LONDON'S GLOBAL UNIVERSITY

Securing Web Server Private Keys with Memory Protection Keys

Hardening Nginx and OpenSSL against private key leakage using secure thread-based privilege separation

Christopher Hammond

Computer Science (MEng)

April 2019

Supervised by: **Prof. Brad Karp**

Disclaimer: *This report is submitted as part requirement for the MEng degree in Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.*

Abstract

With software vulnerabilities in popular software published on an almost daily basis, it is important that software with access to sensitive data is written to withstand hacking attempts by powerful adversaries. Whilst employing best security practices in development is important, this report argues that developers must also actively consider that even their best code is likely still vulnerable, and they must therefore take action ahead of time to limit the possible damage in the event of their programs being hacked. One such method for limiting the impact of software exploitation is *privilege separation*, which involves dividing a program into multiple smaller components, each of which is limited in scope and access. Each of these smaller components should work together to perform the program's tasks, whilst ensuring that a hacker with control over one component cannot easily move laterally into another to steal sensitive data, such as private keys and passwords.

In this project report I present research into protecting RSA private keys for the Nginx Web Server from theft in the event of a remote code execution vulnerability. Two forms of privilege separation are offered. The first build uses two processes with Inter-Process Communication (IPC) via a Unix socket pair to allow the generation of RSA signatures for TLS 1.3 handshakes without the private key being situated in the memory of an Internet-facing process. This is a typical method of privilege separation, and is consistent with the current state of the literature.

The second build uses a novel application of POSIX threads as a privilege separation method, with protection of the in-memory private key handled by Intel Memory Protection Keys (MPK). MPK is a new processor feature included in the Xeon Skylake-SP "Scalable" range of server processors which allows granular control over access to memory regions on a per-thread basis.

The report further presents a full performance analysis of this code, both on a code level and a system-wide level, and demonstrates that using MPK and Threads to separate privilege can be over 55% more efficient when compared to using processes and IPC. The overhead on an RSA encryption operation is shown to be under 1.6% when MPK is used.

Finally, the report critically analyses the experimentation procedure and results, and suggests a plethora of future work to make mass availability of web server privilege separation in the TLS handshake a possibility in the future.

Contents

Acknowledgments

I would initially like to thank Prof. Brad Karp for his continued and unwavering support throughout this project. His guidance, clarity, reliability and direction have all been invaluable to this project's success. Thank you so much!

For keeping me sane throughout the project I would like to thank my best friends. Firstly, special appreciation goes to Hena Ramdany, who I live with. She has been woken up far too many times when I have come back late at night (or, perhaps more accurately, early in the morning) from University after working on my project. I am truly sorry that you lost sleep, but I hope my investment of your sleep into web server security is enough consolation!

Furthermore, I would like to apologise (and give thanks, but mostly apologise!) to Matt Policane and Ashu Savani for all the second hand stress I have given them every time I have used the word *dissertation* over the past six months. They have both been of huge support to me over the past year and not just with this project. Thank you both so much.

And finally, my family. Firstly, to my Mother and late Father, both of whom have made innumerable sacrifices for my education, thank you for everything. Neither of them would have a clue what most of this project means, but they'd probably nod, smile and say "well, it *looks* impressive anyway..." which is good enough for me! I would also like to thank my brother, Nicholas, for being exceptionally supportive and positive.

1. Introduction

1.1. Background Context

THE threat landscape in the technology industry has changed considerably over the past few years, and will continue to change as time goes on. The era of victims accidentally sending viruses disguised as love letters by email that delete some hard drive files [1] has now passed; hackers are now adopting a highly targeted and technical approach to cybercrime aimed at large businesses, governments and NGOs to exfiltrate intelligence. These attacks are often the work of large teams protected and/or funded by nation state actors with the resources to obtain targeted data or access with finesse. Often these nation states, including those within the *Five Eyes* group of countries such as the United States of America, as well as powerful adversaries such as China and Russia, will harvest so-called "Zero Day" exploits for popular software [2] ready for use on a high profile target whenever they see fit.

The practice of governments saving up software vulnerabilities instead of disclosing them to manufacturers is so well known now that GCHQ has published their so-called "Equities Process" online [3], describing the formal process to decide whether to inform vendors of security vulnerabilities they have found in their products. In some cases, this activity has backfired in a very public way. For example, the ShadowBrokers leak [2, 4] made a very powerful exploit for Windows file sharing available to anybody online and it was rapidly used to deploy ransomware including WannaCry, which cost the NHS £92m and 19,000 missed appointments [5], and NotPetya that was responsible for bringing the major international shipping firm, Maersk, to its knees [6].

