SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Towards running legacy applications inside AWS Nitro Type Enclaves

Jalil David Salamé Messina

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Towards running legacy applications inside AWS Nitro Type Enclaves

Auf dem Weg zur Ausführung älterer Anwendungen in Enklaven vom AWS Nitro Typ

Author:	Jalil David Salamé Messina
Supervisor:	Prof. Dr. Pramod Bhatotia
Advisor:	Dr. Le Quoc Do
Submission Date:	15th April 2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15th April 2024 Jalil David Salamé Messina

Acknowledgments

I would like to thank my family and friends for their support and their patience during the moments when I did not stop talking about my work. Especially to Max, Noah, and Eric, who helped me proofread this work.

I am very grateful to Prof. Pramod Bhatotia for this opportunity, and to my advisor Dr. Le Qouc Do for all the support and guidance he provided me during my time working with him.

Lastly, I would like to extend my thanks to the people at Huawei's Munich, Dresden, and Zurich Research Centers for their support, especially Sharan for teaching me about the intricacies of the Linux kernel, and Adamos and Siddharth for reviewing my code.

Abstract

In recent years, the use of Cloud Computing to process large amounts of data has increased. The data often contains sensitive information, for example, personal finances or healthcare records. Therefore the integrity and confidentiality of such data have become critical issues. This motivates cloud providers like Microsoft Azure and Google Cloud to provide confidential computing services that rely on Trusted Execution Environments (TEEs), e.g., Intel Secure Guard Extensions (SGX) and Trust Domain Extensions (TDX), AMD Secure Encrypted Virtualization (SEV) and Secure Nested Paging (SNP), and ARM Trust-Zone. These TEEs unfortunately contain several security vulnerabilities due to speculative execution, as such, Amazon Web Services (AWS) and Huawei Cloud have shown interest in alternative TEEs developing custom hardware: AWS Nitro and Huawei Qingtian enclaves.

However, many services that users want to run inside such enclaves, require modifications to their code. This is because Nitro-type enclaves only support Virtual Socket (VSock) communication with the outside. Meanwhile, legacy services typically use TCP for external communication.

In this thesis, we propose a solution to handle this issue by designing and implementing a secure and lightweight proxy, namely QProxy, that allows legacy services running inside the enclave to communicate with the outside without changing their code. To achieve this we create a TCP to VSock proxy that runs inside and outside the enclave to provide TCP connectivity to the legacy services. It is implemented in Rust, a memory-safe systems programming language. We evaluate QProxy using micro and macro benchmarks and our evaluation shows that QProxy significantly improves the latency and throughput of an application compared to previous work in Nitriding.

Finally, it is important to mention that QProxy is currently in use in production.

Contents

1 Introduction

Cloud Computing has grown as a market in recent years, resulting in an increased user base. This came with concerns over the security and privacy of the cloud as most Virtual Machines (VMs) are shared by multiple users and those that are not shared have no way to verify the systems' integrity and thus prevent leaking private information. A promising approach to address such concerns are Trusted Execution Environments (TEEs). These address privacy concerns by providing exactly what their name hints at; a trusted environment in which users can securely run their code and handle sensitive data.

A TEE provides additional guarantees on the way data is processed; through a combination of encryption and specialized hardware registers, a TEE ensures that the code being run has not been tampered with, will not be tampered with, and will only be read by the process inside the TEE. To achieve this the TEE uses a set of measurements regarding the application being run. This data is available to the application running inside the enclave through an attestation document which is signed by the TEE proving its validity.

A common type of TEE is an enclave which derives its security from the encryption of operating memory and some type of access control. They are widely used because they can act as wrappers over containers or VMs. Containers and VMs are heavily used for cloud deployments making it easy to transition an application deployed through a container to an enclave.

One industry example is Amazon Web Services (AWS), which provides its Nitro Enclaves service. These enclaves have a competitive advantage over other technologies like Intel Secure Guard Extensions (SGX)[7] and AMD Secure Encrypted Virtualization (SEV)[31], as they are not tied to the processor, and instead rely on an external module[32]. They are thus less susceptible to side-channel attacks[1] and can be replaced independently from the rest of the server hardware[26, 3].

Nitro-type enclaves do not have access to traditional network devices, instead, they rely on a Virtual Socket (VSock) connection to communicate with the outside world. VSocks are not supported by most applications and thus make it hard to run applications in an enclave. We will call these programs "Legacy Applications".

In this thesis we will work to support running "Legacy Applications" seamlessly inside the enclave by providing a VSock to TCP proxy called QProxy, this proxy will allow forwarding TCP packets to and from the enclave, thus allowing "Legacy Applications" that use TCP for communication to run normally inside the enclave.

[1] the module does not run arbitrary code like a CPU

1.1 Previous work

Running legacy applications inside an enclave has been explored by many previous works. We see both Enclaver[29] and Nitriding[36] propose different ways to run legacy applications inside enclaves. Their solutions go beyond just running legacy applications and tend towards a more complete solution for enclaves than just a simple shim between the enclave and the confidential application. We propose having the program run alongside the confidential application. This program should provide the necessary network connectivity while also being easy to audit and prove its trustworthiness to the clients.

1.2 Challenges

The main challenge we face is the need to abstract over a connection. A common way to achieve this is to use a proxy like Caddy[12] or Nginx[13]. They abstract over some type of connection, like HTTPS, presenting a simpler interface to the client application. We can provide a proxy over the VSock connection which exposes a normal network socket to the client interface. Additionally, the proxy should be easy to trust, by that we mean it should be simple to prove the proxy does not leak information, and by consequence, QProxy should be easy to audit. An enclave does not protect against processes running inside itself, and, because part of the proxy would be running inside the enclave, users need to trust the proxy.

We also find that properly supporting Remote Attestation is hard. Remote Attestation verifies that an application running inside an enclave is protected by the enclave's encryption (no man-in-the-middle or other type of attack has happened). We handle this by separating attestation from the proxy; meanwhile, Nitriding[36] relies on AWS KMS. We want to be flexible and support other enclave types besides AWS's Nitro enclaves and thus need to work without relying on AWS. By making attestation a separate process from the proxy, it can be customized for the enclave type and application requirements. We demonstrate the benefits of our design through example programs for AWS Nitro and Huawei Qingtian enclaves.

A further challenge is distributed computing. Many workloads like inference and training require distributed systems, but would also benefit from the security provided by enclaves. What becomes hard is the Remote Attestation and coordination of those systems; each machine would have to be individually attested and present proof to the other machines. Nitriding[36] solves this by supporting only homogeneous systems where every machine is the same (same Enclave Image File (EIF) and thus same Platform Configuration Register (PCR) values). This allows them to attest to each other by comparing PCR values, and to share secrets. However, this form of attestation causes problems with heterogeneous systems, and systems that need to be updated, as these types of systems cannot attest each other by comparing their measurements. Our choice of decoupling the remote attestation from the proxy makes testing and implementing different approaches much easier.

1.3 Overview

To create a proxy over a VSock in a trustworthy manner, we observe that working over a lower-level protocol, like TCP, simplifies the proxy implementation, and reduces the ways the proxy can maliciously alter traffic to the client application.

In this thesis, we design and implement **QProxy**, a small proxy for Nitro-type enclaves that allows users to run network applications over a VSock connection. It has the following advantages:

1. **QProxy is small:** QProxy is implemented in less than 1300 lines of Rust (of which only about 850 are code, the rest being either blank or comments). It is easy to audit and verify its trustworthiness.

2. **QProxy is composable:** to allow QProxy to run on any Nitro-type enclave. We make no assumptions about attestation. This gives the client application lots of flexibility regarding Key Management Service (KMS) and attestation and allows frameworks to make use of QProxy as a low-level primitive.

3. **QProxy has a low overhead:** the overhead of QProxy is acceptable for many applications allowing a simpler transition to enclaves compared to rewriting them to make use of VSocks.

4. **QProxy is transparent to the client application:** a client application can use the network as if it were outside the enclave. No major modifications are needed to run an application alongside QProxy.

QProxy delivers a simple way to migrate applications to Nitro-type enclaves that do not leave users locked to a specific cloud infrastructure. In addition, we demonstrate its performance by running Redis a popular in-memory key-value store.

2 Background

Our work relies on a good understanding of TEEs and related concepts. We will therefore go over the required understanding of TEEs in Section 2.1 and the related information about remote attestation in Section 2.2.

2.1 Trusted Execution Environments (TEEs)

A Trusted Execution Environment (TEE) is an environment that provides some guarantees about the environment, for example, the data being processed is not observable by unauthorized entities (e.g. the host operating system). Additionally, a program can verify that it is running inside the enclave.

A common set of guarantees provided by TEEs are confidentiality and integrity of data and code running inside itself. For example, an application with administrative privileges, like the operating system or hypervisor, has access to the program's memory. They may leak some confidential data, accidentally or maliciously, by reading a program's memory. A TEE may guarantee this cannot happen to applications running inside this environment. This guarantee is provided by hardware support, either through special processor instructions[1], or through specialized hardware[2]. Most TEE technologies provide a way to verify these guarantees through some form of Remote Attestation (see Section 2.2).

2.1.1 Intel Secure Guard Extensions (SGX)

Most modern Intel CPUs come with Secure Guard Extensions (SGX) support. These are special instructions that allow an application to enter a TEE. This type of TEE is usually called an Application Enclave, that is, a TEE at the application level. It requires both the application and the hardware to support this environment. Specifically, applications need to be modified to enable support for SGX, and users need to run those applications on an Intel CPU with support for SGX instructions[17].

The main mechanism through which the SGX TEE works is through protected regions of memories; when inside the TEE, a program may request private memory regions called enclaves. These regions will be decrypted on the fly, meaning the unencrypted data is only available within the CPU. This does mean that SGX enclaves are susceptible to side-channel attacks[30].

[1] some examples of this are ARM Trust Zone, Intel SGX and TDX, and AMD SEV and SNP
[2] AWS Nitro and Huawei Qingtian Enclaves

2.1.2 Intel TDX and AMD SEV-SNP

Some modern Intel and AMD server CPUs support TDX and Secure Encrypted Virtualization (SEV) instructions respectively. AMD CPUs may additionally support the more modern SNP instructions. These instructions allow for confidential VMs, which are different from application enclaves in that a full Virtual Machine (VM) runs inside the enclave instead of just part of an application. Users still need hardware that supports the extensions, but the application no longer needs to be aware that it is running inside an enclave.

2.1.3 AWS Nitro Enclaves

AWS Nitro Enclaves are a type of TEE, which leverages the hypervisor to provide isolation by creating the TEE as a separate VM, we call this TEE an "enclave VM" and the parent instance the "host VM". Nitro enclaves also use a Trusted Platform Module (TPM) to provide signatures for remote attestation[32]. All these features are provided by a kernel module loaded into a Nitro Enclave enabled EC2 instance. The Nitro Hypervisor is minimal and relies on Nitro Cards[3]. Side-channel attacks are prevented by never sharing Simultaneous Multi-threading enabled CPUs between instances, because Nitro Enclaves are also normal instances, the same protections apply between the host instance and the enclave. See Figure 2.1 for a diagram.

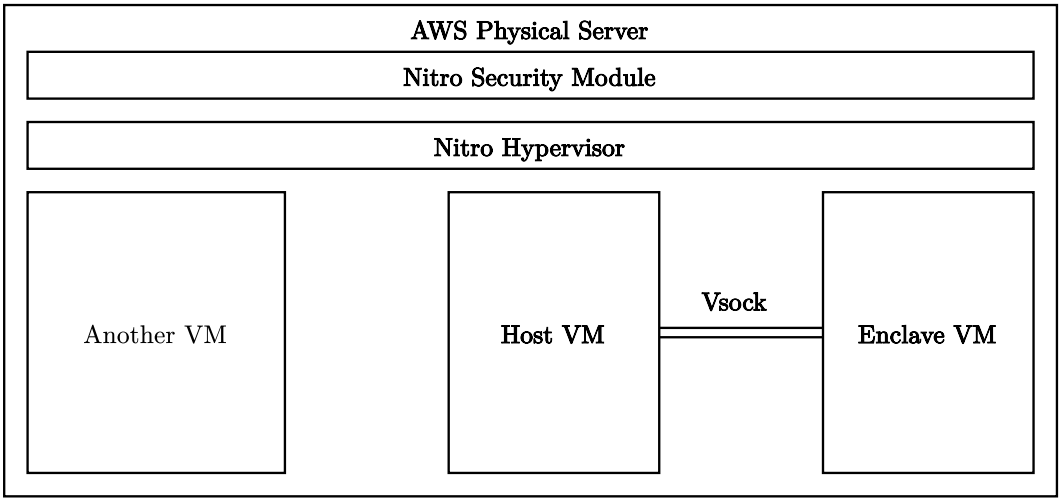


Figure 2.1: Diagram of an AWS Nitro Enclave. The physical server runs the Hypervisor under which the host VM is started. The host VM has a connection to the enclave VM. The hypervisor leverages the Nitro Security Module TPM.

