$ UNCLASSIFIED.

In the lattice model; information, users, and objects are assigned security

classes. A flow model is considered secure if and only if execution of a sequence of operations cannot result in a flow that violates the relation defined by the operation '⟶'.

2.2 Use-Based Privacy

The concept of use-based privacy is a part of the field of information theory. Use-based privacy is fundamentally different from the views originating from the seminal paper on privacy from Brandeis and Warren [12] which resulted in privacy often being defined in terms of access control. Comparatively, use-based privacy is not just concerned with keeping information secret, but also restricting how information is used [13].

To enforce fine-grained user privacy based off use-based privacy, Birrell propose avenance tags which use a new language for expressing privacy policies [14]. Avenance tags are enforced by employing a Privacy Automaton (PRIVATON), a finite state automata which are used in conjunction with the rest of the avenance ecosystem to enforce user privacy. Use-based privacy can either assume malicious adversaries, or accountable ones while still guaranteeing privacy when using SGX [7].

2.3 Microservices

As more companies are adopting cloud computing to deploy applications, the microservices [15] architecture is gaining in popularity [16]. Contrary to common beliefs,the ideas of microservices architecture are not entirely new, but is based on concepts from service-oriented architectures, and is also called the second generation of services [17].

Systems with a monolithic architecture results in tight coupling between system components, making them more difficult to, among other things, maintain and scale [18]. Modern applications have taken the route of splitting these large, monolithic systems into microservices [19]. A single microservice will not provide the same functionality as a monolithic system. Each microservice will provide a limited set of functionality, which combined with the other microservices provide the same functionality as the original monolithic system. Not only does this make it easier to swap out individual microservices and updating them. But microservices are also generally designed so that they contain little to no state, resulting in them being easier to scale horizontally by spawning multiple instances as the total load increase. This is a common development

when running services on a cloud-based infrastructure [20].

Microservices are often deployed in the cloud, using containers such as Docker [21] or Linux Containers (LXC) [22], or Virtual Machines (VMs) on a hypervisor such as Kernel-based Virtual Machine (KVM) [23] or Xen [24].

2.4 Containers

The idea of virtualization and VMs has its origins in 1966 from IBM and CP-40/CMS and described in their seminal paper [25]. Virtual machines are executed on top of a hypervisor using a Virtual Machine Monitor (VMM). The hypervisor provides hardware virtualization, meaning it provides the VMs with emulated CPU, memory, I/O and other devices.

Containers also employ the idea of virtualization, but uses OS virtualization rather than hardware virtualization [26]. The container equivalent of a hypervisor is a container engine which is a part of the host OS, allowing it to provide the containers with OS virtualization. In practice, OS virtualization means a container will use the services of the host OS kernel, and provide the required libraries for the applications it hosts.

There are different implementations of containers. Perhaps the most popular ones for Linux are LXC, and Docker which is a further development based off LXC. An important result of containers only providing a small base and the application itself, is that they are capable of being spawned quickly compared to a VM, while maintaining low overhead in terms of performance and footprint. In some instances, the throughput is higher, and latency lower for an application in a container compared to a VM [27]. Containers are protected from each other, somewhat like VMs, but resulting in a lower level of security than VMs due to multiple containers sharing several different resources they get from the host OS kernel [28].

2.5 Unikernels

The idea of the library OS, and application-level resource management is not a new one [29], [30] and the label unikernel is mostly a refinement and a new label for library OSs built for executing on a hypervisor, only supporting the virtual devices provided [31].

Unikernels are a further development after the initial rise of VMs followed by the

recent emergence and adoption of container technology. Virtual machines only share the hardware and hypervisor, and usually, the same VMM. Comparatively, containers share the same as VMs, but also the kernel of the VM. Unikernels often share the same resources as any other VM, but further research has shown that it is feasible, and a good decision security-wise to compile a specialized VMM together with the unikernel, leaving them to only share hardware and hypervisor [32], [33].

The main difference between regular VMs and unikernels is that VMs usually run a commodity OS, like Ubuntu Linux. In addition to the one service you want to run, commodity OSs also usually contain other services, such as a SSH server and client, web browser and file manager. They are components of the system that are not required, and will make the system easier to compromise, and to further pivot from a compromised system to other systems, due to the increased attack surface further.

Unikernels are statically compiled from the application code and the library OS. Optionally, a specialized VMM is also compiled. The compilation process contains, among other forms of optimization, dead code elimination. Meaning the compiled unikernel will only contain the drivers and capabilities that are used by the application code. As a consequence; if the application is not using disk for storage but only operates in memory, like Redis, the resulting unikernel does not have the ability to access disk devices, reducing the attack surface.

There are a multitude of different unikernel systems, or library OSs, such as IncludeOS [34] written in C++, EsseOS [35] written in Haskell, ClickOS [36] optimized for network functionality, Drawbridge [37] being a Windows 7 library OS, and MirageOS [38] written in OCaml.

2.6 OCaml

OCaml is a functional programming language that allow multi-paradigm code, such as imperative and object oriented programming styles as well [39]. OCaml uses a strong static type system to enforce types, as well as having algebraic data types to ensure further reliability and safety. Algebraic data types are a part of the functional programming paradigm, which has its roots in Church's lambda calculus [40] with a foundation in mathematics. OCaml itself stem from the much older Meta Language (ML), which was standardized by Milner with Standard ML (SML) [41].

OCaml is a language with automatic memory management, meaning the lan-

guage runtime will allocate and free the memory for the user. OCaml employs an incremental Garbage Collector (GC) [42], meaning it is better suited for soft real-time applications than languages with other types of GC [43]. Studies also show that OCaml code yields performance on par or better than Java, and close to that of C and C++ [43].

This results in OCaml being well suited for systems programming, soft real-time applications and other applications requiring high performance, safety and reliably. These qualities make OCaml very attractive compared to other languages generally used for systems programming, such as OSs [44].

2.7 MirageOS

There are multiple library OSs that are capable of creating unikernels, and they have different goals and are programmed using different languages. MirageOS is a library OS for building unikernels using the OCaml programming language with the goal of being a clean-slate approach to the unikernel space [38].

MirageOS is a clean-slate approach as a library OS, meaning that it ignores Portable Operating System Interface (POSIX) compliance, and by that backwards compatibility with existing software. Which means that existing software already written in OCaml must be modified to conform to the interface exposed by the MirageOS libraries, and software written in other languages must be rewritten in OCaml while conforming to the MirageOS libraries' interface.

MirageOS empowers developers to use a three-stage development process, making development and potential debugging easier [38]. This is done by providing multiple compile targets where, in terms of the three-stage development process, the two first targets are common and the final stage will be specialized. The first stage is to compile the unikernel as a Unix application making use of system calls as other applications. Compiling the unikernel as a Unix application with its own direct networking stack using a virtual ethernet device, is the next, and second, stage. Finally, the unikernel is compiled against the specific environment that it will be deployed in, such as a Xen image, or Hardware Virtualization Tender (HVT) for KVM. This three-stage approach enables developers to build the three targets from the same source code, optimally, without needing to make any changes to the source code between testing, using a Unix target, and the deployment on the release platform.

3

Design and implementation of PPCE

In this chapter we give a system overview of PPCE. We describe our design decisions, requirements and scope limitations. In addition, we will describe the implementation of the system components in isolation, and unifying them to compose PPCE.

3.1 System overview

The design goals of the system is for a user, or a class of users, to use the system in a manner that transparently preserves privacy.

More specifically, PPCE should have the following properties:

- Provide users with access transparency.
- Preserve user privacy through isolating computations.
- Minimize resource sharing through hardware virtualization.

From a consumer's point of view, PPCE should act like a single, monolithic

system handling all requests and performing all computation for all users and all classes. The above mentioned system quality is access transparency, meaning a user will always access resources in the same, uniform way in a distributed system [45]. While the users see this, the system will really preserve privacy and security by only allowing a single user, or a class of user to use an application instance. Meaning the system must create and instantiate multiple instances of application services, which will perform the required computation. The goal of this is to eliminate a whole class of bugs that can be leveraged to create information leakage. An example of such a bug is the one leading to the Heartbleed vulnerability in the OpenSSL library [46], which allowed a user to leak information from the system, possibly obtaining information the user is not authorized to view, compute on and have.

The threat model is a user who is authenticated, but will not deliberately perform malicious acts due to accountability. This scenario is very similar to the attacker scenario at hospitals where doctors technically are able to view journals of patients they should not view, but due to the oversight and the consequences of actually accessing the journals without good reason, and even further spreading them, will deter doctors form acting maliciously.

PPCE consists of multiple components, each of which are currently trusted, as they are all part of the core system. In practice, all components act as their own separate systems using the services of other components, and trusting their output. Due to difficulty of debugging and time constraints, Transport Layer Security (TLS) certificates have not been issued and used internally within the system.