Whilst even as of 2019 the vast majority of attacks involve a human aspect, such as careful social engineering, to access company networks, this is by no means always true, as these attacks show. In the Equifax case, for example, unprecedented confidential data leakage occurred due to an unpatched Apache Struts [7] server being left open to the world, ready to be exploited by a determined adversary.

With this knowledge in hand, it seems reasonable that as members of the security community we should approach security from both ends of the attack. We should of course aim to build systems that are secured to the best standards available, with regular reviews to stay up to date with the latest Tactics, Techniques and Procedures (TTPs) in use by adversaries [8]. However, this is not necessarily enough when you do not know who your enemy is or what their capabilities are. Therefore, in addition to following best practices, it is also crucial that we build systems with the expectation in mind that *at some point they will be breached.*

Not only may the system be breached in the future, but it is not always possible to detect straightaway when a breach has occurred, who is responsible, what their motives are, what they may have taken or what they may have left behind.

However, the news is not all negative in the security landscape. Naylor *et al.* [9] have concluded that "the infrastructural cost of HTTPS is decreasing". This is very clear to see from reports online, including Google's Transparency Report [10] which claims that over 85% of time spent by users in Google Chrome as of April 2019 is spent on websites with HTTPS enabled. Another report by ESET [11] states that as of September 2018 the "majority" of the top million websites on the internet have encryption enabled. There are a number of reasons for this rapid uptake in encryption. One very likely reason was the release of the Edward Snowden files that publicly proved for the first time the wide-ranging abilities of nation states to monitor internet traffic [12]; another is inevitably that the real cost of HTTPS is not particularly high. Back in 2010 Langley *et al.* calculated $< 1\%$ CPU overhead when HTTPS was enabled across the board on Gmail [13]. With uptake so high, and ever increasing, we can now as a security community turn to refining this security so that critical data, such as web server private keys, do not fall into the wrong hands. With private keys leaked, HTTPS is rendered essentially useless as stolen keys can be used to impersonate victim sites if the theft is not discovered. This is not an unreasonable attack to protect against, either. There are several examples over the past few years of stolen private keys used for code signing being stolen from legitimate vendors to make malware appear authentic [14], so it is not unreasonable to suggest that a nefarious adversary may attempt to steal web server private keys too.

With all this background in mind, we can start to define new requirements for software security in which programs are not just built to resist attack, but also to reduce the overall impact of a security vulnerability should one be left unnoticed by limiting the access an attacker would have after exploiting it. This technique is called *privilege separation* [15] and it is the basis of this project.

When a program is privilege separated it is divided into multiple smaller components, each isolated from one another, which communicate via a tightly controlled interface. Components have access only to the restricted subsets of data and operating system privileges that allow them to carry out their individual tasks, whilst being sufficiently isolated that if one component is placed under the control of an attacker the other components are not affected. Leveraging this principle allows developers to write programs in such a way that any potentially hostile inputs, such as from the Internet, are processed by a component that is not trusted with any sensitive data, such as private keys or passwords. If a less trusted component has to perform a task that ordinarily would require usage of sensitive data, it can use the interface between components to delegate small workloads to the higher privilege components, and then use the results in its task. This means that the sensitive data is never available directly to a component that may handle hostile inputs, making penetration of that component alone not enough to steal sensitive data.

Sometimes privilege separation happens across large systems due to business choices that have been made, perhaps based on risk or perhaps as a direct result of building separate systems that work together to achieve some common aim. For example, when the genealogy website *MyHeritage* was hacked in October 2017 [16] they claimed that although over ninety-two million customer records were stolen, this data did not include any of the extremely personal DNA records that the company stores to offer its services; DNA data was allegedly located on “segregated systems” with “added layers of security”. However, this project will not concentrate on privilege separation through such business decisions, and instead take a purely technical approach by protecting just one service running on a single server, a web server, from having its private key stolen in the event of exploitation by a bad actor.

1.2. Transport Layer Security (TLS) Protocol

In order to establish a secure tunnel between a web server and a client, a protocol called Transport Layer Security (TLS) is commonplace. TLS describes a handshake by which a client and server can authenticate one another, prior to deriving an ephemeral session key to exchange data during the session. Although both the client and server can authenticate one another, this scenario is rare; the most common scenario is a client ensuring that they can trust that they are truly connecting to a website such as a bank, and not connecting to a fake website impersonating their bank or connecting to their bank via an adversary capable of eavesdropping the traffic. There are several releases of the TLS Specification, but the version that this project will focus on is exclusively the latest version, TLS v1.3, which is defined by the Internet Engineering Task Force (IETF) Request for Comments (RFC) 8446 [17] published in August 2018.