Both VMs are connected through a VSock; when starting an enclave from within the host VM the hypervisor assigns a Context Identifier (CID) to the enclave VM. This CID

[3]hardware devices that provide system control and virtualized I/O independent of the CPU

can be used from within the host VM to connect to the enclave VM. The parent VM has a special CID that can be used to establish a connection from the enclave to the host. The CID is interpreted specially by the Hypervisor. This CID is not documented by Linux as it is specific to the Hypervisor implementation, but both AWS and Huawei Cloud use CID 3 as the host VM's CID ·

The enclave VM creates a TEE by requesting isolated CPU cores and memory from the Hypervisor[32] ensuring no information is leaked through CPU registers or by gaining access to the enclave's memory. Some enclave implementations go further by marking memory modified by a process outside the enclave as dirty thus preventing replay attacks.

2.2 Remote Attestation

Remote attestation provides a cryptographically verifiable document that proves the document was generated inside an enclave. The document is not standardized between different enclave types, but it usually includes the PCR values recorded when starting the enclave[4], and it can encrypt an optional nonce[5].

2.2.1 Attestation of AWS Nitro type enclaves

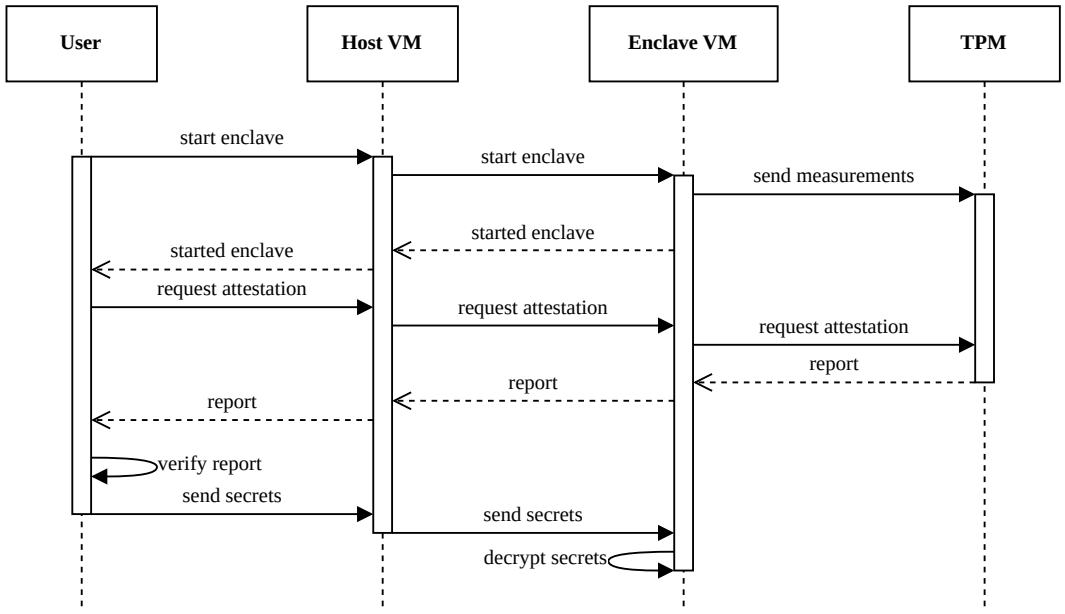


Figure 2.2: A sequence diagram of the attestation of enclaves.

In AWS Nitro enclaves, this process works as detailed in Figure 2.2; when the user starts

[4]used to verify the software running inside the enclave has not been modified
[5]a random value used only once, that helps prevent replay attacks

an enclave some measurements are stored in the PCRs[6] of the TPM. The measurements can then be requested as part of the attestation report. As part of the attestation request, the user should send a nonce that will be returned in the report. Once the user receives the report, it can be verified using AWS's Nitro Attestation Public Key Infrastructure. If the verification is successful, the user can now send the confidential data that should be processed by the enclave.

The Nitro Security Module signs the attestation document with its private key, which is built into the device. The signature can therefore only come from an enclave protected by this specific TPM. The manufacturer of the TPM provides a certificate that verifies the authenticity of any of the signatures produced by their devices and can thus be trusted.

2.2.2 Attestation of AMD SEV and Intel SGX/TDX enclaves

The Intel and AMD enclaves use keys built into their CPUs to sign the attestation report. This report can then be generated by calling an attestation function, which, in AMD's case will return a hash of the enclave's memory that can be compared with measurements taken in a trusted AMD SEV system[1]. Intel Trust Domain Extensions (TDX) similarly records the state of the Trust Domain as it is launched; this state can be measured in a trusted machine and then used to compare with the report generated inside the enclave[18]. In either, system the attestation report is cryptographically signed. The differences stem from the information contained in the report, and the way it is generated.

2.3 Huawei Qingtian enclaves

Huawei Qingtian enclaves are similar to AWS Nitro enclaves; they use specialized hardware[7] to provide cryptographic signatures to the attestation reports. They also integrate with Huawei's KMS and Identity and Access Management (IAM) services[27]. Beyond that, the API of Qingtian enclaves is similar to Nitro enclaves and attestation reports similarly contain the PCR values measured when launching the enclave.

[6] for example the hash of the EIF is stored in PCR0
[7] the QingTian Security Module (QTSM)

3 Overview

This section provides an overview of our objectives, the considered threat model, some challenges we face, and the system architecture.

3.1 Objectives

TEEs like Nitro and Qingtian enclaves require applications to communicate to the outside of the enclave through VSocks. However, legacy applications are not built with VSock communication in mind. Running such applications inside an enclave requires them to be modified, which is often prohibitively expensive. Thus our goal is to provide an abstraction to allow legacy applications to run inside the enclave unmodified.

3.1.1 Previous work on VSock proxies

To solve a similar problem, Nitriding[36] uses a TAP[21] device[1] to create a virtual network interface that forwards all TCP traffic outside the enclave. Enclaver[29] uses an HTTP/HTTPS proxy instead. Both provide an optional way to encrypt the HTTP traffic; Nitriding requests a TLS certificate from Let's Encrypt[2] on startup and Enclaver fetches them from AWS KMS.

We believe that providing TLS encryption is not something the proxy should do. The proxy could then leak the unencrypted data, either by accident (i.e. a dependency logs the payload), or maliciously; by terminating the TLS connection outside the enclave, as long as the code has not been reviewed, the enclave application will be none the wiser to the information leak.

This makes supply chain attacks much more viable; taking over one of the developers' accounts would then enable the attacker to make this modification[3], and a subsequent update of the proxy would include the malicious code. The increased complexity needed to handle TLS termination will also make auditing the proxy harder, reducing the likelihood of such audits, and thus the probability malicious or vulnerable code will be caught of such. We therefore advocate for a simpler proxy that does not handle the termination of encrypted traffic. This helps reduce the attack vectors and reduces the scope of the proxy.

[1] a device that allows managing packets in userspace

[2] a nonprofit certificate that provides SSL certificates on demand

[3] terminate the TLS connection outside the proxy

3.2 Threat Model

As detailed before, we are concerned with supply chain attacks on the proxy itself. Protecting enclave applications from such attacks is outside the scope of a proxy. We are also not concerned with side-channel attacks; AWS Nitro protects against them by offloading the hardware encryption to a device not part of the CPU package.

QProxy aims to protect against attacks on the VSock channel between the host and enclave VMs. This is done in two parts; as QProxy only handles raw TCP connections, it is not susceptible to attacks on the encryption of the TLS connection. It also provides resistance to supply chain attacks, as the information handled by QProxy is already public[4], a supply chain attack on QProxy cannot affect the integrity of the encryption on the packets beyond what any other application can.

3.3 Challenges

To provide a usable proxy we need to not only address the main problem of forwarding packets through the VSock connection but also ensure this is done with reasonable performance.

Firstly, we must ensure the application does not need to be modified to run alongside QProxy; a legacy application should work out of the box. Secondly, Qproxy should minimize the impact it has on an application running inside the enclave (see Chapter 6 for the performance evaluation). Thirdly, QProxy should work well in resource-constrained environments and scale up when given more resources (see Chapter 6).

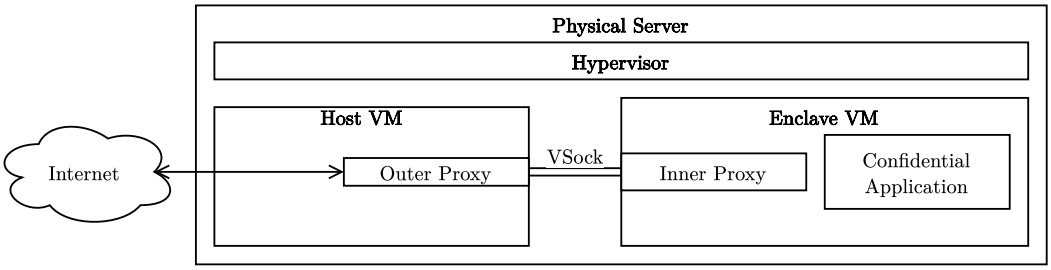


Figure 3.1: Diagram of a proxy over a VSock inside an enclave environment. The outer proxy is inside the host VM and the inner proxy is inside the enclave VM. The proxies are connected by a VSock connection. The confidential service and an attestation service are running inside the enclave. The outer proxy is connected to outside the server, represented by the internet in this figure.

[4] the packets are forwarded to the network, and are thus "exposed"

3.4 Architecture

Figure 3.1 shows that the basic components of QProxy are an inner proxy[5], and an outer proxy[6]. This stems from the need to convert from TCP to VSock connections between the enclave and the host.

The inner proxy forwards traffic to and from the VSock so that the client application has access to the host network. The outer proxy handles the forwarding between the network and the VSock connection.

As seen in Figure 3.1, QProxy runs on both sides of the enclave. The outer proxy is connected to both the VSock and the internet, while the inner proxy is connected to the VSock and provides network access to the confidential service and attestation service. This allows the confidential service to act as if it had a normal network connection, and ignore the fact that it is running inside an enclave.

[5] a proxy that runs inside the enclave

[6] a proxy running outside the enclave in the host VM

4 Design

In this chapter, we present our design goals and revisit the challenges presented in Chapter 1. To support legacy applications in Nitro-type enclaves we define our goal to be: developing a secure, transparent, and performant proxy. A secure proxy is resistant against Man in the Middle (MITM) and similar attacks, and will not leak the confidential data of the legacy application. A transparent proxy works without requiring changes to the confidential application running inside the enclave. A performant proxy requires few resources and has an acceptable impact on an application's latency and throughput.

As presented in Chapter 1 we need to proxy over a VSock connection to provide TCP connectivity to legacy applications running inside the enclave. This requires a process to be running on both the parent VM[1] and the enclave. Thus having both a trusted process[2] and an untrusted process[3]. This is described in Figure 4.1. Enclaves also impose a performance penalty on applications running inside them because of the integrity checks performed by the hardware. We do not want to add unnecessary overhead with our inner proxy either. The outer proxy is not as resource-constrained, but it is also desirable for it to use as few resources as possible.