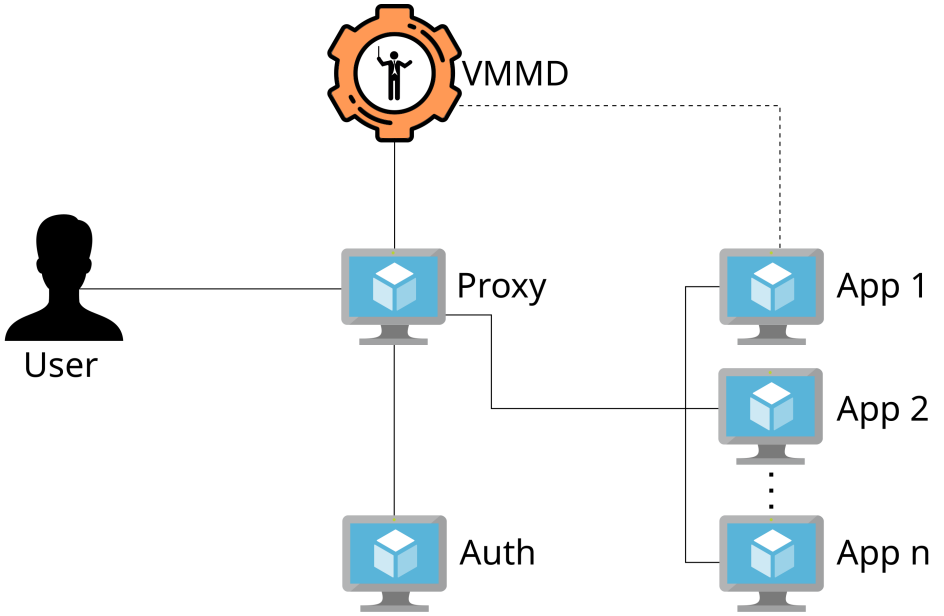


Figure 3.1: Architecture of PPCE with communication lines between the different components

Figure 3.1 shows that the PPCE consists of four distinct parts; a combined proxy, Domain Name System (DNS) server and Load Balancer (LB), an authentication service, a Virtual Machine Manager Daemon (VMMD), and multiple application services.

The proxy service is the point of entry to PPCE and the only component the user should interact with, and the rest should not be accessible through the internet, but only via the internal network the proxy is connected to. Resulting in the user seeing the proxy service as the whole monolithic system, and not any of the other microservices. The user will send a request to the proxy service, which will make sure the user is authenticated and authorized, before handing over the request to the appropriate computation service.

As shown in Figure 3.1, there are not much communication going on between the different components. Most communication is between the proxy and the other microservices, resulting in the proxy acting as a hub for communication and the system as a whole.

3.2 Communication Between Components

We will now describe the communication between the different components of the system as we can see in Figure 3.1 on the previous page. One thing to note is that these lines of communication are not enforced as the only lines of communication, but only as the default lines of communication in the system and between components.

When a user makes a request to the system, the proxy initially receives the request, and is responsible for routing it to the intended destination. But before the request can be routed to its destination, the user needs to be authenticated, though it is possible to allow a user to not be authenticated before communicating with the system, possibly allowing the user access to very general and otherwise fully open information. In most cases, and as the system currently functions, the user must be authenticated, which is not something the proxy is able to do on its own, and it relies on the auth service, sending the user credentials to the authentication service, which performs its internal authentication routine, and returns to the proxy which class of user the authenticated user belongs to, if any.

Based on the user class, the user might be outright denied access if the credentials are wrong, or the request may proceed to the next check. Once a user is authenticated and the proxy knows which class the user belongs to, the proxy will find a suitable application service to forward the request to. The request is then forwarded to the appropriate application, which by its rules performs computations, potentially querying a database or performing some other arbitrary action with which the proxy is not concerned. After the application service is finished with its computation, the response will be returned to the proxy, which will forward the response back to the user.

The communication pattern described above only applies when there is a suitable application service available for the particular user class, and for the particular computation needs of the user. If such an application is not currently present, the proxy need to make sure there is one available, and queries the VMMD to spawn a new application that is suitable for that user's clearance level and computation needs. If the VMMD has an appropriate application registered, it will use it's internal logic to create and start a new application service, and respond with the necessary routing information. The proxy then uses the information from VMMD to register the application service. After which the proxy will try and forward the request to the application service, retrying until the application is ready to receive network traffic. The subsequent communication will at this point be the same as the communication between the proxy and application described above.

3.3 Proxy

As mentioned in section 3.1 on page 11, the proxy service is not only a proxy, but also a LB and has some of the characteristics of a DNS server. A LB is a front facing system which receives requests and distributes them to the different available instances of a service. A LB balances the load according to some scheme, which may be segmenting users geographically, yielding lower latency for users due to geographical locality [47], or in a round-robin fashion - evenly spreading the requests over multiple instances [48]. Some LBs require the services to be configured at startup, or at compile-time, while a dynamic LB allows services to register during runtime.

A proxy is a service with the ability to forward a request based on a pattern, to another service generally not available externally, but only through the proxy server. Proxy servers serve some of the same functionality as a LB, such as the ability to proxy connections to different other applications. A proxy does not need to perform load balancing in addition to proxying, and a proxy usually proxies based on some more complex routing scheme than a LB that can spread to the actual application behind the LB. These are sometimes combined into a proxy server that can load balance. One example of this is the Nginx web server which is a web server, but also serves as a LB and proxy if configured to do so.

The proxy server implemented here acts as a proxy server, matching routes to proxied services, as well as load balancing services by providing multiple instances of the same service. By allowing services to register themselves with the proxy, it provides dynamic load balancing. A great difference between this proxy and other proxy services is the fact that this proxy is closely coupled with the authentication service, which provides the clearance level for a user, impacting which instances that user is allowed to access.

Using the linear lattice show in Figure 2.1 on page 6, a user with `NATO SECRET` clearance is not allowed to use a service graded `COSMIC TOP SECRET`. If the user with `NATO SECRET` clearance is allowed to use a service graded `NATO CONFIDENTIAL`, that service's grading must be changed to `NATO SECRET` denying users with only `NATO CONFIDENTIAL` clearance to gain access to the service, because of information potentially leaked from the request by the user with clearance `NATO SECRET`. This is required to maintain secure information flow. The most straight forward way to maintain the security of the information flow is to only allow a user to interact with a service with the exact same clearance level as his or her own.

The proxy can receive a registering request from a service where it specifies the required clearance level for access and with the Internet Protocol (IP) address of

itself. Registering will place the required information into a mapping structure between the tuple (*application*, *clearance*) and a set of IP addresses serving that particular combination of (*application*, *clearance*).

Upon receiving a request from a user, the proxy will extract the username and password from the request, and send that to the authentication service, which responds with a Universally Unique Identifier (UUID) which is used to identify the user from that point forward. Together with the UUID, the authentication service also returns the user's clearance level, which will be used to determine which, if any, service it is allowed to access.

If an appropriate service is already registered, the request is forwarded to it, and later returned back to the user. If on the other hand, no appropriate service is registered, the proxy will determine the name of the service, and request the VMMD to spawn a new instance with that specific clearance level. If successful, the VMMD will respond OK with the clearance and IP address of the newly spawned unikernel, which the proxy uses to register the application in its internal routing tables. This enables the proxy to subsequently forward user's requests to it.

3.4 Authentication

In order for an authentication service to authenticate a user, it must have access to either a token store, username and hashed passwords, or both. Most authentication systems still require users to log in by providing a username and password. The most straightforward, but inherently problematic from a security point of view, is to store the username and passwords in plain text, and compare the provided credentials with the stored ones. In earlier systems, it was commonplace to store plain text passwords for a long time, but the practice has almost disappeared, partly due to the large number of major data breaches reported during the last decade [49]. In spite of this, we still observe issues relating to storing passwords in plain text [50], [51], and systems designed with the assumption that their systems or networks will never be breached. This issue grows even larger when users reuse their passwords at multiple sites, and the compromise of one service lead to accounts on other sites being compromised too [49], [52].

To properly secure passwords, it is recommended that a sufficiently hard-to-compute algorithm is used, as well as salting the password when hashing it. An easy-to-compute hash can either be reversed due to mathematical vulnerabilities being discovered, or due to the ability to brute force, or rainbow table them very quickly [53]. By rainbow table, we mean a precomputed mapping

from hash to password, using storage to reduce the required computation [54]. A salt is a random value combined with the password, and is stored together with the username and hashed password. Passwords should be salted when they are hashed to make sure that two equal passwords are not hashed to the same value. Meaning that salt ensures that two equal passwords with different salt yields two different hashes, making brute forcing or rainbow tables much more time consuming, and on some occasions; infeasible.

The authentication service of this system uses salt from a high quality randomness source, along the password hash to verify a user's username and password combination. When the authentication service receives the username and password from the proxy, it will pass them along to the authentication service for validation. Upon receiving the username and password, the authentication service fetches the user with that particular username, and uses the Argon2 algorithm [55] for verifying the password's validity against the stored digest together with the stored salt.

If the password is verified, the authentication service returns a UUID which will be the user's cookie for authentication during the rest of the session. In addition to the UUID for identifying the user's session, the user's clearances which is stored with the credentials is also returned so the proxy service can determine the clearances of the newly logged in user. Meaning the proxy service is able to use that information to determine which application service the user should be proxied to.

3.5 Virtual Machine Manager Daemon

In our PPCE, the VMMD orchestrates application services, allowing them to be started on demand, and managing them while they are running. If the proxy does not have an application registered for the combination of application service and clearances, it must request the VMMD to spawn the appropriate application service at the required level.

The VMMD orchestrates the different application services by starting new instances, stopping running instances, registering new ones, de-registering registered ones, and listing the current status of the different services available for starting as well as currently running. The most important aspect of such an orchestrator is to manage the lifetime of the services it launches, both being able to start and stop, in addition to monitoring [56].

One can draw parallels between the VMMD and both Virt-Manager from Red Hat [57] for regular VMs, or Docker swarm mode [58] and Kubernetes [59]

for Linux containers. The VMMD is closer to Virt-Manager in the way that it orchestrates VMs and not containers, but more similar to Kubernetes or docker swarm mode in the way it orchestrates microservices while being a microservice itself. In particular, it differs from the others by being more light-weight and specialized, in addition to being implemented in OCaml, just as the other components of PPCE.

The VMMD is implemented as a Unix daemon with a Representational State Transfer (REST) interface for actions such as registering the unikernel service, deregistering a unikernel, starting a new unikernel, killing a unikernel, as well as listing available unikernels. The VMMD is not implemented as a unikernel itself, nor can it be implemented as one. It needs to be executed as a regular application on the host OS to use the OS capabilities required to create and run VMs using the KVM subsystem of the Linux kernel.

Registering a new unikernel requires the requester to provide the path to the HVT image of the unikernel along with a descriptive name, and the default clearance level. When a unikernel is registered, the information is stored in memory to be used by the VMMD for subsequent requests.