The handshake combines authentication and session key derivation into three packets in the basic case. The authentication phase requires the server to present a certificate signed by a mutually trusted Certification Authority (CA) identifying itself, along with a signature proving to the client (*e.g.* a web browser) that it is in fact in possession of the private key associated with the certificate’s public key. If the client is able to successfully authenticate the server whilst performing a Diffie-Hellman key exchange [18], an ephemeral symmetric session key

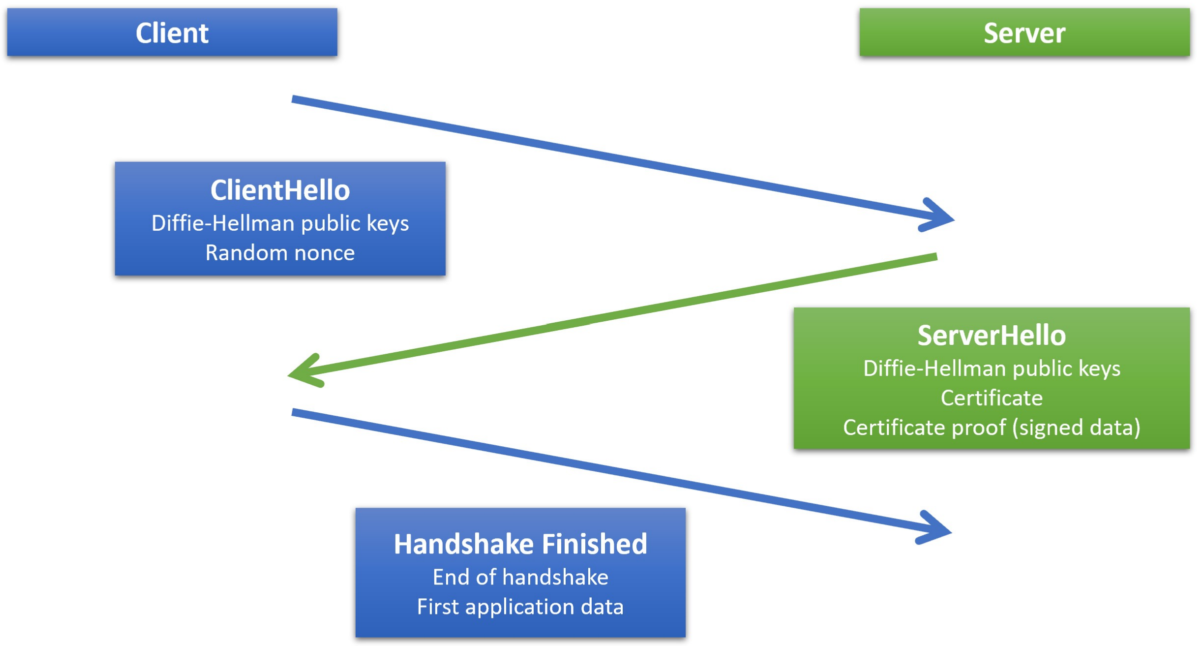


Figure 1. TLS v1.3 Handshake Overview

is derived, and this is used to exchange encrypted data for the duration of the session. Figure 1 depicts a simplified view of this handshake in the standard case of a web browser connecting to a website that offers TLS v1.3 support that it has not connected to before.

This handshake has been accepted as a modern industry standard, but there is still a single point of failure in the establishment of trust between two endpoints: the private key. The private key is issued by a CA, typically for a period of a number of years, meaning that there is a long time for a potential adversary to work on stealing it. If that private key were indeed to be stolen and the link between the client and the server hijacked in some way (e.g. via a DNS attack, or by a nation state capable of eavesdropping on and tampering with the communications infrastructure) then the communication would no longer be secure. This is because an evil third party could convince the user that it is the server the user is expecting to communicate with, and it could convince the real server that it is indeed the client, since clients rarely authenticate themselves to servers. This works so long as neither the server nor the client ever communicate directly for the duration of the session, and the adversary would have full visibility of all transactions with the ability to tamper with the communication in both directions. For example, if an adversary had stolen the private key for a bank, it could sit between a bank and a legitimate user and redirect any transaction requests to an adversary-