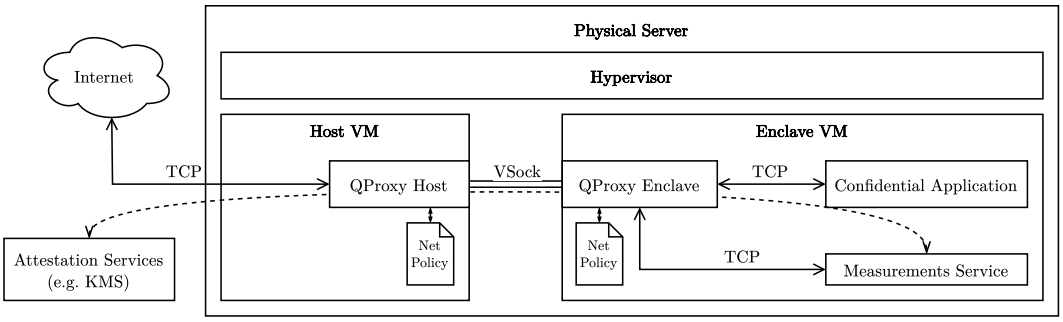


Figure 4.1: Enclave deployment using QProxy. It depicts how QProxy connects a confidential application running inside the enclave with the internet. It is also visible how QProxy facilitates remote attestation, and how a service written by the client may communicate with the service running inside the enclave.

[1] the VM that spawns the enclave
[2] the inside proxy which runs inside the enclave
[3] the outside proxy which runs outside the enclave

4.1 Design Goals

As stated previously, our goals are to provide a transparent, performant, and easy to use proxy. We will go over how to achieve these goals in the following sections.

4.1.1 Transparent Proxy

To achieve full transparency we need to expose all addresses inside the enclave to the outside and all external addresses to the enclave. This defeats the purpose of the enclave; we do not want everything to have access to the services inside the enclave nor do we want the services inside the enclave to have access to everything outside the enclave, we go over how we design this in Subsection 4.1.2.

The transparency we want to achieve is regarding the transport protocol; we want to take a TCP connection and forward it over a VSock. An example would be running an HTTP server with Nginx inside the enclave; the server would listen on ports 80 and 443 inside the enclave. The proxy would then have to listen on those same ports outside the enclave and forward them over the VSock connection. Then, inside the enclave, the proxy would listen on the VSock connection and forward the traffic to TCP ports 80 and 443 (the ports the confidential service is listening on).

To support accessing external services, like databases or a foreign API, then the proxy needs to forward requests to that service through the VSock. This can be done by intercepting DNS requests and redirecting those requests to the proxy address. For example, if an application inside the proxy needs to access *api.example.com* it can instead send that request to a specific port in the proxy (for example port 6000), which will forward the request to the outside proxy to deliver it to the actual address (*api.example.com*).

Alternatively, if the proxy is implemented on top of a TUN/TAP interface, users can directly forward the packets over the interface and let the host VM handle the DNS forwarding. This, however, makes it harder to filter connections as we will go over in the next section.

4.1.2 Access Control

As mentioned before in Subsection 4.1.1, we want some kind of access control regarding which addresses are exposed to the enclave, and which are reachable from within the enclave. We do not need a complex system like Cloudflare's Denial of Service (DOS) protections[4]; what we want is something rudimentary that ensures the client is in control of what data leaves the enclave. This limits the access control to just connections from inside the enclave to the outside.

We can achieve this by having a way to configure the proxy to only forward replies to external connections and connections to specific addresses. This comes naturally when the proxy forwards connections by opening TCP sockets, as the default is then to not forward anything. A connection then needs to explicitly request forwarding through some

[4] this requires profound knowledge in both network engineering and distributed systems

kind of configuration. When using a TAP/TUN interface the packets need to be filtered out instead. This requires a deny list instead of an allow list, which adds complexity as the proxy would need to inspect each packet.

4.1.3 Performance

Although it might seem counterintuitive, the cloud is a resource-constrained environment; as stated in many articles[14, 25, 34, 19] and corroborated by the growing interest in more efficient ways of managing cloud resources. It is also important to mention that the energy consumption of data centers is about 0.1% of the world's energy consumption[15, 16], which might not seem like much, but it is around 7.4% of the annual energy consumption of Germany. This is a significant amount of energy, so reducing the load on Cloud computers, or at least, ensuring the resources are used as efficiently as possible, is a small step we can take to reduce our impact on the world's climate.

To build a very performant proxy we need to consider the problem space; a proxy has to efficiently transfer data between sockets. There are multiple methods to do this, and some of them can be much faster than others. Marek Majkowski a Cloudflare engineer has done a thorough analysis of some options in a blog post[20]. There they analyze the performance of a simple echo server (a server proxying data between the read and write ends of a socket) using different proxy strategies; a blocking read write loop, leveraging the splice syscall, leveraging the io submit syscall, and using SOCKMAP through an eBPF program. Their results show that a simple read-write loop maxes out the hardware in their benchmark, but we have to take into account that they are using a now outdated Linux kernel (version 4.14) and there has been a lot of work on improving the eBPF VM[33]. This indicates to us that it might be worthwhile to re-evaluate their numbers using a more modern kernel, and maybe try some other kernel APIs like the relatively new io_uring [5, 6, 4] or take a look at eBPF again.

We also need to consider that we are expecting the enclave side of QProxy to run inside an enclave and the memory integrity checks of the enclave may affect the performance of these syscalls in ways we do not expect. We will do this analysis in Chapter 5.

4.1.4 Ease of Use

To make the proxy easy to use we want to support the common use cases with minimal configuration. We can achieve this by allowing for some configuration to be handled by the Command Line Interface (CLI), and some configuration to be handled automatically. For example, the common case of exposing a service to the outside without needing to access other external resources[5] could be handled directly in the CLI. Making the outer proxy a daemon allows it to be configured through the inner proxy through a command channel. This removes the need to configure the outside proxy but adds complexity to the inner proxy.

[5] like an in-memory database

4.2 Proxy Design

As stated before (see Figure 4.1), we need to split the proxy into a part that runs inside the enclave and a part that runs outside it. The requirements of the inner and outer proxies are different; the inner proxy will connect to a single outer proxy, while the outer proxy may connect to multiple inner proxies. The inner proxy is protected by the enclave while the outer proxy is not.

4.2.1 Inner Proxy

The inner proxy handles three types of communication; control messages (messages from the outer proxy), inbound communication (from the host to the enclave), and outbound communication (from the enclave to the host).

The control messages of QProxy consist only of heartbeat messages that diagnose connection issues. When a heartbeat message is not received in a certain amount of time, QProxy will assume the other side of the proxy has disconnected and will interrupt all incoming and outgoing traffic. Additionally, QProxy will not start proxying until a control channel with the outer proxy is established.

Inbound communication works like in a reverse proxy, except over the VSock. QProxy listens for connections and forwards them to the corresponding service inside the enclave. The service can then reply through the same channel, but it cannot initiate connections back to the other side.

Outbound communication is the exact opposite; QProxy listens in the enclave's network and forwards those connections to the VSock. This allows the enclave to connect to external resources like a database or a different enclave.

4.2.2 Outer Proxy

The outer proxy handles the same types of communication, except outside the enclave. This means that we have to be careful not to expose sensitive data to the environment. We sidestep this issue by forwarding TCP packets directly, thus requiring the service inside the enclave to encrypt the traffic before it leaves the secure side. This allows the encryption to be upgraded without requiring changes to QProxy.

Same as the inner proxy (see Subsection 4.2.1), the outer proxy sends and receives only heartbeat messages. Cutting off all communication to the enclave if the inner proxy is determined to be dead.

The inbound communication is the same; the source is the host network, and the destination is the enclave's VSock. Outbound communication is much of the same; the proxy listens on the enclave's VSock and forwards to the host network. Additionally, the proxy may perform DNS lookups if it needs to resolve a URL (like api.example.com).

4.2.3 Comparison with other designs

Unlike Nitriding[36] and Enclaver[29], we do not perform TLS termination on the inner proxy. This increases the complexity of the proxy and we believe it provides little value as legacy applications tend to have a preferred way to do TLS termination (e.g. Nginx). Both of which see active development and have robust TLS implementations. Duplication of this work while making the proxy a security-sensitive product does not improve the security of the enclave.

We also do not provide attestation facilities like Nitriding[36]. Attestation is very much linked to the KMS service in use, and because QProxy is cloud provider agnostic, we do not want to add support for all cloud providers as part of QProxy. We go into more depth in Section 4.3.

QProxy supports any enclave as long as it provides a VSock interface. Nitriding[36] only supports AWS Nitro. Although Enclaver[29] plans to support Azure Confidential VMs and other technologies, it currently only supports AWS Nitro.

We provide a lighter environment (just a proxy), but that gives us wider support and easier adoption of new platforms. We propose that separating the concerns of attestation and proxying improves the security and ease of implementation of the enclave ecosystem.

4.3 Attestation

Attestation is handled by a separate program. Examples will be available alongside QProxy's code and can be used for testing. It is recommended to adapt the examples to the service being run; for example either starting QProxy or the confidential service service only after verifying the attestation document. This way, the service is never exposed to the network before the enclave is verified.

The attestation code is separate from QProxy as it needs to fulfill requirements that conflict with QProxy's mission: it needs a substantial amount of code specific to the enclave technology used, depending on the type of enclave some code may be dead (never used), and it needs to produce TLS messages.

Implementing this into the proxy would incur additional complexity without much benefit to the customer: the proxy does not currently link to any SDKs and is therefore "platform independent". All of the code is "live", thus having a leaner attack surface. It also does not care about TLS, which means it does not handle any of the security of the confidential application, therefore being unaffected by critical security vulnerabilities to the encryption software.

5 Implementation

To implement QProxy we have the following requirements:

1. Secure Programming Language: Many problems should be prevented or caught before running the program (at compile time).

2. High Performance: QProxy should not interfere with the client's application, this means it should not interfere with the latency or throughput of the application.

3. Auditable: Ideally, it should be easy to audit QProxy, and verify its claims.

The first requirement limits us to memory-safe language, hopefully with a great type system that can represent problems with types and thus catch many issues at compile time. Languages which fit this requirement are memory-safe and statically typed programming languages. This mostly limits us to garbage-collected languages with static types, as those prevent most memory issues like double frees and use after frees by using a garbage collector.

The second requirement steers us towards compiled languages and systems programming languages that do not abstract much over the hardware. We will be working with a lot of system interfaces and thus good compatibility with C code will be especially helpful.

The third requirement means that we should keep QProxy's codebase small and straightforward, ideally relying on well-known and well-audited dependencies. This makes the burden of auditing QProxy smaller and increases trust, allowing more customers to use it.

5.1 Programming Language Choice

Rust seems to fit these design requirements perfectly: it is a memory-safe language whose type system is heavily influenced by ML languages like OCaml and Haskell. This gives library authors many tools to provide APIs that cannot be misused by performing compile time checks. The Rust community embraces this design and has shown its commitment to safety by creating amazing libraries like Serde, which provides a generic and type-safe way to serialize and deserialize data, without any knowledge of the data being serialized, nor the serialization format being used.

Another way in which Rust excels is that it is a Systems Programming Language, providing low-level access to system resources, providing performance comparable to the

fastest programming languages, and a high degree of control over the performance characteristics of the program.

It is harder to audit a Rust program compared to other high-level languages, as it requires more code than dynamic languages like Python or Ruby, and there are fewer experts compared to other low-level languages like C++. Rust does give us access to a thriving ecosystem of open-source libraries, which means that we do not need to write as much code as we would have to in C or C++.

The previous reasons steered our choice of Rust as the implementation language of QProxy.

5.2 Proxy Approach

In Chapter 4 we mention multiple methods to create a VSock proxy. We decided to directly proxy the connections without using newer kernel APIs like io_uring or eBPF, or by using older APIs like splice. This is because the necessary packages to build a naive VSock proxy were available, but not to create a proxy over io_uring or splice, this is especially challenging in Rust, as such APIs require the use of "unsafe" Rust; a non-memory-safe superset of Rust. This defeats or purpose of making the proxy easy to audit, by relying on external dependencies we delegate the auditing of unsafe Rust to the whole Rust community instead of only those who use QProxy. Evaluating these options and generating memory-safe code that makes use of them is outside the scope of this thesis.

5.3 External Dependencies

We rely on some external, well-tested, Open Source dependencies (commonly called crates in the Rust community) to avoid reimplementing functionality in a potentially under-tested or even unsafe way. Unsafe Rust does not guarantee memory safety at compile time. For an unsafe Rust program to be memory-safe many rules have to be followed. Ensuring all these rules are followed properly is hard for both the programmer to do and someone auditing the code to verify. Relying on external dependencies allows us to get the benefits of unsafe Rust code[1] while using well-tested code.