When the proxy requests a new service to be started, it will provide the name of the application service to launch and the optional required clearances. The clearances are optional, and if they are not provided, the VMMD will use the default stored clearances. To spawn the new unikernel, two things that are not in the host OS by default are required: the HVT VMM, and a tap device. The HVT application, which is a specialized VMM is required, and will be located by the HVT image which will be booted. It is for all practical purposes a small and highly specialized version of QEMU. Because the unikernel does not use the host Transmission Control Protocol (TCP)/IP stack, it needs ethernet frames, which it can get through the tuntap network subsystem in the Linux kernel. Tap interfaces are virtual devices located at layer 2 of the Open Systems Interconnection (OSI) model, yielding ethernet frames. They must first be created and a route must be added, or it can be added to a bridge device in Linux, which acts as a virtual switch, routing ethernet frames and performing all other actions that a physical switch would, such as containing the Address Resolution Protocol (ARP) table.

When the VMMD receives a request to start a new unikernel service, it must first create a new tap device and add it to the appropriate bridge to ensure proper routing. After which the HVT VMM is launched, attaching the tap interface and starting the application service unikernel.

Upon creation, the VMMD stores a UUID of the started unikernel, which can be listed out, and used to stop the appropriate unikernel. It is also able to list out

the available unikernel images as well as the currently running unikernels, with all required information by serializing the internal data structures containing the appropriate information.

3.6 Applications

The previously described components of our PPCE allow users to execute code using the application services while preserving privacy. Applications can perform arbitrary computations, including querying a database, a website, and compiling statistics from that data. A very important concept in the application services is the information grading as well as taint tracking. In Section 2.1 on page 5, we introduced the concept of information flow, which is a formalized system for how information flows through a system and to determine the resulting information classification based on the information it receives as input and uses for computing.

In practice, it could return the same information as it received and not do any operation on it, which would be of no issue. It could also potentially query information it previously had not been tainted with, meaning it would be tainted with that information access. Resulting in the clearances tag of the service being modified, prohibiting some users from using it due to insufficient clearance.

On the other hand, the application could be retrieving information that the end user is prohibited to retrieve, but through a certified code path that can declassify the information. An example of such a declassification could be retrieving information about a number of users, but sending the user information through a function that only yields some statistical measurement of the input data. The resulting statistics could then possibly be classified lower than the original raw data, and the user has clearance to view the declassified statistical information, but not the raw data. On the other hand, now, the system is tainted with the higher classified data, and it's clearances must be updated to reflect that, meaning the original user may be unable to use the same instance again.

Information flow, and in particular the lattice model makes it possible to design clearance systems which are not directly vertical, but more complex, as described in Section 2.1 on page 5. Such a system could be used with health data from athletes, where the player themselves would belong to a special class, trainers to another one, researchers one and medical professionals a different one. Some applications may require information from multiple classes, meaning the user must also have clearance to view both classes, and the application

service would be tainted with information from both classes, resulting in them not being usable by users without all required clearances.

An example application could fetch certain predetermined fields from a database of health records. A user may obtain all data related to him or her, while there may be certain types of information different classes of health professionals are prohibited from receiving. All of these would be different applications. In short, each application service should be single-purpose both to make them easier to audit for bugs and making them light-weight, resulting in the spawn time for them being lowered.

3.7 Modularity

All the different parts of the system use services from the other components. This only require that the component adhere to the Application Programming Interface (API) required by the requesting service, but does not enforce anything regarding internal implementation. Making the system modular in such a way that all components could be swapped out for new ones as long as they provide the same interface. This is due to the microservices architecture implemented by PPCE. Meaning we can take a component, change its internals slightly to yield better throughput or lower latency, and swap the components, or have them running side-by-side. In the experiments chapter, we show how we can change different components to drastically change the user-perceived end-to-end latency.

4

Evaluation

This chapter will evaluate the PPCE by conducting experiments to measure the end-to-end latency, and discussing the results. The initial version of PPCE described in Chapter 3 on page 11 will be evaluated first. Further, two microbenchmarks will be performed to determine the latency impact of the two operations; creating a new tap device and attaching it to an existing bridge, and starting and connecting to a unikernel. Because the initial version of PPCE is fully Just in Time (JIT), two system optimizations are proposed, implemented and their latency evaluated. Finally, the estimated and measured latency results of all three versions of PPCE are compared.

4.1 Experimental Setup

Figure 4.1 on the next page illustrates the hosting environment when all system components of PPCE are hosted on the same machine. KVM is the hypervisor at the very bottom of the stack, executing directly on hardware, which has been excluded from the figure. As explained in Section 3.5 on page 17, VMMD is executed as a process on top of a Linux host. Because unikernels are VMs, the unikernel components of PPCE execute directly on top of KVM. They are illustrated by the combination of both a kernel and an application combined to emphasize that unikernels are applications compiled together with the OS kernel into a single image.

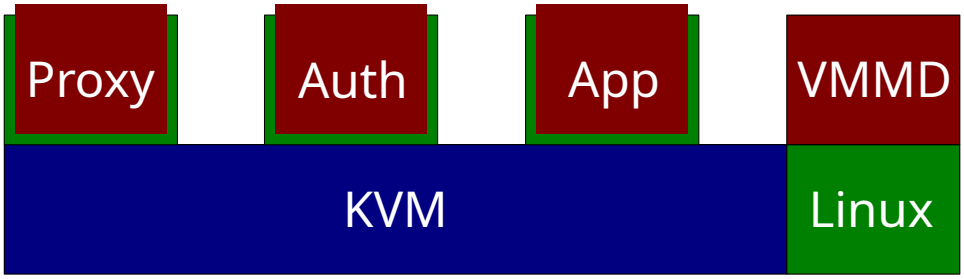


Figure 4.1: Hosting environment for the unikernels: Proxy, Auth and App, and the Linux process: VMMD

All three full system experiments will be hosted according to Figure 4.1 on the same machine. By hosting all components of PPCE on the same machine, the network latency will be reduced, providing more stable experimental results.

In order to have confidence in the experimental results, a sufficiently large sample size is required. All full system experiments are conducted with 20 samples, and the PPCE is restarted between each sample, removing potential artefacts from previous runs that potentially can influence the experimental results. The unikernel boot microbenchmark was also conducted with 20 samples, due to an issue discovered during experimentation which resulted in the time to run a sample increasing drastically. The microbenchmark for creating a new tap device and attaching it to an existing bridge resulted was run with 1000 samples, because of the absence of the type of issue encountered with the unikernel boot microbenchmark. For all experiments, the mean and standard deviation latency are provided as results.

It is important to note the computing environment in which the experiments have been carried out. The experiments have been run and compiled on a computer running Solus Linux with the following configuration:

- Linux kernel version 5.0.
- Intel Core i7-8650U with 8 logical cores and 4 physical running at 4.2GHz.
- 24GiB Double Data Rate (DDR)4 Random Access Memory (RAM) at 2400MHz.
- Samsung PM981 Non-Volatile Memory Express (NVMe) Solid State Drive (SSD).
- OCaml version 4.07.0

- MirageOS version 3.5.0

4.2 Network Setup

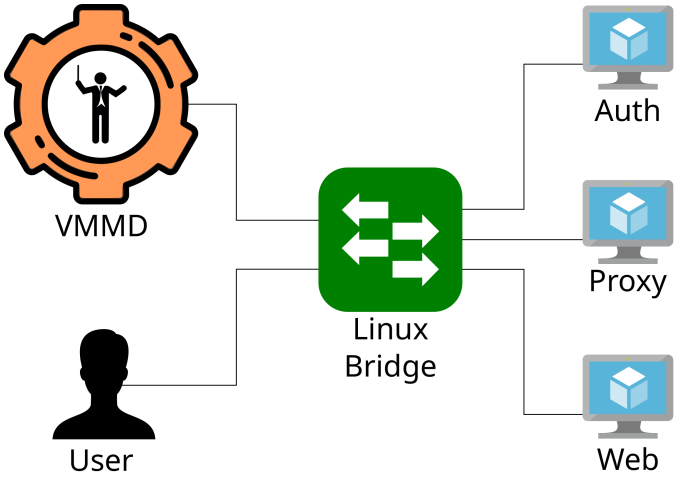


Figure 4.2: Network setup for system experiments showing all components communicating via a Linux bridge.

Figure 4.2 illustrates the network setup of the architectural drawing Figure 3.1 on page 13. All unikernels are attached to the same Linux networking bridge with each of their own virtual ethernet devices, called taps. The bridge functions as a virtual router, and will route traffic in its defined subnet to the appropriate tap devices, resulting in the traffic arriving at the correct unikernel. When the bridge receives data which it does not know where to route, it will be routed outside the bridge, and into the rest of the Linux networking subsystem, allowing bridged devices to communicate with services outside the bridge, meaning services running on the host, or even outside the computer. VMMD will start the VMM processes that are running the unikernels, meaning it must be running as an application on the host operating system. Because the bridge routes a private subnet, the requests bound for the unikernels will be routed through the bridge and to the specific tap devices attached to the correct unikernel.

4.3 Initial Full System Experiment

We want to perform an initial full-system experiment, measuring the user-perceived end-to-end latency. Note that this experiment is performed in a cold-boot manner, meaning there are no running application services that can

handle the request of the user.

Between each run, the system is restarted, removing potential artefacts from previous runs that potentially can influence our experiment. Each experiment is carried out by a user with a bearer token requesting a resource, which must be authenticated before handed off to the correct application service. Each of the requests are submitted manually using Postman[1], which will measure the end-to-end latency.