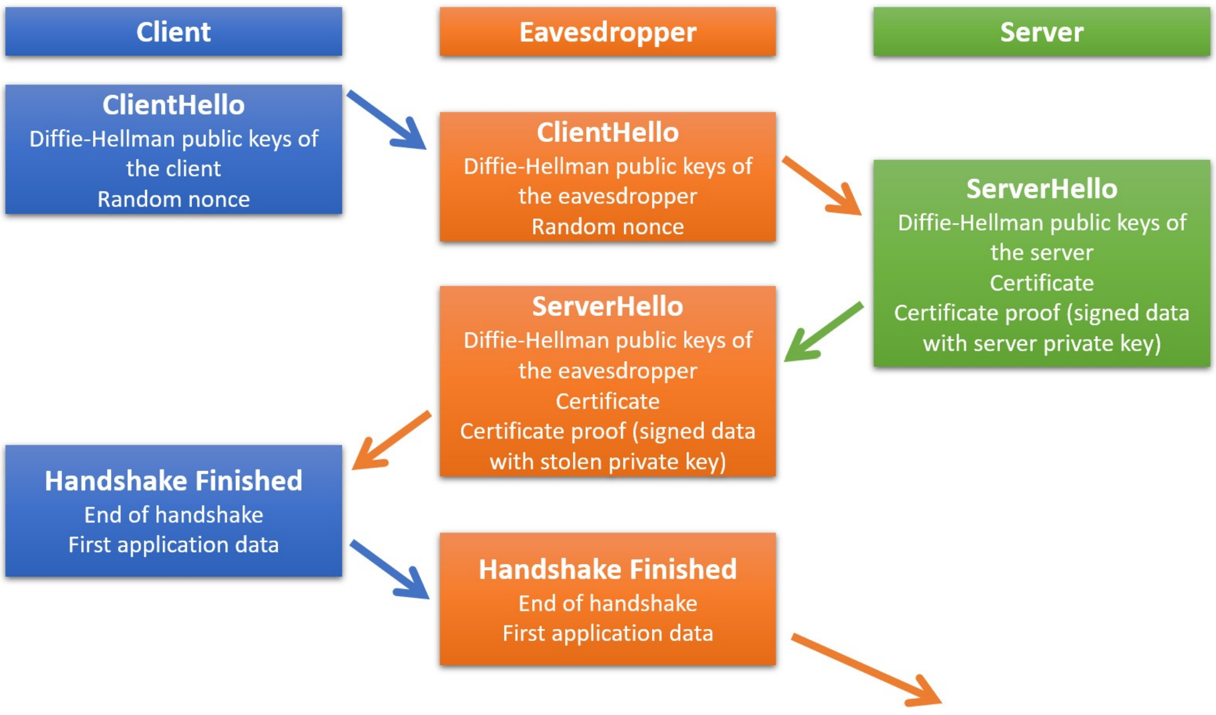

Client
Eavesdropper
Server
ClientHello
Diffie-Hellman public keys of the client
Random nonce
ClientHello
Diffie-Hellman public keys of the eavesdropper
Random nonce
ServerHello
Diffie-Hellman public keys of the server
Certificate
Certificate proof (signed data with server private key)
ServerHello
Diffie-Hellman public keys of the eavesdropper
Certificate
Certificate proof (signed data with stolen private key)
Handshake Finished
End of handshake
First application data
Handshake Finished
End of handshake
First application data

Figure 2. Man-in-the-Middle Attack on a TLS v1.3 Handshake with a Stolen Private Key

controlled account. Figure 2 demonstrates how this attack might be carried out in the case that the adversary is able to hijack traffic by controlling a router between the server and the client, a type of *Man-in-the-Middle Attack* [19].

1.3. The Nginx Web Server

Analysing and augmenting web server security requires a base web server to work with. Nginx is a highly performant [20], extremely popular [21] event driven web server application that can both serve web pages and act as a web services (HTTP/HTTPS) proxy. It is frequently used to terminate HTTPS connections by creating a secure Transport Layer Security (TLS) channel between the server and each client that is making connections and requesting resources. Nginx may then service requests either by serving static content up itself from storage, proxying connections back to an internal service via an HTTP(S) request or Unix socket, or indeed combine both into a useful response for a client.