5.3.1 Asynchronous Runtime

Rust does not have a default asynchronous runtime, but we need to process the requests asynchronously; we expect most of the CPU time spent by the proxy to be either copying buffers or waiting around for I/O. This steers us towards an asynchronous program.

We decided to use the tokio-rs[2] runtime, but another option would have been the smol[3] runtime. We decided against it because it does not have a VSock crate, so we would have

[1] e.g. direct access to the operating system's APIs
[2] https://crates.io/crates/tokio
[3] https://crates.io/crates/smol

had to build an asynchronous wrapper interface around the Rust bindings for the VSock interface or use a compatibility layer with tokio-vsock[4]. Smol does allow users to build specialized asynchronous executors so if we needed the extra flexibility or performance from a specialized executor, we would have opted for it instead. Although, as we show in Chapter 6, the performance of tokio-rs is enough to achieve our goals.

Tokio-rs is focused on network applications and IO-bound applications. Tokio-rs is also the most popular async runtime for a reason: it is fast, reliable, and most importantly; it takes security issues seriously by publishing patches and Common Vulnerabilities and Exposuress (CVEs) promptly in response to any security issue.

5.3.2 Logging

To debug our application and provide extra feedback to the user when something is happening we generate logs. Having a logger that is aware of the asynchronous nature of our program is very helpful. This is the reason we opted for tracing[5] instead of the standard log[6] crate. Tracing is developed and recommended by the tokio-rs developers.

5.3.3 Additional Dependencies

We use serde[7] to have a type-safe way of defining and reading the configuration file. Toml[8] for TOML[9] file support for serde. Color-eyre[10] as a way to pretty-print fatal errors. And clap[11] to parse command line arguments to the program.

5.4 Proxy

The proxy works by first parsing the command line options with clap, then using serde to read the configuration file. These sources provide information on what connections the proxy should open, and in which direction they operate. We then proceed to start the asynchronous runtime so that we can start establishing the connections.

The first connection we establish is the control connection through which we start sending and receiving heartbeat messages. We use the control connection to diagnose the state of the enclave/host proxy and cut off connections when one side goes offline. This reduces unnecessary traffic and logs.

Once the control connection has been established we start all the proxy tunnels. Each consists of a source and a target, the source is where we will listen for new connections, and the target is where we will forward the packets from those new connections.

[4] https://crates.io/crates/tokio-vsock
[5] https://crates.io/crates/tracing
[6] https://crates.io/crates/log
[7] https://crates.io/crates/serde
[8] https://crates.io/crates/toml
[9] the configuration language we use for QProxy's configuration
[10] https://crates.io/crates/color-eyre
[11] https://crates.io/crates/clap

To forward data between the source and the target, we read from both sockets and copy the read data between the source and the target. Once both the source and the target close the connection, we close both sockets.

If one of the proxies stops running[12], the other will close all active connections, stopping the transfer of data, and resume trying to connect to the proxy. Once the proxy is back up all the proxy tunnels will be reestablished.

5.5 Remote Attestation

We determined the complexity of Remote Attestation to be outside the scope of a proxy. This allows us to easily share the proxy between multiple different enclave technologies[13]. The remote attestation component can be written as a separate service that runs inside the enclave, this service depends on the enclave technology but is isolated from the proxy.

Reducing the complexity of the proxy makes it easier to audit the code and ensure no security vulnerabilities enter the codebase. It also isolates the more sensitive attestation service from the proxy.

Our example attestation service requests an attestation document from the enclave's TPM device and sends it over the VSock connection. Users can then verify the document with our provided tool to ensure the integrity of the enclave.

It also means that QProxy is completely independent of the Qingtian SDK and will work over any Vsock connection, but it is specialized in the use case of enclaves, and might not suit other applications.

5.6 Other Tools and Contributions

We implemented a tool to forward the stdin, stdout, and stderr of a process over a VSock connection. This is a simple way to get a debug interface into the enclave. This tool also works for forwarding logs of applications running inside the enclave to the outside. We modified the oha[11] HTTP benchmarking tool to support VSock connections and sent a patch[23] which has since been merged and included in version v1.4.0. We have also contributed to tokio-vsock[24] with multiple API fixes.

[12] for example if we shut down the enclave

[13] AWS and Huawei enclaves at the moment

6 Evaluation

One of the important design goals of a proxy intended to run alongside an application is that it should not bottleneck said application. It is unavoidable to increase the latency of the application when running through the proxy, but ideally, this impact should be minimal and thus sufficient for most applications.

If the impact of QProxy is too big, the application itself might need to be rewritten to use VSocks instead of TCP Sockets. Most applications and libraries are not made with VSocks in mind and thus support will need to be patched in. Maintaining these patch sets and keeping them up to date is hard and thus not realistic. We would expect them to instead stick to a version that might not receive security updates compromising the data inside the enclave.

6.1 Goals

As we detail in Chapter 4, our goals are: transparently exposing networked processes inside the enclave to the host machine/internet, having some rudimentary access control, reducing our impact on the confidential service, and ease of use. We detail how this can be achieved in Chapter 4 and how we hope to achieve it in Chapter 5. We evaluate these goals in this chapter:

We achieve transparency by running Redis unmodified inside both Nitro and Qingtian enclaves, and accessing it through memtier_benchmark from the host VM.

Access control is achieved by only exposing the services detailed in QProxy's configuration file which is inside the enclave (and thus protected from host processes).

Ease of use , QProxy is a single binary, unlike Nitriding[36][1], which makes it arguably harder to deploy. On the other hand, we do not provide automatic HTTPS or KMS integration for the reasons detailed in Chapter 4.

QProxy's performance by doing a series of benchmarks detailed in the following sections. We compare against Nitriding[36] and a VSock implementation (when applicable).

[1] Nitriding requires gvproxy to be running in the host VM and gvproxy[10] needs to be notified of which ports to expose by a separate process (curl in Nitriding's examples).

6.2 Testing Methodology

To test the performance of QProxy we run a custom benchmark suite; first, we do a series of synthetic benchmarks, and then we show results on a real-world application.

6.2.1 Synthetic Benchmarks

The benchmarks are run through a bash script and all packages are taken from nixpkgs[2] except for iperf, oha, and memtier_benchmark, iperf because we use a fork, and the other two because we use unreleased (at the time of testing) versions to receive additional functionality. We explain the reasoning for this further below along with how we run the benchmarks.

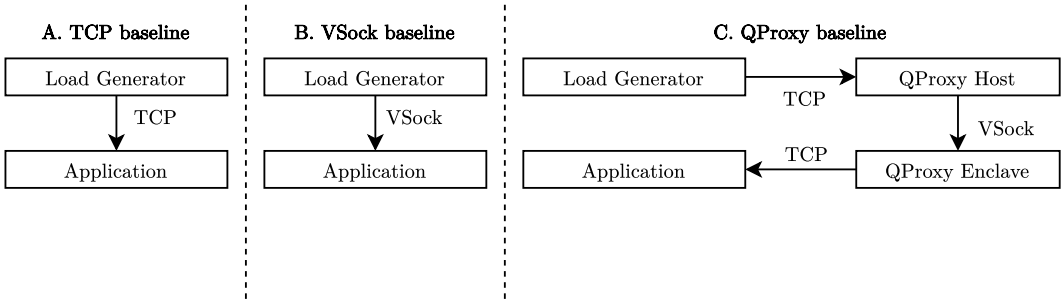


Figure 6.1: Diagram detailing how we apply a load to the tested application when we run benchmarks on the loopback device.

As a baseline for the performance, we run benchmarks on the loopback device outside the enclave. We run three types of benchmarks locally; a baseline for expected performance using the normal TCP transport of the application, a baseline for VSock performance[3], and a baseline for QProxy performance. The load generator loads the application directly, or through QProxy as detailed in Figure 6.1. We expect QProxy to perform worse than the VSock loopback benchmark (about half to a fourth of the throughput) because it will run both the inner proxy and the outer proxy in the same VM, halving the available resources for the benchmark, if the results go under that, then QProxy is not performing well enough.

When running the benchmarks inside the enclave, the load generator and the host side of QProxy both stay outside the enclave. We also run a benchmark without QProxy if the load generator and the service support VSock connections as detailed in Figure 6.2. Here throughput should be about half of what can be achieved with the VSock connection as the resources are split between the inner proxy and the confidential application. When running inside the Nitro enclaves we will also test Nitriding[36] as they only advertise Nitro support.

[2] nixos-unstable branch, commit 9099616b93301d5cf84274b184a3a5ec69e94e08
[3] not available for memtier_benchmark

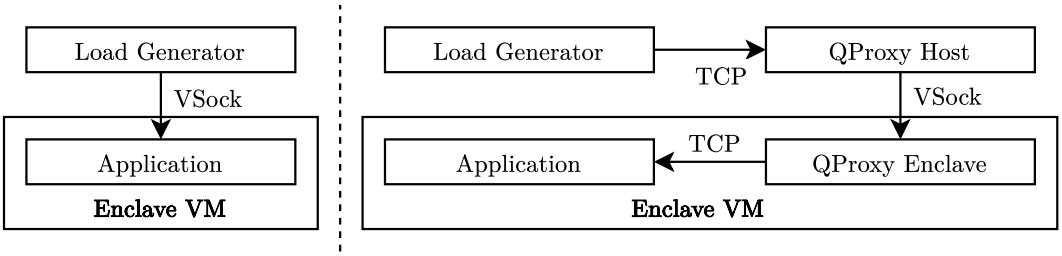


Figure 6.2: Diagram detailing how we apply a load to the tested application when we run benchmarks inside the enclave.

We run an iperf3[9] fork with patches for VSock support to measure the throughput of the proxy and the underlying VSock channel. Iperf3 self describes as a network bandwidth measurement tool. It has a server and a client; the server awaits client connections, then it either sends or receives data from the client, measuring how much data it can receive in a limited amount of time.

We also ran an unreleased version of oha[11] which has our patches for VSock support applied. Oha contains the patches since version v1.4.0. Oha is an HTTP load generator that can be used to test the performance of HTTP servers. It does not come with a test server, so we wrote a small HTTP server using hyper[22], an HTTP library for Rust. We used hyper, not because of its promised speed, but because it is the highest level HTTP library we could find that allowed us to use VSocks for the HTTP transport. This simple HTTP server works either over TCP or VSock and responds only to HTTP 1.1 requests. It always replies with "<h1>Hello, World!</h1>". This allows us to measure the impact of QProxy in one of the worst cases; a static, small response, that maximally loads QProxy. We used the default options in oha, except we ran the benchmark for five minutes. This means, that oha established 50 connections to the enclave and used all available threads to produce the highest load QProxy and the HTTP server could handle.

Lastly, we test the Redis in-memory key-value store with the memtier_benchmark to have an example of something closer to a real-world load on a production application. Redis does not support VSock connections so we only benchmark QProxy and Nitriding.

The tests are run for five minutes over the loopback device or the enclave, with and without QProxy or Nitriding (if available). We also measure against TCP to see if the VSock channel itself might be a bottleneck.

6.3 Huawei Qingtian Enclave Performance

We use a c7.8xlarge.4 Huawei EC2 instance which provides us 32vCPUs and 128GiB of RAM; we allocate 1GiB and 2vCPUs to the enclave during the tests unless otherwise specified. We limit the resources to the enclave because that way the results are comparable to those of the Nitro enclave.

6.3.1 Latency Benchmarks

We first benchmarked QProxy's latency using oha in the Qingtian enclave, (see Figure 6.3). The table shows the interface over which the traffic was directed, be that the host kernel's loopback device, or the connection between the host and the enclave. It also shows the benchmarked application; TCP is used as a baseline performance metric to compare against the VSock application, this shows us the difference between the TCP and VSock network stack. QProxy can then be compared with the VSock benchmark to see the benefits of circumventing QProxy and rewriting the application to make use of VSock connections.

We plot the results in Figure 6.3, which records the average latency, the 95th, and 99th latency percentiles (p95 and p99 respectively) of the distribution.