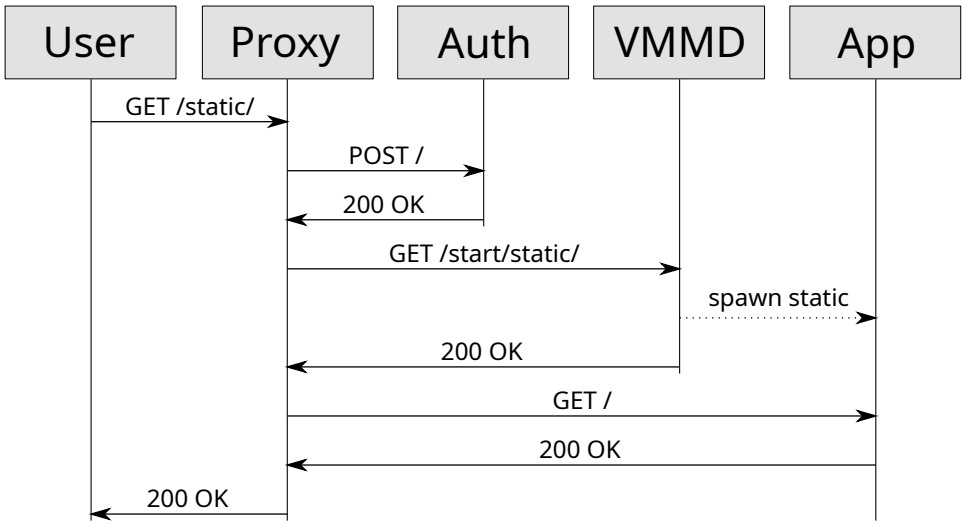


Figure 4.3: Initial full system experiment with communication flow between components.

As shown in Figure 4.3, the full system experiment involve four different services, and a total of four requests in addition to the spawning of the unikernel by VMMD. When the user requests a resource, the request will contain a bearer token for identifying the user, and determining the user's clearances. This request will initially hit the proxy which will request the service of the auth service to determine the clearances of the requesting user. The auth service will find the bearer token and the clearances for that user, and return it to the proxy service. The proxy service will then find an appropriate application service matching both the application requested and the user clearances as provided by the auth service. As this is a cold-boot experiment, the proxy will not find a suitable application, and will request VMMD to spawn one. VMMD will search through its registered applications, find the correct one, create a new tap device and attach it to the bridge, before spawning the unikernel with the correct clearance level, and return the address of the newly spawned

1. https://www.getpostman.com/

unikernel back to the proxy. After which the proxy service will register the newly spawned unikernel in its internal routing table before requesting the appropriate resource from the application service, and trying until the unikernel is ready to receive network traffic and accepts the connection. The application service will handle the request in a way it seems fit, which for the sake of this experiment will return a static string depending on the clearance level at which it was spawned. Once the proxy receives the response from the application service, it will relay the response back to the user, concluding the experiment.

Description	Time(ms)
mean	63.15
std	7.56

Table 4.1: Statistics from the initial full system experiment over 20 samples. Figure 4.3 on the facing page shows the experiment's communication flow. PPCE is restarted between each sample to remove potential artefacts.

Table 4.1 shows, over 20 runs, the mean end-to-end latency for a user is 63.15 ms, with a standard deviation of 7.56 ms. Contrasting this to the end-to-end latency for subsequent requests to the same resource from the same user, which is approximately 4 ms, means that using the services of VMMD significantly slows down the end-to-end latency for a request. The most time consuming tasks performed by VMMD are creating and attaching the tap device to the bridge, and spawning the unikernel. Which will be benchmarked next to determine the magnitude of their impact on the full-system end-to-end latency.

4.3.1 Discussion

Though the latency statistics shown in Table 4.1 could be perfectly acceptable in a number of contexts, in particular given the number of resources commonly fetched from a web sever to make up a website and how great the user-perceived latency for loading a web page is [60]. We hypothesize that we can lower it further, making it more viable as a hosting solution. Because this experiment will not take into consideration the network latency, as we are only using the local networking subsystem of Linux, the actual internet latency would potentially be considerably higher, further emphasizing the problem of the mean latency being above 60 ms. Conversely, if this would be used in conjunction with a website requiring information from different sources, the amount and latency induced by other requests would reduce the significance of the latency from our system [61].

4.4 Microbenchmark: Creating and attaching tap to bridge

We previously explained that creating the new tap device and attaching it to the bridge are some of the most time consuming actions performed by the vmmd, and to properly measure the impact from creating the tap device and attaching it to the bridge, we create this micro benchmark where we isolate the two actions and separate them from all other actions. In practice this means that we need to create a new application made just for benchmarking this particular operation.

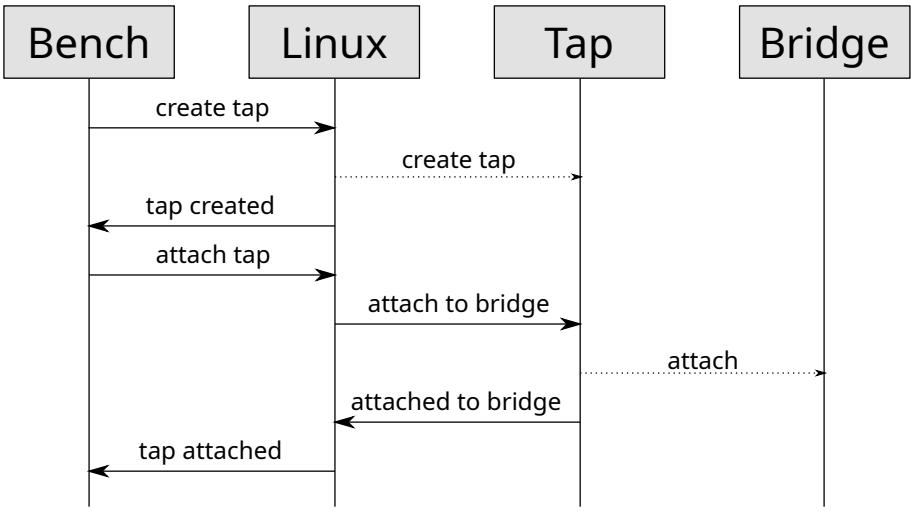


Figure 4.4: Microbenchmark where a tap is created and attached to an existing Linux bridge with process flow for illustrative purposes.

As illustrated by Figure 4.4, the experimental setup for the creation of the tap device and attaching it to a bridge, showing the steps that must be taken, and that are involved with the microbenchmark. Technically, the illustration is not entirely truthful, but serves well as an illustration of the actions required to create the tap and attaching it to a bridge. Both the tap and bridge columns of Figure 4.4 are parts of the networking subsystem in Linux, meaning it really is Linux which will perform all actions requested by the benchmarking application.

We created a new benchmarking application that measures the time, creates the new tap device, then attaches it to the existing bridge before stopping the time. After which the tap device is removed from the bridge and deleted. This is the cycle of each point of measurement in the benchmark. After performing 1000

runs, we calculate the mean and standard deviation of the time for creating the new tap device as well as attaching it to the bridge. Both actions are performed in the same manner as VMMD does.

Description	Time(µs)
mean	6788.9
std	3232.7

Table 4.2: Statistics from the microbenchmark creating and attaching a tap to a bridge over 1000 samples. Figure 4.4 on the facing page shows the process flow.

As we can see in Table 4.2, the creation of the tap and attaching it to the bridge micro benchmark yields a mean of 6788.9 µs and a standard deviation of 3232.7 µs, over 1000 runs. This seem significant, but does not account for the bulk of the time difference between the hot and cold request experiments for the full system.

4.4.1 Discussion

For each experiment run, the tap was deleted after the timer was stopped as to reduce the amount of potential side effects of the system, as well as to provide as accurate results from the benchmark as possible. As show in Table 4.2; creating a new tap device and attaching it to the kernel resulted in a mean of 6.79 ms and a standard deviation of 3.23 ms over 1000 samples. This sample size should be sufficient to get accurate results, but the standard deviation is still significant in comparison to the mean, which may be due to a number of different factors. Such as the other applications running on the system, as well as different forms of caching and buffering in the Linux networking subsystem, which have not been investigated further.

It is not surprising that the creation of a new tap device and attaching it to a bridge is somewhat time consuming, given the method that it is created with. In particular, the first non-existing tap is discovered by probing for a particular tap id, and continually increasing that id until we discover a non-existing tap device, which will be the tap device we create, and attach to the bridge. Both of these operations currently require both spawning new processes, as well as transitions to the Linux kernel for performing the actual networking subsystem tasks for the processes.

Given the mean time required to create the new tap device and attaching it to the bridge, we would expect that only preallocating tap devices before spawning the unikernels would reduce the mean end-to-end latency for a user from the first initial experiment to be reduced to $63.15 - 6.79$, namely

56.36 ms. This reduction is significant; just over 10%, but not as significant as we would want. Though this reduction is at only 10%, we need to implement the improvement into the full system and perform another experiment to have conclusive evidence, as there will be some difference between the isolated event of the microbenchmark and the full system experiment.

4.5 Microbenchmark: Unikernel boot

The other significant action performed by the VMMD is booting the unikernel, which technically is a VM, and might account for a large part of the latency experienced for a user in the cold-boot experiment. To properly isolate the booting of the unikernel, as well as ensuring it is ready to receive network traffic, from other actions performed by the rest of the system, we build a custom benchmarking application just for this purpose.

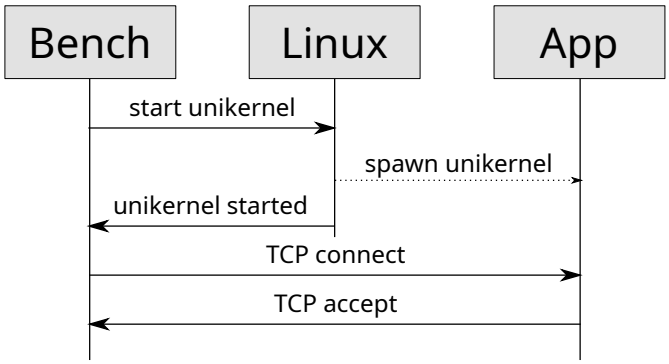


Figure 4.5: Microbenchmark where a new unikernel is started and connected to over TCP. The communication flow omits TCP connect retries.