The current state of Nginx is that it will load private keys into the memory of a Internet-facing worker processes via OpenSSL so that they can authenticate themselves to clients, but

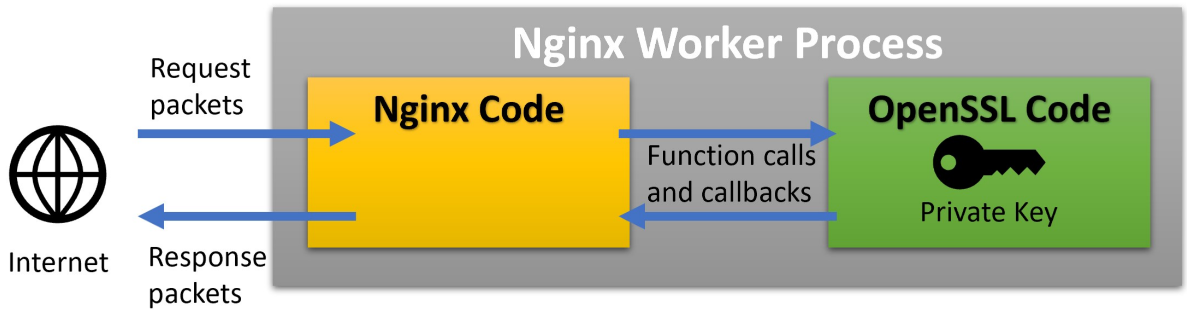


Figure 3. Location of the TLS Private Key in Nginx Worker Process Memory

this also means that should Nginx be remotely exploited by a powerful third party in such a way that arbitrary memory locations can be read, it's perfectly possible that a private key could be read out of memory and in turn fall into the wrong hands. Figure 3 shows very basic overview of the set up of a single worker process, with Nginx and the OpenSSL library sharing memory to service requests that come in from the outside world.

Given that the private key for the certificate forms the basis of this chain of trust in the handshake, it follows that a systems administrator in charge of configuring an Nginx web server utilising this key should do their best to protect it and keep it from falling into the wrong hands.

1.4. Project Description

This project will focus on the common case of a single website hosted on a single server with a single certificate and a 2048-bit RSA private key. To assist in mitigating against the risk of a private key being leaked in the event of an adversary with remote code execution capability, two methods of protecting the private key from being accessed will be explored. The first will be a more traditional multi-process approach in which the private key is loaded into totally separate memory, with encryption happening via a thin Remote Procedure Call (RPC) interface between the two processes (Figure 4). In the second approach, Intel's new Memory Protection Keys (MPK) [22] feature will be employed so that access to the RSA private key's memory can be isolated to a special thread dedicated only to encryption. The main worker thread will then be denied access to the private key's memory which would prevent attacks on that worker thread from being able to exfiltrate the private key without somehow unlocking that memory, or hijacking the second thread.

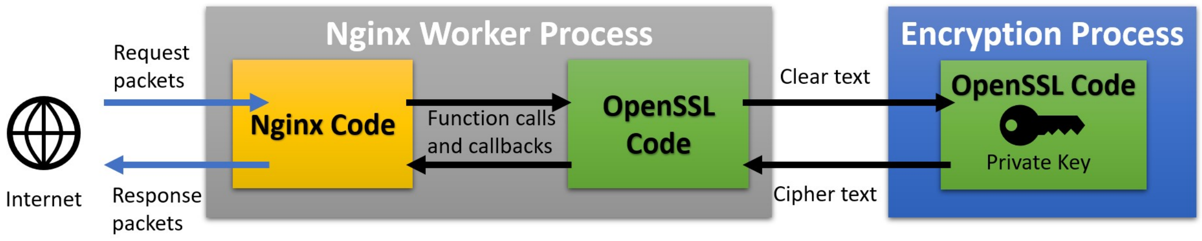


Figure 4. Architecture of Nginx with Process-Based Privilege Separation

MPK is a processor feature first made available in the new Intel Xeon Skylake-SP ”Scalable” Server Architecture. It allows the creation of up to sixteen *protection keys*, each of which can be set to allow full memory access, allow reading but not writing or deny all access. To control access to some memory, it must be mapped using the standard C `mmap` system call, after which the new `pkey_mprotect` call can be passed a pointer to tag the pages of memory pointed to with a protection key that defines their access level. When a protection key's access level is changed, all the pages of memory *tagged* with that key will now have new access permissions. Furthermore, the configuration for each protection key is stored within the new MPK register, `PKRU`, which means that each key can have a different access level depending on the thread that is executing, even if those threads reside within the same process. Protection settings can be changed by the program when needed, allowing locking and unlocking of memory by the program at run time. Figure 5 shows how thread-based privilege separation with MPK would work. Note that even though the private key is technically in memory shared between both threads, as memory between threads within the same process is shared, only the encryption thread has permission to read that memory thanks to MPK.