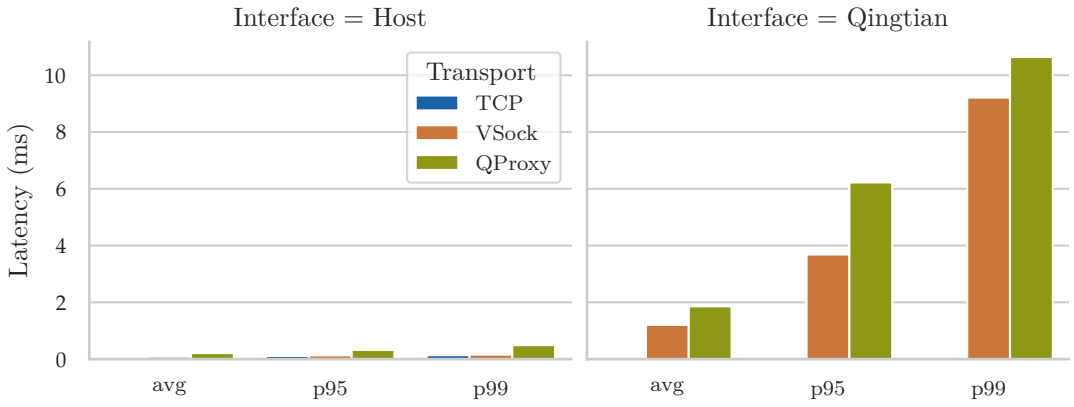
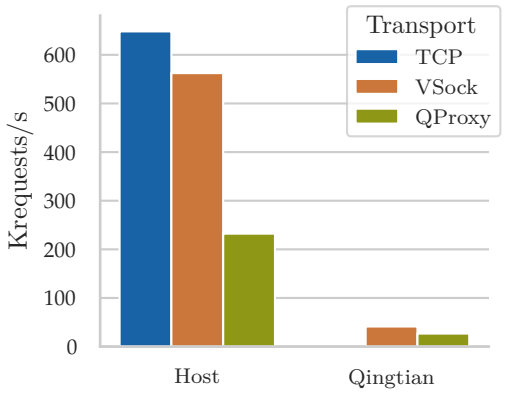


Figure 6.3: Average latency, 95th percentile, and 99th percentile of an HTTP server as measured by oha (deployed on a Huawei EC2 server).

The results on the loopback interface show a clear impact when using QProxy, but almost no difference between VSock and TCP transport. The latency we observe on the host instance is well under a millisecond. As a comparison, the latency between two data centers in the same region is roughly between 5 and 30 milliseconds[2].

Once we use the enclave, we see the VSock's average latency jump up to a little over a millisecond and QProxy's to just under two milliseconds, the 95th and 99th percentiles show a drastic jump, this may be because of the reduced number of cores, or because of the enclave's overhead itself. One thing to point out is that we see QProxy follow the VSock performance closely, always within one or two milliseconds of the VSock performance.

These measurements are not a good comparison though as the load was uneven (see Figure 6.4). Oha tries to load the application maximally and that means that although these numbers are great for an analysis under maximum load, they do not represent the real latency impact QProxy has on the confidential application. We will limit the requests per second that oha emits to test all configurations with an even load. The limit we chose was 10k requests per second as this gave us a good 60% gap from the maximally loaded



interface	benchmark	rps
loopback	TCP	648.2krps
	VSock	562.4krps
	QProxy	232.3krps
enclave	VSock	41.2krps
	QProxy	26.8krps

Figure 6.4 & Table 6.1: HTTP server throughput as measured by oha (deployed on a Huawei EC2 server).

scenario (see Table 6.1).

Analyzing the HTTP throughput we see a 15% impact on the rps when using VSock instead of TCP transport. QProxy is limited by the VSock performance but achieves only 41% of its performance. This is expected as now the host VM's resources are being shared among oha, the HTTP server, and two instances of QProxy (the host and the enclave side). Inside the enclave, the resources are shared between the HTTP server and QProxy, so a performance drop (about 65% of the VSock transport) is expected.

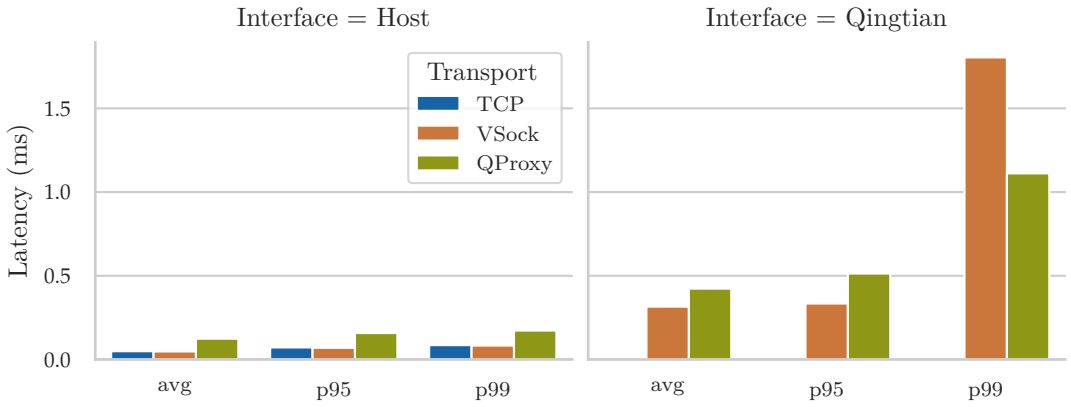


Figure 6.5: Average latency, 95th percentile, and 99th percentile of an HTTP server as measured by oha when limiting the load to 10krps (deployed on a Huawei EC2 server).

When the load is limited to 10krps we observe much better latency; the average and the 95th percentile are hovering around half a millisecond, and the 99th percentile is between 1.7 and 1 millisecond. QProxy appears to have stabilized the load in such a way the 99th percentile of requests have lower latency than those going directly through the VSock

transport, this could be attributed to outliers.

6.3.2 Throughput Benchmarks

Unlike QProxy and oha, the throughput benchmarking tool we use, iperf, runs on a single thread, we therefore do not expect the reduction of cores inside the enclave to affect its measurements. We use this to measure the bandwidth of the different interfaces and how much of it QProxy achieves. Iperf has a client and a server, we run the server inside the enclave (if relevant) and the client from the host.

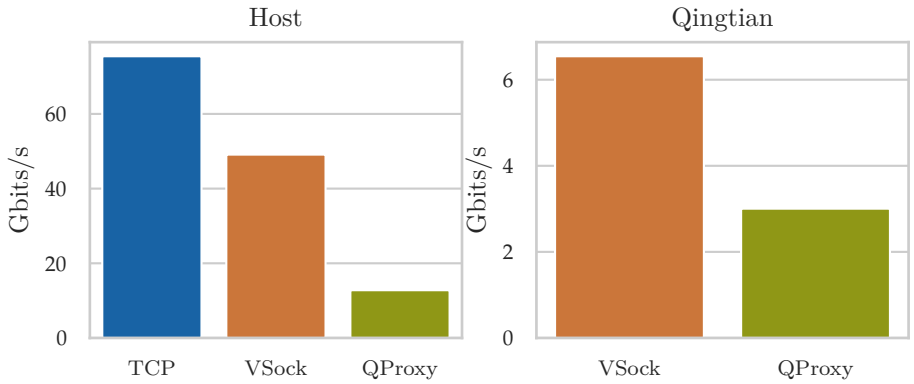


Figure 6.6: Huawei EC2 server's interfaces throughput as measured by iperf ($Gbit/s = 10^9 bit/s$).

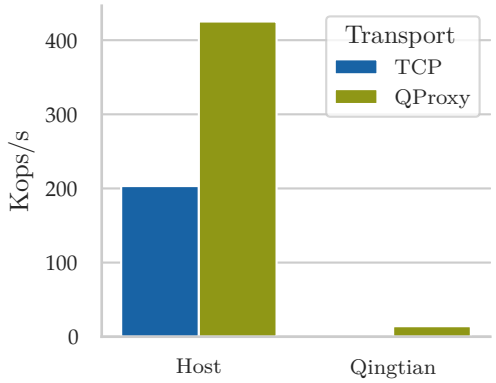
We observe a big difference in performance between TCP and VSock transport; VSock transport only reaches about 66% of the throughput that the TCP transport can carry. This does not cause a problem with the rps when testing with oha (see Table 6.1), because the requests oha sends are much smaller (a few hundred bytes) than those sent by iperf (128KiB in our configuration). QProxy is not optimized for throughput; the way it works causes many buffers to be copied (see Chapter 5).

QProxy reaches about half the throughput that the VSock transport allows to the enclave. We must highlight that the throughput to the enclave is severely limited: it is barely above 10% of the throughput of the loopback interface at just above 6Gbit/s. The result QProxy achieves is nonetheless good as the resources inside the enclave are being shared between QProxy and iperf, we expect QProxy's throughput to improve as more cores are allocated to the enclave.

6.3.3 Redis Benchmarks

Until now, we have only measured QProxy against synthetic benchmarks. This is useful to test the limits of what can be achieved through QProxy, but the loads we have generated are fully artificial; they are designed to show how QProxy behaves under extreme circumstances. A more realistic load is the in-memory key-value store Redis. We use memtier_benchmark to generate a load on Redis and measure the latency (average,

95th percentile, and 99th percentile), and the number of operations performed per second (ops/s).



interface	benchmark	ops/s
loopback	TCP	203.2kops/s
	QProxy	425.2kops/s
enclave	QProxy	14.2kops/s

Figure 6.7 & Table 6.2: Redis' throughput as measured by memtier_benchmark (deployed on a Huawei EC2 server).

Again, we cannot compare the host results to the enclave results as the environments are completely different. For some unknown reason, performance improves when QProxy is used (see Figure 6.7). This is unexpected, but we can trace this to a higher hit rate in the cache.

We do not know if this is caused by memtier_benchmark or our environment, but this does show that QProxy can handle high loads inside the host machine. The load handled inside the enclave is only 14kops/s (see Table 6.2), but we do not have a baseline to compare against as Redis does not support VSock transport.

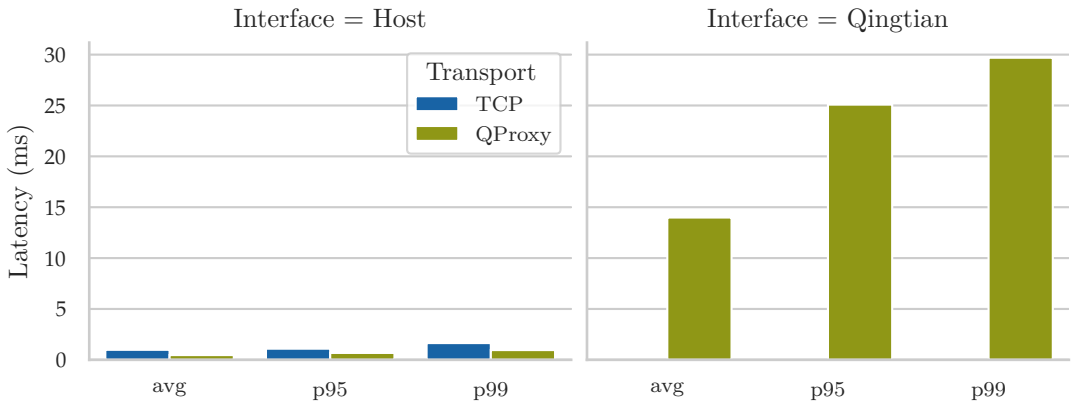


Figure 6.8: Average latency, 95th percentile, and 99th percentile of Redis as measured by memtier_benchmark (deployed on a Huawei EC2 server).

Assuming it scales similarly to the HTTP benchmark, we would expect the throughput to be much lower (around 8.4krps). The most likely cause is that Redis is being

bottlenecked by the I/O or memory throughput inside the host VM, while the enclave presents a different bottleneck. We can see this from the latency number in Figure 6.8, the latency under load is unacceptable; it is on average almost 15 milliseconds and 5% of the operations have more than 25 milliseconds of latency.

To address this bottleneck, we reduced the load on Redis by setting memtier_benchmark to only produce 10kops/s (to do this we used a pre-release version of memtier_benchmark, as the current release[4] does not support rate limiting the operations[28]).