We can see in Figure 4.5 that the unikernel experiment consists of spawning the unikernel, then attempting to create a TCP connection to it, and retrying until it is accepted, much in the same way the proxy operates. A plain TCP connection is used instead of a Hypertext Transfer Protocol (HTTP) request, yielding better isolation of determining when the unikernel is ready to receive network traffic, ignoring the application layer handling of the HTTP request and transferring data. After the TCP connection is accepted, it closes the socket and stops the timer. Initially, we encountered issues with initially reusing the same tap as

the previous unikernel, resulting in latency of upwards to 1000 ms. Due to this, the benchmarking application needed to perform extra actions beside starting the unikernel and connecting to it before stopping it. In particular, before spawning the unikernel, the benchmarking application will also create a new tap device and attach it to the bridge before the timer is started. After stopping the timer, the unikernel is killed, the tap device is destroyed, and the benchmarking application sleeps for several seconds to allow the internal routing table caching in Linux return to normal, resulting in eliminating the false measurements of approximately 1000 ms.

Description	Time(µs)
mean	21731.8
std	2361.2

Table 4.3: Statistics from the microbenchmark booting and connecting to a unikernel over 20 samples. Figure 4.5 on the preceding page shows the communication flow.

Over 20 runs, we see in Table 4.3 that the mean time from spawning the new unikernel and connecting to it over TCP is 21 781.8 µs, with a standard deviation of 2361.2 µs. While the creation of a new tap device and attaching it to the bridge doesn't account for large parts of the extra latency experienced, the spawning of the unikernel will account for a larger portion. Combined, these two account for a very significant portion of the difference between the cold-boot experiment, and requests performed when the unikernel is already spawned, circumventing VMMD.

4.5.1 Discussion

Table 4.3 shows that when performing 20 samples, we saw a mean of 21.73 ms and a standard deviation of 2.36 ms . Only 20 samples were taken due to the amount of time required to run a sample. Conducting the experiments, it was discovered that a sample could be executed in approximately 21 ms, but some samples would require just over a second to complete. This initially resulted in a very large standard deviation. Our hypothesis is that this effect is experienced due to the networking subsystem in Linux, probably by it caching its routing table with a collision of IP addresses with different Media Access Control (MAC) addresses. Due to this issue, we determined to add a pause in the benchmark allow the bridge with its routing table to flush its cache before spawning a new unikernel. After this pause was included, we got the results presented in the experiments chapter.

Compared to the tap bridge microbenchmark, the implication of this task seem to be much greater. Spawning the unikernel cannot be offset without

creating and allocating the tap and attaching it to the bridge, meaning our hypothesized new latency must include that as well. With this assumption; both allocating the new tap device and spawning the unikernel AOT, the new, improved latency should be approximately 63.15 − 6.79 − 21.73, resulting in 34.63 ms in expected end-to-end latency. This is not including the elimination of communicating with VMMD to have it spawn the new unikernel. Meaning the actual expected end-to-end latency should be somewhat lower than 34.63 ms. This results in an expected 45% end-to-end latency reduction. A 45% reduction of end-to-end latency is a big leap, meaning that the system in theory would be able to support approximately double the number of clients during the same period of time compared to the initial unoptimized system.

A latency of approximately 35 ms would be acceptable, but one would not expect the rest of the system to yield such high latency. Again, we would need to implement the optimizations in the full system and run the experiment again to get realistic values.

4.6 Improved Full System Experiment

The results of both microbenchmarks show that there are a potential for reducing the user-perceived end-to-end latency of the full system if we can reduce the amount of time spent performing these two actions. One promising technique would be to hide the latency associated with creating the new tap device, attaching it to the bridge as well as booting the unikernel. The latency of these could be hidden by performing all or some of these actions AOT, instead of JIT. To experiment with these solutions, we have implemented the improvements and will perform the same full system experiment as with the initial system without any optimizations.

To reiterate the full system benchmark, the users token must be sent to the auth service for validation and to retrieve the clearances of the user before the proxy will request VMMD to spawn a new unikernel, before the proxy performs the request for the user, and finally returning the response from the application. Hiding the latency from creating the tap device and attaching it to the bridge can be done by spawning appropriate unikernels AOT instead of JIT, or a hybrid approach by allocating a pool of attached taps AOT and spawning unikernels JIT.

4.6.1 Ahead-of-time spawning of unikernels

If the proxy spawns a number of unikernels AOT which is ready to receive requests, the latency is hidden, and when a user requests the proxy, it should not need to access the services of VMMD, meaning the end-to-end latency should be close to that of the end-to-end latency experienced by users for subsequent requests. Depending on the unikernel life-cycle, the results from this experiment could be viewed as the end-to-end latency for a warm system.

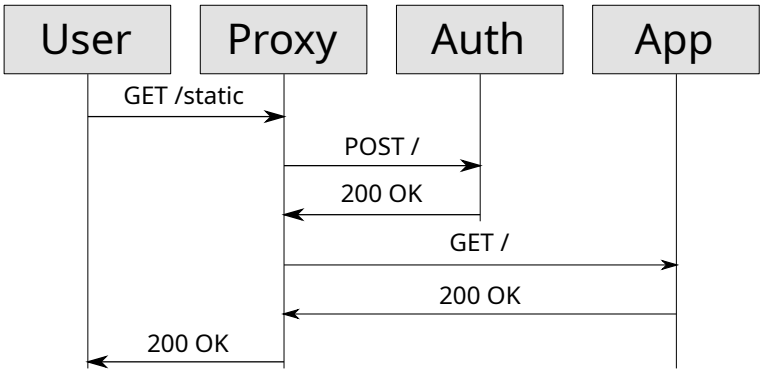


Figure 4.6: Optimized full system experiment, where all unikernels are booted AOT, with communication flow between components.

As Figure 4.6 depicts, for the AOT spawning of unikernels experiment, the proxy will initially request the VMMD to spawn a unikernel at each clearance level, as to cater to all possible users without the need for the proxy to spawn a new unikernel including the allocation of a new tap device and attaching it to the bridge. This means that the proxy already has multiple application services registered in its internal routing table, resulting in the proxy not needing to request VMMD, enabling the proxy to forward the request to the appropriate application service.

Description	Time(ms)
mean	4.45
std	1.15

Table 4.4: Statistics from the AOT experiment over 20 samples. Figure 4.6 shows the communication flow.

The results of this experiment in Table 4.4 clearly shows that spawning and

registering unikernels AOT is very beneficial to the user-perceived end-to-end latency. Compared to the initial full-system experiment, we see a mean end-to-end latency of 4.45 ms with a standard deviation of 1.15 ms, which compared to the JIT full system experiment is a 14.2x reduction of mean latency and a 6.6x reduction of standard deviation.

As we will discuss later; this significant latency reduction is not achieved without problems.

Discussion

Based on the results of the microbenchmarks, the end-to-end latency should be around 34.63 ms. But as we have seen in Table 4.4 on the preceding page, the actual end-to-end latency for the AOT full system experiment resulted in a mean of 4.45 ms and a standard deviation of 1.15 ms, which is vastly different from our initial expectations. The only possible conclusion we can draw from this is that one or both of the microbenchmarks results in smaller numbers than their tasks actually take in the full system. Applying the AOT optimizations results in a 93% end-to-end latency reduction, which would translate to the system being able to serve over 14x the number of users within the same time span as the original system without any optimizations.

This optimization results in a massive reduction of the user-perceived end-to-end latency, but it comes at a cost. The application tested only require 16MB of RAM, but with one application service running for each of a multitude of clearance levels, the small footprint of each service will result in a large system footprint. This is further enhanced by providing a number of applications that each need to have multiple clones running at the same time to accommodate all users' needs. With 24GB of RAM, it would only be possible to accommodate 1500 executing unikernels of this application, discarding the memory requirements of all other system components as well as the Linux kernel itself. The current configuration contains 6 distinct clearance levels, resulting in the system only being able to accommodate 250 applications in total, provided each of them only require 16MB of RAM. We ca easily see that this does not scale very well, as all applications are expected to provide a very specialized service, and one could expect a system needing to accommodate thousands of small, specialized application services, which would require all too much memory. It would also greatly increase the time required to start the full system. This would deem the fully AOT optimization unusable regardless of its very beneficial end-to-end latency.

4.6.2 Hybrid tap pool

The previous latency reduction improvement resulted in a great reduction of latency, but has scalability issues. Another approach to reducing the user-perceived end-to-end latency is to allocate the tap devices and attach them to the bridge AOT, while still spawning unikernels on demand, as in the initial experiment.

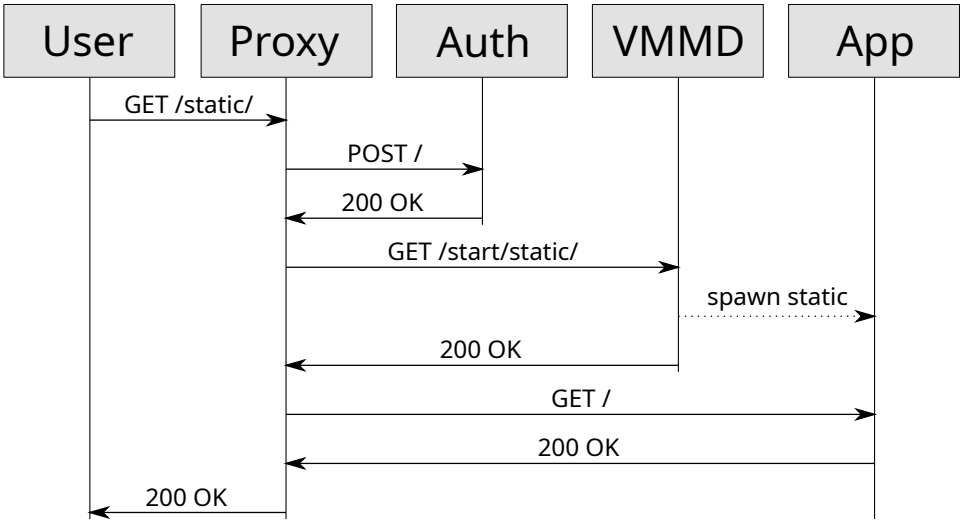


Figure 4.7: Optimized full system experiment, where unikernels are booted JIT, but tap devices are created and attached AOT, yielding a hybrid solution, with communication flow between components.