When rate-limiting memtier_benchmark to 10kops/s, the average latency drops to acceptable levels (see Figure 6.9). The 5% and 1% worst requests are still high at 10 and 15 milliseconds respectively, but we cannot attribute the latency increase to QProxy or Redis, just from these results.

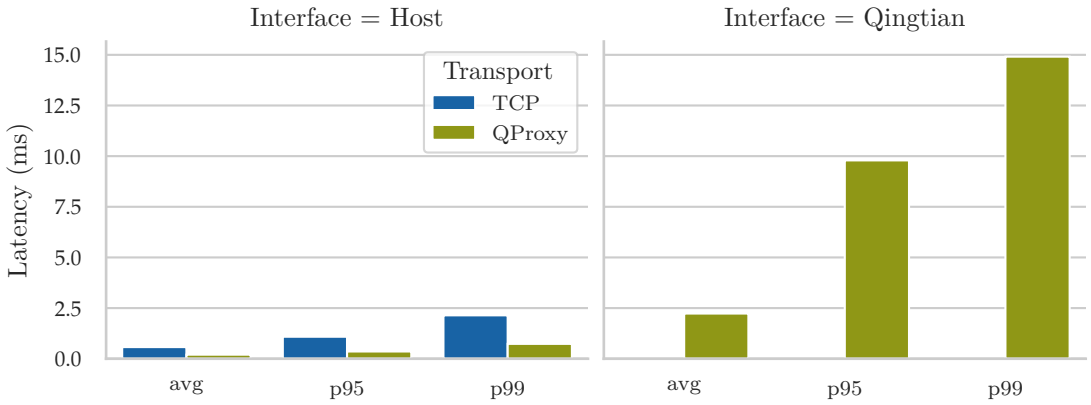


Figure 6.9: Average latency, 95th percentile, and 99th percentile of Redis as measured by memtier_benchmark when limiting the load to 10kops/s (deployed on a Huawei EC2 server).

6.4 AWS Nitro Enclave Performance

To compare against Nitriding[36], we also run our benchmark suite in an AWS Nitro EC2 server, specifically a c6.8xlarge[5] server with 32vCPUs and 64GiB of RAM. Unless otherwise specified, we isolate 2vCPUs and 1536MiB[6] of RAM for the enclave.

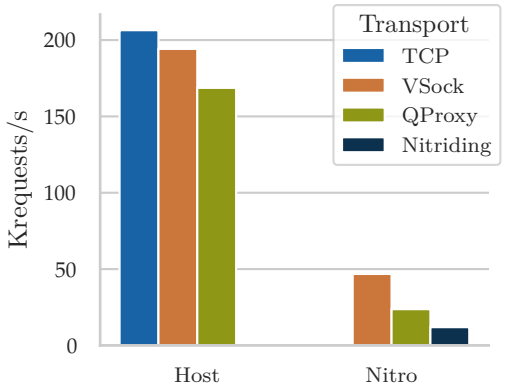
Although both EC2 instances (AWS's and Huawei's) use Intel CPUs of the same generation they have subtly different models: AWS uses an Intel® Xeon® Platinum 8375C CPU @ 2.90GHz and Huawei uses an Intel® Xeon® Platinum 8378A CPU @ 3.00GHz. Huawei Cloud's CPU has a slightly higher base clock than AWS's and a slightly smaller

[4] memtier_benchmark v2.0.0

[5] We tried using a c7i.8xlarge server, but it did not have support for Nitro enclaves.

[6] Nitro enclaves have a higher minimum memory requirement than Qingtian enclaves and our Redis image did not fit in the 1024MiB originally assigned.

L3 cache, but their boost clocks are the same. We expect the results to be fairly similar outside differences between Nitro and Qingtian enclaves.



interface	benchmark	rps
loopback	TCP	206.5krps
	VSock	194.2krps
	QProxy	168.7krps
enclave	VSock	46.8krps
	QProxy	23.8krps
	Nitriding	12.1krps

Figure 6.10 & Table 6.3: HTTP server throughput as measured by oha (deployed on an AWS EC2 server).

6.4.1 Latency Benchmarks

The unrestricted results are in Figure 6.11, and the requests per second achieved by oha in Figure 6.10

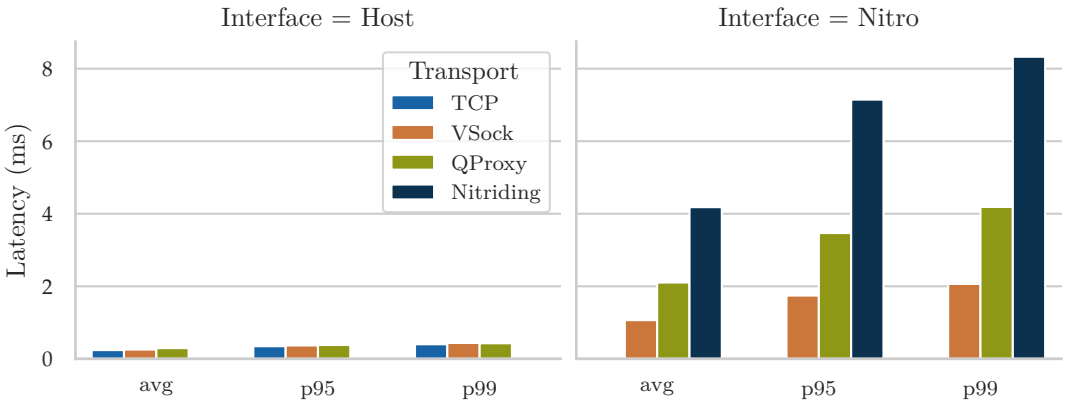


Figure 6.11: Average latency, 95th percentile, and 99th percentile of an HTTP server as measured by oha (deployed on an AWS EC2 server).

AWS's EC2 instance HTTP throughput is much lower than what we observe in Huawei's EC2 instance. We do not believe that the instance was under memory pressure, so we can only theorize that either the different software stack (kernel version and compile options) or the different hardware, played a role in the difference we see in Figure 6.10. The Nitro enclave's throughput is a closer match to the Qingtian enclave's; again QProxy achieves

about half the throughput of VSock transport, while Nitriding achieves about half the throughput of QProxy (see Table 6.3).

The host VM's latency was again negligible (under a millisecond) with QProxy closely matching the VSock and TCP latencies. Inside the enclave, we measured an average latency of one millisecond with 5% peaks of almost 2 milliseconds and 1% peaks of over two milliseconds through the VSock transport. QProxy's latency is circa double the VSock's, while Nitriding's latency is circa double QProxy's.

As stated before, because we measure under a high-load scenario, the latency is higher than under a reduced load. We can limit the load on the server to obtain better latency measurements. Conveniently, we can again use 10krps as a baseline load as all of the tested configurations can achieve more than that (although Nitriding is close with only 12krps).

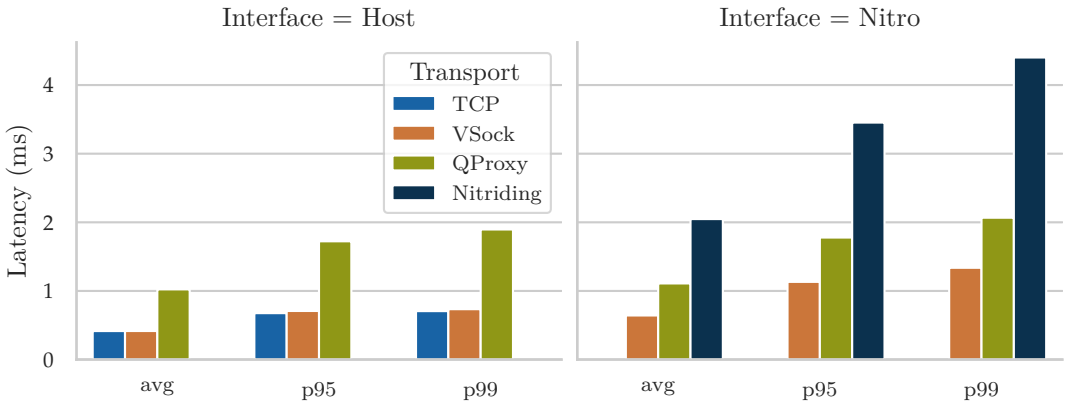


Figure 6.12: Average latency, 95th percentile, and 99th percentile of an HTTP server as measured by oha when the load is limited to 10kops/s (deployed on an AWS EC2 server).

The results we obtain under these conditions are less spread out, but Nitriding still exhibits worse performance having almost twice the latency of QProxy on all measurements (see Figure 6.12). There is less of a gap between QProxy and the baseline VSock transport, however (only about half a millisecond more). And we see the gap between the enclave's and the loopback interface's VSock latency to be within half a millisecond.

6.4.2 Throughput Benchmarks

The throughput results in Figure 6.13 are substantially lower than the Huawei measurements in Subsection 6.3.1, both in the host, and the enclave. The enclave results can be attributed to the difference between Nitro and Qingtian enclave's performance, but the loopback interface results can only be due to differences in the CPU or software, as iperf is the same between both benchmarks, this must be due to differences in the network stack or due to differences in the CPUs.

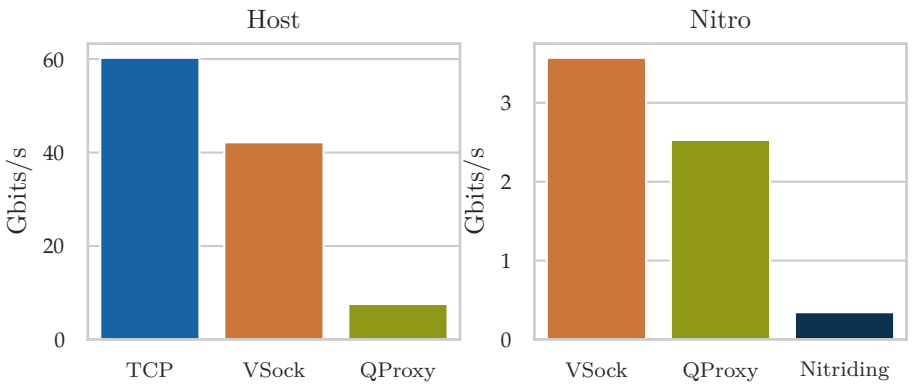
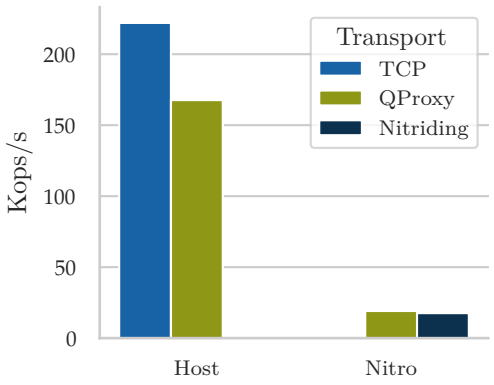


Figure 6.13: AWS EC2 server's throughput as measured by iperf ($Gbit/s = 10^9 bit/s$).

QProxy achieves a competitive throughput, reaching about 70% of the native VSock transport's throughput. The absolute value is lower than the one achieved within the Qingtian enclave, but it is a higher percentage of the maximum throughput we reach. We could reproduce the values measured by Nitriding[36] and we match the expected value from host to enclave, we do not display the reverse (from enclave to host), as we measured no difference in throughput in all benchmarks but Nitriding's. Nitriding achieves higher throughput in that case (about 1Gbit/s) which is less than half of QProxy's throughput.

6.4.3 Redis Benchmarks



interface	benchmark	ops/s
loopback	TCP	222.0kops/s
	QProxy	167.7kops/s
enclave	QProxy	19.1kops/s
	Nitriding	17.6kops/s

Figure 6.14 & Table 6.4: Redis throughput as measured by memtier_benchmark (deployed on an AWS EC2 server).

Our stress tests have shown a big gap between QProxy and Nitriding. QProxy's HTTP throughput is about double that of Nitriding's, the latency is halved, and the interface's throughput is doubled. This is not the case with Redis; as seen in Figure 6.14, Nitriding closes the gap with QProxy achieving 92% of its throughput in terms of operations per second (see Table 6.4). Redis throughput is slightly higher in AWS's EC2 instance

compared to Huawei's, being between 7% and 36% better.