We can see from Figure 4.7 that we introduce VMMD as a component again, but allowing it to skip the allocation and attaching of tap devices to the bridge by allocating a pool of them AOT. Meaning VMMD will only need to fetch an already allocated tap device from its internal pool, and use it when spawning the new unikernel, reducing the amount of work carried out by VMMD on demand.

Description	Time(ms)
mean	29.35
std	3.92

Table 4.5: Statistics from the hybrid experiments over 20 samples. Figure 4.7 shows the communication flow.

Intuitively, one would assume the latency reduction to be relatively close to the initial full system experiment when subtracting the time required for the tap creation microbenchmark. But as Table 4.5 shows, with mean latency of

29.35 ms and standard deviation of 3.92 ms, this is not the case. If one were to calculate the intuitive expected mean latency of 63.15 ms−6.79 ms = 56.36 ms, we get a difference of 27.01 ms, which is not accounted for. In practice, the latency reduction achieved by allocating the pool of tap devices AOT while still spawning unikernels on demand, we achieve a 2.2x reduction of mean latency and a 1.9x reduction of standard deviation.

Discussion

As discussed in Section 4.4 on page 26, the expected latency reduction of applying this optimization was only 10%. On the other hand, when actually performing the experiment on the hybrid system, the results differ greatly from the expected values. As seen in Table 4.5 on the preceding page; with 20 samples, we saw a mean of 29.35 ms and a standard deviation of 3.92 ms, meaning a reduction of 63.15 − 29.35, 33.8 ms, which is much greater than our expected improvement.

Based on these data, we can assume that the microbenchmark for allocating and attaching tap devices to the bridge either was subject to great optimization compared to the rest of the system, the implementation was not correct, or that the Linux kernel performed some optimization when performing the benchmark that was not experienced in the full system. Comparing the initial full system experiment with the new hybrid approach, where the only difference is getting a preallocated tap from the tap pool, we can see the practical overhead of spawning and attaching tap interfaces to the bridge on demand as we do in the initial full system experiment. This reduction is 33.8 ms, meaning the practical overhead of creating and attaching the tap interface seem to be the major source of latency in the system as a whole, yielding the optimization more beneficial than originally anticipated.

Comparing the end-to-end latency of the hybrid approach to the fully AOT approach, we see that the additional overhead associated with both communicating with VMMD and having it spawn the unikernel is only 29.35 − 4.45, 24.9 ms. This result is close to our expectation from the microbenchmark of spawning unikernels, but is slightly higher due to the microbenchmark not taking into account the communication with VMMD, fetching the tap from the pool, returning JavaScript Object Notation (JSON) from VMMD, and parsing it in the proxy service.

Spawning tap interfaces AOT will not incur great overhead, and seem to be an optimization without much cost associated with it. This optimization yields relatively low latency, without the extreme memory requirements of spawning all unikernels fully AOT.

4.7 Summary

In this chapter, we have described and shown results of an initial system together with two different system improvements to reduce the user-perceived end-to-end latency. Along these three experiments, we have used two microbenchmarks to illustrate the amount of time required for different tasks to complete, and where we can apply techniques for latency reduction, such as latency hiding, as we have done here.

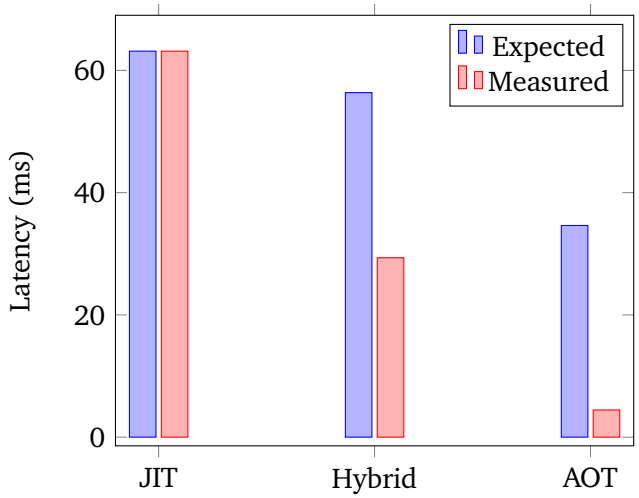


Figure 4.8: Comparison of the expected and measured values for initial, hybrid and AOT implementations of PPCE.

Figure 4.8 shows our two different approaches to achieve latency reduction both succeeded and exceeded initial expectations, resulting in a 14.2x and 2.2x reduction of latency for spawning unikernels AOT, and allocating tap pool AOT, respectively. In addition, it also shows that the difference between expected and measured latency is 27.01 ms and 30.18 ms for the hybrid and AOT optimizations, respectively.

5 Discussion

We previously discussed the experiments, and their results. In this chapter we will discuss the feasibility of the design, security and privacy properties, as well as possible shortcomings of the system.

5.1 Security and Privacy

The concept privacy-preserving in the context of this thesis is meant to encompass the system to preserve the privacy of its users by both access restriction and reporting of user behavior and data access. One of the concepts we draw from information theory and information flow control is the concept of secure information flow. If an information flow is considered safe, there is a guarantee that the user will only receive data to which the user have the correct clearance. Which is only true if the system is without any bugs that let the user accidentally or maliciously gain access to data to which it should not have.

Our system will allow application developers to create their own information flow control system in addition to the benefits provided by the design of our system. Applications are programmed using statically compiled OCaml, which has a strong type system, and is suited for formal verification, and there are tools available that can extract propositions that can be formally proved using tools like Coq [62], [63]. The combination of our applications and operating system being implemented in OCaml, and application services only serving users at

a specified clearance level, or only a single user, will result in a reduction of potential information leakage in case of either a logical programming mistake, a bug, or a security vulnerability. Compared to other systems, this would reduce the Trusted Computing Base (TCB) and the resulting unikernel will only contain the code required for it to perform its tasks, and nothing more.

In QEMU, the most commonly used VMM for Linux, a bug in a floppy device driver [33] resulted in the guest being able to execute arbitrary code on the host operating system, which is critical in a system handling sensitive information. This happened regardless of the guest operating system needing to use a floppy device or not, which for our system would not be the case, as the entire operating system is statically linked, resulting in all components not used by the application via the operating system being stripped out.

Studies show [64] that applications written in C and C++ are still very prone to errors, which at best results in a crash, and at worst results in a total system compromise, and that the number of bugs in an application is proportional to the number of lines of code. Not only are there fewer bugs per line of code of OCaml, but the number of lines will be drastically different, due to the statically linkage of the operating system components together with the application, as well as the VMM also being statically compiled based on the services required by the unikernel, the trusted computing base will be reduced dramatically. As an example, one could opt not to use network, but Vchan, which is a high speed shared memory-only alternative to using a network stack, originally from the Xen hypervisor. Using Vchan would result in an attacker not being able to directly use network to either exfiltrate data, or to pivot to different services using the network. If something is not used by the application service, it is not present, meaning an attacker discovering a critical bug would not be able to use any services the guest unikernel is not using.

We have previously stated that OCaml and MirageOS unikernels usually tend to result in more secure and safe applications, and the Bitcoin Piñata [65]. is a testament to the security of these unikernels. The Bitcoin Piñata is a project where a unikernel using TCP, TLS and X.509 certificates were built, and it is intended as an open and transparent bug bounty. The piñata contains a wallet private key, meaning that if you are able to find vulnerabilities and exploit them such that you are able to leak the private key, you can use the leaked private key to transfer the Bitcoins to your own account, displaying to the world who managed to exploit it. It has been online since march of 2015, but the majority of the funds were allocated to other projects after three years, in march of 2018. During that time, and still, there has not been another transaction on that account, meaning the piñata has still not been broken after being online for over 4 years, and the first three years, it contained 10 Bitcoin (BTC), which at todays rate would be $80 000 , and at the very top - during Christmas of

2017, where the bug bounty would be worth $197 830.6

5.2 Service Lifetime Configuration

PPCE can also be configured to support different styles of application service lifetimes, such as either only having one service serving only one request, a service serving only one user, or to serve a class of users.

Each of which can automatically be killed after a certain amount of time where the service is not used, both to allow the system host to regain resources allocated by the VMs. This form of life-cycle for the unikernels would also promote the privacy aspect of the system as well, regarding the amount of information that can be leaked from the system in case a bug is discovered and exploited in such a way that a user may gain access to additional information.

Different service lifetimes will lead to different system constraints in terms of resource requirements, user-perceived latency and the privacy and safety of the systems in case of information leakage and other bugs. Most web services and other applications have shared instances of the service for most users, regardless of their user clearances, meaning that a information leak bug could potentially lead to private and sensitive information being leaked to unauthorized users.