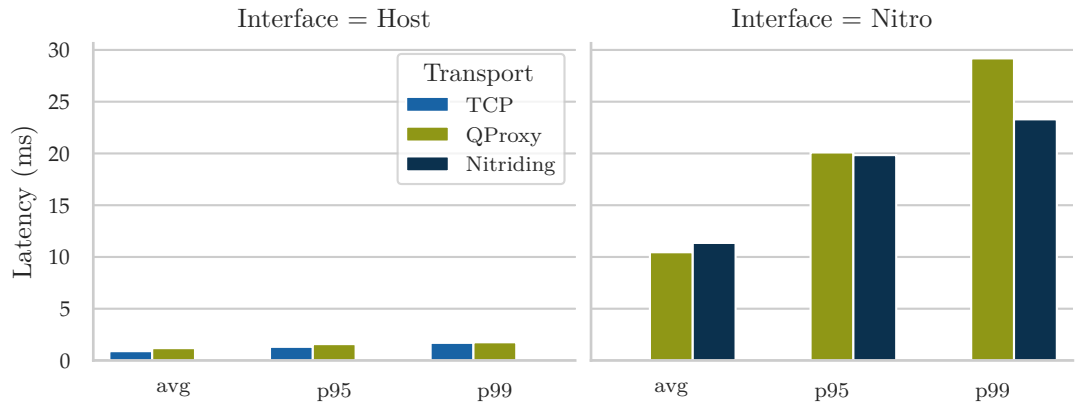


Figure 6.15: Average latency, 95th percentile, and 99th percentile of Redis latency as measured by memtier_benchmark (deployed on an AWS EC2 server).

The latency experienced at this load is slightly lower than in Huawei's EC2 server, and Nitriding has slightly higher 95th percentile latency and much lower 99th percentile latency. Again, these measurements are under high load. Nevertheless, it is an important observation that QProxy's behavior under high load could still be improved (see Figure 6.15).

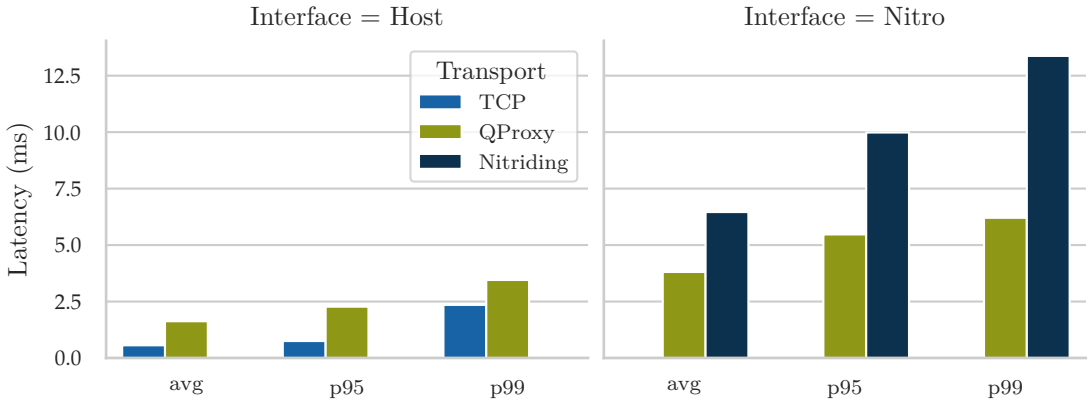


Figure 6.16: Average latency, 95th percentile, and 99th percentile of Redis latency as measured by memtier_benchmark when the load is limited to 10kops/s (deployed on an AWS EC2 server).

Under our usual 10k operations per second, we measure better performance with QProxy and better performance than Huawei's EC2 server. Using QProxy 99% of the requests are delivered in under 6 milliseconds. Nitriding is back to being about twice as slow as QProxy under this load (see Figure 6.16).

6.4.4 Compared to Nitriding

In our measurements, QProxy was always better than Nitriding, except for the latency while running Redis inside the Nitro enclave (see Figure 6.15). We consider this sufficient to prove that we have achieved the performance requirements that we established in Chapter 4. Our codebase is small, and we did not write any unsafe Rust code inside QProxy.

6.5 Scalability Analysis

To conclude our testing, we performed the same benchmarks changing the number of cores provided to the enclave. We started by isolating 16vCPUs and then subsequently ran the benchmarks giving the enclaves 2 cores, 4 cores, 6 cores, and so on, all the way to 16 cores. The memory allocated to the enclave and the EC2 instances stayed the same.

6.5.1 Throughput Benchmarks

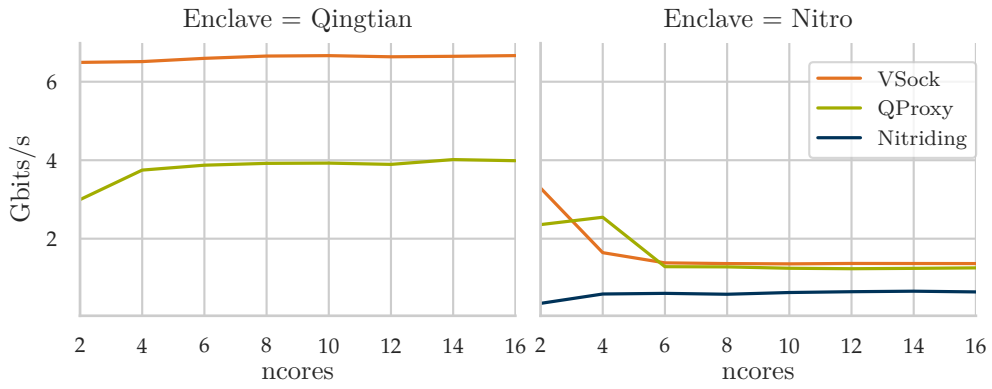


Figure 6.17: Enclave throughput measured by iperf in an AWS EC2 server, and a Huawei EC2 server.

In the Huawei EC2 instance, we measure a slight increase in throughput until the enclave has 6 cores, and then the throughput stays constant. The extra cores increase QProxy's throughput to almost 4Gbit/s, reaching just about 60% of the raw throughput (see Figure 6.17).

The Nitro enclave has an unusual behavior, with throughput dropping until 6 cores are allocated, then staying constant. QProxy's throughput increases over the VSock's throughput when using 4 cores, but then follows it and drops to about 1.7Gbit/s. This is not expected and probably indicates a bug in the enclave's VSock driver. Nitriding has a small increase in throughput at 4 cores and stays constant after that.

6.5.2 HTTP Throughput Benchmarks

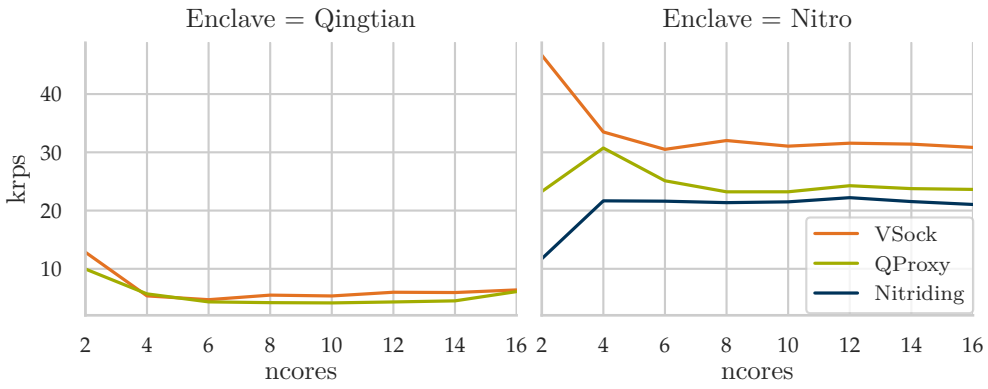


Figure 6.18: Enclave HTTP throughput measured by oha in an AWS EC2 server, and a Huawei EC2 server.

In this case, Huawei's HTTP throughput is heavily impacted. We have isolated this to an issue with the way cores are isolated (when isolating more cores, even if they are not used, the enclave receives a performance penalty on this benchmark). The issue has been reported and is being addressed by the Huawei Cloud team (see Figure 6.18).

Nitro shows the same behavior as with the interface's throughput; QProxy's throughput increases with 4 cores and decreases with 6 or more, the VSock's throughput decreases until 6 cores, and Nitriding's throughput increases with 4 cores, and stays stable with more than 4 cores.

6.5.3 Redis Throughput Benchmarks

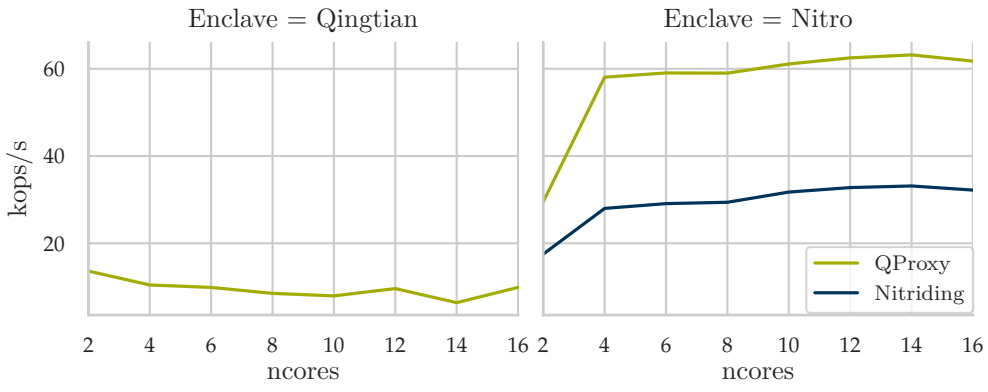


Figure 6.19: Enclave Redis throughput measured by memtier_benchmark in an AWS EC2 server, and a Huawei EC2 server.

Redis shows similar problems in Huawei as performance decreases with the increase in the number of cores, although it fluctuates more with higher core counts. Nitro shows a big improvement with 4 cores, almost tripling QProxy's performance, and doubling Nitriding's, then moderate gains until 16 cores where it is slightly lowered (see Figure 6.19).

7 Related Work

Two other VSock proxies serve the same niche as QProxy, although they only work on AWS Nitro type enclaves; Nitriding[36] and Enclaver[29]. Both of them are being actively worked on and have some very interesting features.

7.1 Nitriding

Nitriding is a tool implemented in Go that allows users to run applications inside an AWS Nitro enclave by setting up a TAP device that forwards packets over the VSock interface. They also automatically generate Let's Encrypt HTTPS certificates allowing them to automatically encrypt HTTP traffic to and from the enclave. This allows them to set up an HTTPS encrypted attestation endpoint, and provide a way to horizontally scale the enclaves by securely sharing the HTTPS certificate with attested enclaves.

This comes with a few drawbacks; there is no way to do an incremental upgrade on an enclave fleet as the enclaves authenticate each other by comparing their PCR values. A new version of the application would have a different PCR value and thus not be authenticated by the current fleet. They also rely directly on the AWS Nitro SDK, which means users would have to fork the application to provide support for other enclaves like Huawei's.

The TAP interface forwards all traffic from the enclave to the insecure host, making it harder to control what information is being exposed to the host, and providing their own Let's Encrypt certificate allows the Nitriding daemon to read the unencrypted messages.

These features provide a very easy to use application that allows users to spin up an enclave with access to the network with basically zero configuration. Go being a memory-safe language means that the application should be less susceptible to memory vulnerabilities and thus reduce the attack surface of the enclave.

We instead build QProxy as a component of a larger system, with the attestation and HTTPS certificates having to be provided by the enclave application. What we lose in ease of use, we gain in flexibility, allowing QProxy to run both in AWS Nitro and Huawei Qingtian enclaves. This reduces the attack surface of QProxy and makes it easy to verify our claims.

The TAP interface allows unrestricted traffic to and from the enclave. We instead prefer the more restricted approach of a TCP proxy, as the fine-grained control it provides is desirable for us. Building packet filtering on top of the TAP interface is a possibility, but it increases the attack surface of the proxy.

7.2 Enclaver

Enclaver is implemented in Rust and unlike Nitriding, it aims to replace the AWS Nitro CLI by creating a docker-like[8] CLI, with some additional features on top; a proxy similar to QProxy, and a wrapper for the AWS KMS API that will automatically attach the attestation document. Simplifying the process of securing information through the KMS and retrieving it inside an attested enclave.

It is built as a TCP tunnel instead of over a TAP interface which means it provides similar functionality to QProxy. Unlike QProxy it also provides a DNS resolver which allows applications inside the enclave to look up specific URLs and get them forwarded automatically. Enclaver also strives to support other enclave technologies, but it currently only supports AWS Nitro enclaves. However, the number of features relying on AWS-specific APIs might make it hard to port Enclaver to other enclave technologies.

Our analysis of Enclaver suggests it is a more secure program than Nitriding, as it does not expose the whole network inside the enclave to the host VM. This aligns more with our goals detailed in Chapter 4.

We believe QProxy improves upon Enclaver by decoupling the enclave-specific features from the VSock proxy. This allows building an ecosystem of tools around QProxy, without requiring the user to stay tied to a specific enclave technology. On the other hand, both Nitriding and Enclaver provide a better user interface than QProxy. We believe further improvement in that area is worth researching as it limits the adoption of secure enclaves.

7.3 QProxy

As we established in this work, QProxy is a VSock proxy with comparable performance to Nitriding in terms of latency and has better throughput. We also show that QProxy heavily relies on the number of cores allocated to the enclave and that the throughput of the VSock connection to the enclave is surprisingly low. So far, we have not seen QProxy bottlenecked by the throughput, but we hypothesize that if we were to allocate more cores to the enclave, we would quickly come to measure such a bottleneck.

Unlike Nitriding and Enclaver, we do not integrate with KMS services or provide attestation features. We expect this to be mitigated by future work that integrates with those services by building separate binaries. That way QProxy can keep focusing on providing a small and performant proxy that works regardless of the enclave technology, and other tools can provide attestation and integration with KMS services.

8 Conclusion

Users interested in the security of Nitro-type enclaves find that one of the main limitations of these enclaves is the use of VSocks to communicate with the enclave. Many existing applications have no support for VSocks and they would need to be modified to be used inside the enclave.

In this thesis, we presented QProxy, a VSock proxy that works with both Huawei Qingtian and AWS Nitro enclaves. We proposed limiting the design to only a proxy to allow for this portability. From our analysis of Nitriding[36] and Enclaver[29] in Chapter 7, this approach makes it less user-friendly, as many features users would want to implement are tied to the enclave provider; such as remote attestation and KMS integration. We believe the functionality that Nitriding and Enclaver provide by integrating directly with AWS Nitro and KMS services can be provided by an external service. This allows the proxy to focus on a narrower interface, keeping the codebase simple and easy to audit.

As we show in Chapter 6 we achieve our goals of being transparent to the confidential service, providing some access control, being easy to use, and achieving better throughput and latency than Nitriding while keeping the codebase smaller and the interface narrower. We also provide examples to retrieve the attestation document from different enclave technologies as well as a basic attestation document verifier showing that it is simple to implement a subset of the features provided by Nitriding and Enclaver with a small section of platform-specific code to handle the underlying enclave technology. This, among other reasons, is why QProxy is currently running in production in Huawei Cloud.

The source code of QProxy will be made available in conjunction with the next Qingtian release.

9 Future Work

In future work, we would like to see the following areas of interest developed:

- Provide remote attestation inside the enclave.
- Explore alternative proxy designs.
- Investigate the Qingtian enclave's performance issue.
- Improve the VSock driver's performance.

9.1 Provide Remote Attestation

Even though we provide examples of how to generate the attestation documents for AWS and Huawei enclaves, providing these features through a separate library would open possibilities such as supporting AMD SEV and Intel SGX enclaves, which do not require a VSock proxy, but would benefit from a common API.

9.1.1 Integrate with KMS

A follow-up topic would be integrating the proposed remote attestation library with different KMS offerings, allowing one to retrieve data stored in the KMS when an enclave is attested. This would provide a similar ecosystem to what Enclaver[29] is building, but it would be modular, meaning a user could remove the proxy component if they were running their application inside an SGX enclave.

9.1.2 Integrate with the Linux Kernel

Current remote attestation solutions build around the specifics of the host enclave (e.g. AWS Nitro), this duplicates some of the work required to support other enclaves a promising alternative is the configfs-tsm[35] Linux kernel ABI which has been included since version. It is currently only supported by Intel TDX and AMD SEV-SNP, and it does not define a common format for the attestation report. More standardization efforts like this are welcome to improve the current state of remote attestation for TEEs.

9.2 Explore Alternative Proxy Design

We eliminated some proxy options mentioned in Chapter 4 because there was no current way to implement them in safe Rust. More work is needed to evaluate the performance difference of such options.

io_uring is especially interesting as the runtime itself can be single-threaded, thus removing the Send and static bounds on Rust Futures. This is desired because Send and static bounds usually mean the futures need to be boxed[1] which may decrease performance due to unneeded allocations and memory fragmentation. Send bounds also prevent the use of thread-local variables on Futures; removing the bounds would allow for thread-local arenas which provide very fast allocation without a need for synchronization.

9.3 Improving Enclave Performance

Although we did not evaluate this, the performance impact of the enclave's memory protections should be measured; a lot of common wisdom about how to structure applications could be wrong because of the memory encryption applied by the enclave. Investigating the penalty imposed on common operations like memcopy, malloc, and random array accesses is worthwhile.

We also see weird behavior (see Chapter 6) when isolating more than two CPUs in the Qingtian enclave, this should be investigated and fixed.

9.4 Improving VSock Performance

The VSock drivers lag behind the TCP drivers, even though VSocks can theoretically be faster and lower latency as there never is a physical device that needs to be interacted with, and thus all operations can be done in memory. This performance problem is detailed in Chapter 6, but it is clear to see, that there is still a big gap between the drivers' throughput as measured with iperf3.

The kernel is not the only piece that lacks proper support for VSock connections. As we show, AWS's and Huawei's enclaves also have low throughput when communicating with the host through VSock channels. Both the kernel and the hypervisors should focus more on the performance of VSock connections if we want to run applications relying on higher network throughput.

[1] allocated on the heap

Abbreviations

TEE Trusted Execution Environment

SGX Secure Guard Extensions

TDX Trust Domain Extensions

SEV Secure Encrypted Virtualization

SNP Secure Nested Paging

PCR Platform Configuration Register

TPM Trusted Platform Module

VM Virtual Machine

CID Context Identifier

VSock Virtual Socket

AWS Amazon Web Services

EIF Enclave Image File

QTSM QingTian Security Module

KMS Key Management Service

IAM Identity and Access Management

MITM Man in the Middle

CLI Command Line Interface

CVE Common Vulnerabilities and Exposures

List of Figures

List of Tables

Bibliography

[1] AMD. *Secure Encrypted Virtualizetion API Version 0.24*. Apr. 2020.

[2] M. Azure, ed. Oct. 8, 2023. URL: `https://learn.microsoft.com/en-us/azure/networking/azure-network-latency?tabs=Europe%2CCentralEurope` (visited on 03/05/2024).

[3] R. Buhren, H.-N. Jacob, T. Krachenfels, and J.-P. Seifert. "One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization." In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 2875–2889.

[4] J. Corbet. "Rethinking splice()." In: *LWN.net* (Feb. 17, 2023).

[5] J. Corbet. "Ringing in a new asynchronous I/O API." In: *LWN.net* (Jan. 15, 2019).

[6] J. Corbet. "The rapid growth of io uring." In: *LWN.net* (Jan. 24, 2020).

[7] V. Costan and S. Devadas. "Intel SGX explained." In: *Cryptology ePrint Archive* (2016).

[8] I. Docker, ed. *Docker*. 2024. URL: `https://www.docker.com/` (visited on 02/13/2024).

[9] S. Garzarella. *iperf3-vsock*. 2023. URL: `https://github.com/stefano-garzarella/iperf-vsock` (visited on 03/01/2024).

[10] *gvisor-tap-vsock*. 2024. URL: `https://github.com/containers/gvisor-tap-vsock` (visited on 04/04/2024).

[11] hatoo. *oha*. 2024. URL: `https://github.com/hatoo/oha` (visited on 02/29/2024).

[12] M. Holt. *Caddy Web Server*. 2024. URL: `https://github.com/caddyserver/caddy` (visited on 03/12/2024).

[13] M. Holt. *Igor Sysoev*. 2024. URL: `https://nginx.org/en/` (visited on 03/12/2024).

[14] A. R. Hummaida, N. W. Paton, and R. Sakellariou. "Adaptation in cloud resource configuration: a survey." In: *Journal of Cloud Computing* 5 (2016), pp. 1–16.

[15] IEA. *Global data centre energy demand by data centre type*. report. IEA, 2010-2022.

[16] E. Institute. *2023Statistical Review of World Energy*. report. Energy Institute, 2023.

[17] Intel. *Intel Software Guard Extensions Developer Guide*. 2014.

[18] Intel. *Intel® Trust Domain Extensions*. Feb. 2023.

[19] Y. C. Lee and A. Y. Zomaya. "Energy efficient utilization of resources in cloud computing systems." In: *The Journal of Supercomputing* 60 (2012), pp. 268–280.

[20] M. Majkowski. *SOCKMAP - TCP splicing of the future*. URL: https://blog.cloudflare.com/sockmap-tcp-splicing-of-the-future/ (visited on 11/02/2023).

[21] F. T. Maxim Krasnyansky Maksim Yevmenkin. *Universal TUN/TAP device driver*. 2002.

[22] S. McArthur. *hyper*. 2024. URL: https://github.com/hyperium/hyper.

[23] J. D. S. Messina. *oha: Add VSOCK support*. 2024. URL: https://github.com/hatoo/oha/pull/416 (visited on 03/22/2024).

[24] J. D. S. Messina. *tokio-vsock Contributions*. 2024. URL: https://github.com/rust-vsock/tokio-vsock/pulls?q=is%3Apr+author%3Ajalil-salame (visited on 03/22/2024).

[25] S. K. Mishra, R. Deswal, S. Sahoo, and B. Sahoo. "Improving energy consumption in cloud." In: *2015 Annual IEEE India Conference (INDICON)*. IEEE. 2015, pp. 1–6.

[26] A. Nilsson, P. N. Bideh, and J. Brorsson. "A survey of published attacks on Intel SGX." In: *arXiv preprint arXiv:2006.13598* (2020).

[27] *QingTian Enclave*. White paper. Accessed: November 2023. Huawei, Sept. 27, 2023.

[28] RedisLabs. *memtier benchmark*. 2024. URL: https://github.com/RedisLabs/memtier_benchmark/pull/237 (visited on 03/07/2024).

[29] A. C. Russel Haering Rob Szumski and E. Yakubovich. *enclaver*. 2023. URL: https://github.com/edgebitio/enclaver (visited on 11/02/2023).

[30] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. *Malware Guard Extension: Using SGX to Conceal Cache Attacks*. 2019. arXiv: 1702.08719 [cs.CR].

[31] A. Sev-Snp. "Strengthening VM isolation with integrity protection and more." In: *White Paper, January* 53 (2020), pp. 1450–1465.

[32] *The Security Design of the AWS Nitro System*. White paper. Accessed: October 2023. Amazon Web Services, Nov. 18, 2022.

[33] *The State of eBPF*. report. The Linux Foundation, Jan. 2024.

[34] N. Tziritas, S. U. Khan, C.-Z. Xu, T. Loukopoulos, and S. Lalis. "On minimizing the resource consumption of cloud applications using process migrations." In: *Journal of Parallel and Distributed Computing* 73.12 (2013), pp. 1690–1704.

[35] D. Williams. *configfs-tsm: Unified attestation report ABI for v6.7*. Nov. 2, 2023. URL: https://lore.kernel.org/lkml/654438f4ca604_3f6029413@dwillia2-mobl3.amr.corp.intel.com.notmuch/.

[36] P. Winter, R. Giles, M. Schafhuber, and H. Haddadi. *Nitriding: A tool kit for building scalable, networked, secure enclaves*. 2023. arXiv: 2206.04123 [cs.CR].