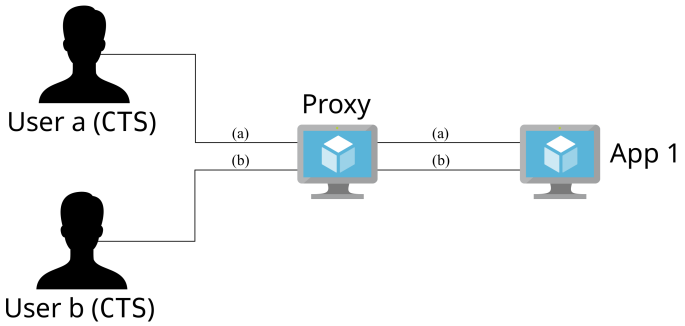


Figure 5.1: Lifetime configuration where users with the same clearance can use the same application instance

The least privacy-preserving lifetime configuration of PPCE, illustrated by Figure 5.1 would be an entire class of authorized users sharing the same computing service, information leakage would in this case be reduced to the information retrieved by other users with the same clearances as you. Although never a desired property, information leakage can potentially represent an acceptable risk in certain circumstances to balance the privacy-preserving aspect of the system with the resource footprint and user-perceived latency.

When sharing an instance between all users of a class, only the first request in a class will result in a new unikernel being spawned, and for all subsequent requests, the latency will be minuscule, as we shown with the fully AOT full system experiment.

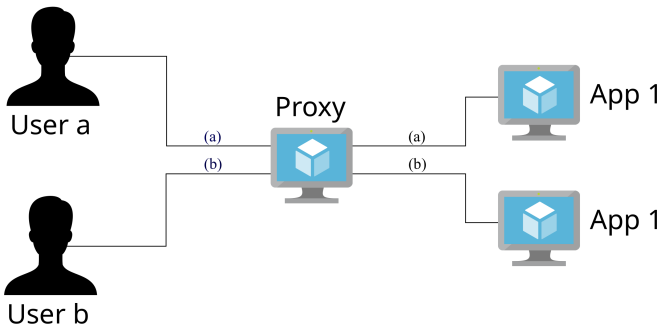


Figure 5.2: Lifetime configuration where each user get their own application instance

Figure 5.2 shows the lifetime configuration where unikernel is restricted to a single user, and not an entire class of users, any information leakage from bugs would only be accessible to the user, that previously requested that information. The risk associated with this type of information leakage would be even smaller than that of sharing an instance with entire classes of users. On the other hand, it would require as many instances as there are active users, plus some more because the instances would still be available for each user for some particular amount of time before they are deallocated, and subsequent requests would result in a new instance being spawned. But the average user-perceived end-to-end latency would grow greater for users, even though only the first request will require VMMD to spawn a new unikernel.

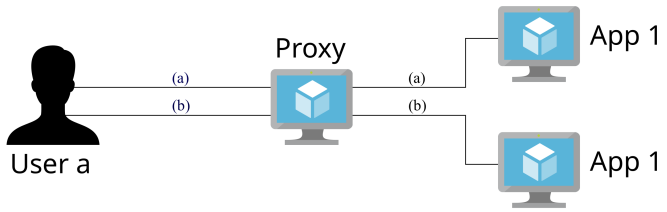


Figure 5.3: Lifetime configuration where each application instance only handles one request

The configuration with the least risk would be spawning a new instance for each new request, and destroying the instance upon request completion. This is illustrated by Figure 5.3 and would result in the greatest user-perceived end-to-

end latency, but probably lower resource footprint than allocating an instance for each user, as the unikernels would need to be kept alive for some time after the user has completed its requests. It would require the greatest amount of computation to be completed by the system, and in particular VMMD.

Different use-cases would result in a different choice of service lifetime configuration, and for some particular use-cases, it would probably be enough to allow entire classes of users to use the same application service instances. On the other hand, applications closely related to a user's private, and often sensitive, information would benefit from providing each user with an instance of their own. In 2012, the tax return for a Norwegian man named Kenneth was available for all users that logged in to the Altinn system to check their own tax return [66]. Using a system like ours, with a user-specific service lifetime configuration, the probability of such an event happening would be dramatically lowered, as no users but Kenneth would have access to the system where the data of Kenneth resides. Though we cannot say for sure the event would not have occurred, as software will inevitably have a certain number of bugs, but designing the system to take into account that programmers make mistakes and bugs do occur will increase the system resiliency and help preserve user privacy.

5.3 System Shortcomings

Though we describe the system as privacy preserving and specify that application services are only containing information for the specific class of users, or specific users themselves, our current implementation do not perform the same kind of separation for other services, which may show itself as a potential privacy or security concern. All users currently share the same auth service, as well as the same VMMD to spawn the different unikernels at different clearance levels. This current implementation detail might prove problematic in the face of vulnerabilities. On the other hand, neither VMMD, nor the auth service handle any of the private or sensitive information that the application services would handle, and one can argue that the privacy of the users are still preserved.

The proxy service currently also is shared between all users of the system, and may in some regard be considered a critical flaw, potentially jeopardizing the privacy preserving aspect of the system. In case of taint tracking, it will lead to the proxy service being tainted with the highest level of clearances, meaning the proxy can no longer serve requests at a lower level without rendering the information flow insecure. Regarding secure information flow within the application services, it would not pose any problems with a monotonically

increasing clearance level, as we showed initially. If users are properly tagged, as they should be, this would also be the case when dealing with more complex clearance lattices. The issue on the other hand is that the current system design only handles the taint tracking on a user-basis, meaning a user with the maximum clearance level performing an action requiring a lower level of clearances would end up improperly tainting the data with a higher clearance level than what is necessary, something that is not desirable.

6 Conclusion

This chapter will conclude the thesis by summarizing our results and relating them to our thesis statement. In addition, we will compare PPCE to other systems and describe future work.

6.1 Related Work

PPCE is a unikernel-based distributed system enabling users to perform privacy-preserving computations. There are other related works either unikernel-based or privacy-preserving, some of which inadvertently providing parts of the privacy-preserving aspects of this system. Albatross [67] is closely related to VMMD in our system, effectively enabling developers to build a similar unikernel-based distributed systems, but it is not a full system enabling privacy-preserving computations for its users. Jitsu [68] is a DNS server providing JIT spawning of unikernels for resources by DNS name, which results in each DNS request to spawn a new unikernel to handle that resource. If coupled with authentication, it could enable privacy-preserving computations, but as it is a DNS server, the user would need to be authenticated at the application service in addition to at the DNS server. The unikernels spawned by Jitsu can either be short-lived, providing a class- or user-specific unikernel to handle requests, or one-off unikernels providing the same possible service lifetime configuration as our system, but additional authentication is required at the application service to restrict malicious users to gain unauthorized access to an

application service and its resources by guessing the appropriate DNS name. One area where Jitsu shines in comparison with our system is regarding taint propagation to the proxy, as Jitsu has none. In Jitsu, each service would use TLS to communicate directly with the user, maintaining the information flow secure. In terms of implementation, Jitsu is using Xen as a backend, compared to our system, which is using KVM for virtualization.

6.2 Concluding Remarks

In this thesis we have shown the design and implementation of PPCE; a unikernel-based privacy-preserving computing environment. We then ran experiments, resulting in implementation of both an AOT and a hybrid version of PPCE. The unoptimized system achieved a mean user-perceived end-to-end latency of 63.15 ms.

With our fully AOT version of PPCE, we managed to reduce the mean end-to-end latency to 4.45 ms, but concluded this approach does not scale well, and results in the system having a very large resource footprint. The hybrid version of PPCE, the mean end-to-end latency was reduced to 29.35 ms, less than half of the end-to-end latency of the unoptimized system performing both actions in a JIT fashion, without sacrificing scalability, and still maintaining virtually the same resource footprint as the original system.

Recalling our original thesis statement:

> *MirageOS can be used to build a microservice architecture for privacy preserving computing while providing reasonable user-perceived end-to-end latency.*

Using latency-hiding techniques, the system reduces the user-perceived end-to-end latency by 53.5%, while still maintaining it's goal of being privacy preserving. It also achieves this while still maintaining a system footprint comparable to that of the original, unoptimized system, enabling the system to potentially scale to the thousands of application services executing concurrently, while the user interacts as if with a single coherent system. Thereby confirming our thesis.

6.3 Future Work

With our current design, the proxy service will render the information flow insecure due to acting as a proxy for requests at all clearance levels. To solve this, we would in future work need to repurpose the proxy service into a redirection gateway, which based on authentication results would trigger an appropriate DNS request. In the same way Jitsu spawns unikernels on demand [68], a unikernel should be spawned at the correct clearance level, which would enable the application service to authenticate and authorize the user to access it's resources. By allowing the user to directly communicate with the application service, the information flow would be secure. Combining these aspects, a future system supporting different service lifetime configurations and privacy-preserving computations while maintaining the security of the information flow could be constructed.

It could also be possible to programmatically show that the information flow would be secure by performing internal routing and handing connections off inside of the proxy. This would not isolate different clearance levels from one another using different VMs, as the application services, but merely by the programmer showing and the compiler verifying that it should not be possible for the information flow to be insecure. Resulting in the possibility of vulnerabilities or other bugs leaking information to users that are unauthorized to view and possess that information.

Bibliography

[1] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the virtual library operating system," *Queue*, vol. 11, no. 11, p. 30, 2013.

[2] Å. Kvalnes, D. Johansen, R. van Renesse, F. B. Schneider, and S. V. Valvag, "Omni-kernel: An operating system architecture for pervasive monitoring and scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 10, pp. 2849–2862, 2014.

[3] H. D. Johansen, R. V. Renesse, Y. Vigfusson, and D. Johansen, "Fireflies: A secure and scalable membership and gossip service," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 2, p. 5, 2015.

[4] H. Johansen, D. Johansen, and R. van Renesse, "Firepatch: Secure and time-critical dissemination of software patches," in *IFIP International Information Security Conference*, Springer, 2007, pp. 373–384.

[5] A. T. Gjerdrum, R. Pettersen, H. D. Johansen, and D. Johansen, "Performance principles for trusted computing with intel sgx," in *International Conference on Cloud Computing and Services Science*, Springer, 2017, pp. 1–18.

[6] H. D. Johansen, E. Birrell, R. Van Renesse, F. B. Schneider, M. Stenhaug, and D. Johansen, "Enforcing privacy policies with meta-code," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, ACM, 2015, p. 16.

[7] E. Birrell, A. Gjerdrum, R. van Renesse, H. Johansen, D. Johansen, and F. B. Schneider, "SGX enforcement of use-based privacy," in *Proceedings of the 2018 Workshop on Privacy in the Electronic Society - WPES'18*, ACM Press, 2018. DOI: 10.1145/3267323.3268954.

[8] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976. DOI: 10.1145/360051.360056.

[9] C. E. Landwehr, "Formal models for computer security," *ACM Computing Surveys (CSUR)*, vol. 13, no. 3, pp. 247–278, 1981.

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

[11] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, Feb. 2000, ISSN: 1094-9224. DOI: 10.

1145/353323.353382. [Online]. Available: http://doi.acm.org/10.1145/353323.353382.

[12] L. Brandeis and S. Warren, "The right to privacy," *Harvard law review*, vol. 4, no. 5, pp. 193–220, 1890.

[13] E. Birrell and F. B. Schneider, "A reactive approach for use-based privacy," 2017.

[14] E. Birrell and F. B. Schneider, "Fine-grained user privacy from avenance tags," 2015.

[15] J. L. Martin Fowler. (Mar. 25, 2014). Microservices, [Online]. Available: https://martinfowler.com/articles/microservices.html.

[16] Dimensional Research, "Global microservices trends," Tech. Rep., 2018. [Online]. Available: https://go.lightstep.com/global-microservices-trends-report-2018.html (visited on 05/26/2019).

[17] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and ulterior software engineering*, Springer, 2017, pp. 195–216.

[18] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, IEEE, 2015, pp. 583–590.

[19] G. Kakivaya, L. Xun, R. Hasha, S. B. Ahsan, T. Pfleiger, R. Sinha, A. Gupta, M. Tarta, M. Fussell, V. Modi, *et al.*, "Service fabric: A distributed platform for building microservices in the cloud," in *Proceedings of the Thirteenth EuroSys Conference*, ACM, 2018, p. 33.

[20] Y. Tsuruoka, "Cloud computing-current status and future directions," *Journal of Information Processing*, vol. 24, no. 2, pp. 183–194, 2016.

[21] Docker Inc. (2018). Why docker? [Online]. Available: https://www.docker.com/why-docker (visited on 12/14/2018).

[22] Canonical Ltd. (2018). What's lxc? [Online]. Available: https://linuxcontainers.org/lxc/introduction/ (visited on 12/14/2018).

[23] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "Kvm: The linux virtual machine monitor," in *Proceedings of the Linux symposium*, Dttawa, Dntorio, Canada, vol. 1, 2007, pp. 225–230.

[24] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SIGOPS operating systems review*, ACM, vol. 37, 2003, pp. 164–177.

[25] R. J Adair, R. U Bayles, L. W Comeau, and R. J Creasy, "A virtual machine system for the 360/40 - cambridge scientific center report 320," May 1966.

[26] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[27] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th International Middleware Conference*, ACM, 2016, p. 1.

[28] D. Williams, R. Koller, and B. Lum, "Say goodbye to virtualization for a safer cloud," in *10th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, 2018.

[29] D. R. Engler and M. F. Kaashoek, "Exterminate all operating system abstractions," in *hotos*, IEEE, 1995, p. 78.

[30] D. R. Engler, M. F. Kaashoek, *et al.*, *Exokernel: An operating system architecture for application-level resource management*, 5. ACM, 1995, vol. 29.

[31] A. Happe, B. Duncan, and A. Bratterud, "Unikernels for cloud architectures: How single responsibility can reduce complexity, thus improving enterprise cloud security," *Submitt. to Complexis*, vol. 2016, pp. 1–8, 2017.

[32] D. Williams and R. Koller, "Unikernel monitors: Extending minimalism outside of the box.," in *HotCloud*, 2016.

[33] The MITRE Corporation. (2015). The venom vulnerability, [Online]. Available: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456` (visited on 12/14/2018).

[34] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services," in *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, IEEE, 2015, pp. 250–257.

[35] K. Stengel, F. Schmaus, and R. Kapitza, "Esseos: Haskell-based tailored services for the cloud," in *Proceedings of the 12th International Workshop on Adaptive and Reflective Middleware*, ACM, 2013, p. 4.

[36] 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*, USENIX Association, 2014, pp. 459–473.

[37] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library os from the top down," in *ACM SIGPLAN Notices*, ACM, vol. 46, 2011, pp. 291–304.

[38] A. Madhavapeddy and D. J. Scott, "Unikernels: The rise of the virtual library operating system," *Communications of the ACM*, vol. 57, no. 1, pp. 61–69, 2014.

[39] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, "The ocaml system release 4.02," *Institut National de Recherche en Informatique et en Automatique*, vol. 54, 2014.

[40] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys (CSUR)*, vol. 21, no. 3, pp. 359–411, 1989.

[41] R. Milner, "A proposal for standard ml," in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, ser. LFP '84, Austin, Texas, USA: ACM, 1984, pp. 184–197, ISBN: 0-89791-142-3. DOI: 10.1145/800055.802035. [Online]. Available: `http://doi.acm.org/10.1145/800055.802035`.

[42] C. Bozman, M. Mauny, F. Le Fessant, and T. Gazagnaire, "Study of ocaml programs' memory behavior," *OCaml Users and Developers*, 2012.

[43] Y. Minsky, "Ocaml for the masses," *Communications of the ACM*, vol. 54, no. 11, pp. 53–58, 2011.

[44] T. Imada, "Mirageos unikernel with network acceleration for iot cloud environments," in *Proceedings of the 2018 2nd International Conference on Cloud and Big Data Computing*, ACM, 2018, pp. 1–5.

[45] A. S. Tanenbaum and M. Van Steen, *Distributed systems*, 3rd ed. distributed-systems.net, 2017.

[46] CERT Coordination Center. (2014). Openssl tls heartbeat extension read overflow discloses sensitive information, [Online]. Available: `https://www.kb.cert.org/vuls/id/720951/` (visited on 05/30/2019).

[47] V. Cardellini, M. Colajanni, and P. S. Yu, "Geographic load balancing for scalable distributed web systems," in *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No. PR00728)*, IEEE, 2000, pp. 20–27.

[48] S. Sharma, S. Singh, and M. Sharma, "Performance analysis of load balancing algorithms," *World Academy of Science, Engineering and Technology*, vol. 38, no. 3, pp. 269–272, 2008.

[49] T. Hunt. (2013). Have i been pwned? [Online]. Available: `https://haveibeenpwned.com/` (visited on 05/30/2019).

[50] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The tangled web of password reuse.," in *NDSS*, vol. 14, 2014, pp. 23–26.

[51] B. Krebs. (Mar. 2019). Facebook stored hundreds of millions of user passwords in plain text for years, [Online]. Available: `https://krebsonsecurity.com/2019/03/facebook-stored-hundreds-of-millions-of-user-passwords-in-plain-text-for-years/` (visited on 05/26/2019).

[52] B. Ives, K. R. Walsh, and H. Schneider, "The domino effect of password reuse," *Communications of the ACM*, vol. 47, no. 4, pp. 75–78, 2004.

[53] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full sha-1," in *Annual International Cryptology Conference*, Springer, 2017, pp. 570–596.

[54] H. Kumar, S. Kumar, R. Joseph, D. Kumar, S. K. S. Singh, and P. Kumar, "Rainbow table to crack password using md5 hashing algorithm," in *2013 IEEE Conference on Information & Communication Technologies*, IEEE, 2013, pp. 433–439.

[55] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: New generation of memory-hard functions for password hashing and other applications," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2016, pp. 292–302.

[56] E. Casalicchio, "Autonomic orchestration of containers: Problem definition and research challenges," in *10th EAI International Conference on Performance Evaluation Methodologies and Tools. EAI*, 2016.

[57] M. J. Hammel, "Managing kvm deployments with virt-manager," *Linux J.*, vol. 2011, no. 201, Jan. 2011, ISSN: 1075-3583. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924401.1924408.

[58] Docker Inc. (2019). Docker swarm mode overview, [Online]. Available: https://docs.docker.com/engine/swarm/ (visited on 05/27/2019).

[59] D. Vohra, *Kubernetes microservices with Docker*. Apress, 2016.

[60] A. S. Asrese, S. J. Eravuchira, V. Bajpai, P. Sarolahti, and J. Ott, "Measuring web latency and rendering performance: Method, tools & longitudinal dataset," 2017.

[61] B. Briscoe, A. Brunstrom, A. Petlund, D. Hayes, D. Ros, J. Tsang, S. Gjessing, G. Fairhurst, C. Griwodz, and M. Welzl, "Reducing internet latency: A survey of techniques and their merits," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 2149–2196, 2014.

[62] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, *et al.*, "The coq proof assistant reference manual: Version 6.1," PhD thesis, Inria, 1997.

[63] E. M. Clarke, E. A. Emerson, and J. Sifakis, "Model checking: Algorithmic verification and debugging," *Communications of the ACM*, vol. 52, no. 11, pp. 74–84, 2009.

[64] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, ACM Press, 2014. DOI: 10.1145/2635868.2635922.

[65] Robur. (2015). The bitcoin piñata, [Online]. Available: https://robur.io/Projects/Pinata (visited on 05/21/2019).

[66] Nærings- og handelsdepartementet, "Dnvs rapport om altinn ii," 2012. [Online]. Available: https://www.regjeringen.no/no/dokumenter/dnvs-rapport-om-altinn-ii/id675893/ (visited on 05/27/2019).

[67] H. Mehnert. (2018). Albatross - provisioning, deploying, managing, and monitoring virtual machines, [Online]. Available: https://hannes.nqsb.io/Posts/VMM (visited on 05/21/2019).

[68] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. J. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, *et al.*, "Jitsu: Just-in-time summoning of unikernels.," in *NSDI*, 2015, pp. 559–573.