The eventual aim of the project is to ascertain whether process-based or MPK-based privilege separation work in this context, to what extent they can protect against attacks where an attacker has remote access to program memory and, importantly, what the performance overhead is for such mitigation methods. By building a version of the Nginx web server with privilege separation enabled, the idea is that even a software bug as devastating as 2014's *Heartbleed* vulnerability [23], which allowed reading of the web server's process memory remotely, should not make it possible to steal the server's private key because the memory containing the key will either not be addressable (in the case of process-based separation) or will be locked (in the case of MPK).

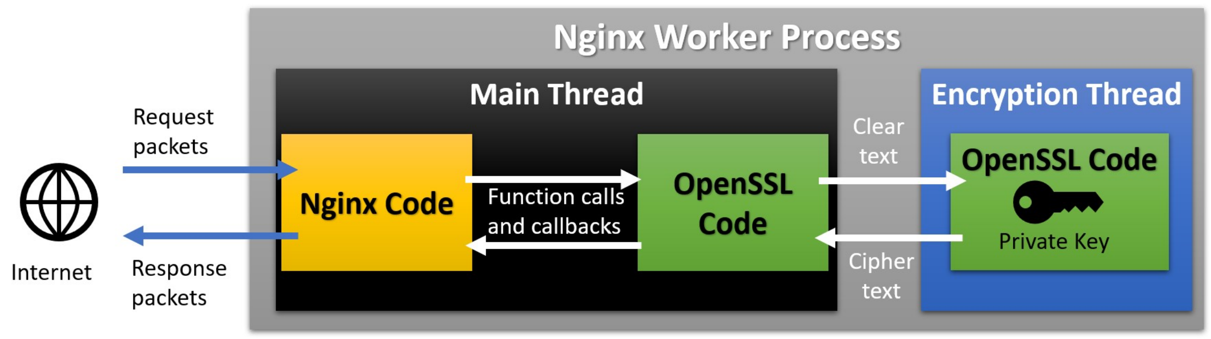


Figure 5. Architecture of Nginx with Thread and MPK-Based Privilege Separation

2. Related Work

Privilege Separation is not a new topic in the fields of Computer Security and Computer Systems, and many computer scientists have worked in this field over the years. This section covers a number of the prior works that inspired this project and discusses their achievements, along with some limitations that this project will attempt to address.

2.1. Privilege Separated OpenSSH

Preventing Privilege Escalation [15] describes work done in 2002 to split up the OpenSSH server into multiple isolated components. SSH by design is a service that requires a high level of privilege on the server, and indeed `sshd`, the SSH server daemon, is owned and run as root. This is because the program allows remote login including validation of credentials, such as keys and account passwords, for all users on the system; this is something only the root user can do.

The major danger of running a program like an SSH server as root is that potentially hostile inputs coming from the Internet get parsed and processed by it, and if there is a programming error in the SSH daemon then it could be possible for an attacker to take control of the process and execute arbitrary code as the root user. Such an attack could therefore grant a bad actor unfettered access to the remote system, potentially without even the need for any authentication. The authors of this paper recognise that such software bugs are possible, and indeed have happened in the past [24], and propose therefore that by building OpenSSH with privilege separation such bugs would be far less damning as they would be limited to “unprivileged slave” processes which do not have access to, for example, the password database.

The project was able to successfully split up OpenSSH by delegating higher risk operations that communicate with the outside world to a lower privilege process. The separation is achieved at a technical level by using the `fork()` system call to make a copy of the *monitor* process, creating a *slave*. The slave processes are also given different User IDs to ensure that they cannot signal the monitor (e.g. to cause denial of service to it) or use a tool like `ptrace` to gather intelligence on a running monitor from a slave. Processes are also placed in a `chroot` jail, in which their new root directories are empty, to ensure that any injected malicious code cannot read any files on a vulnerable host without successful authentication.

Data is exchanged between the higher privileged monitor and lower privileged slaves using a combination of shared memory and the Linux `socketpair` system call. The data serialisation method is similar to that of Sun’s 1987 External Data Representation (XDR) format [25]. The team only had to change around 2% of the existing code to privilege separate it, and they found that the extra security “imposes no significant performance penalty”.

This is clearly a very successful project to protect the OpenSSH server from the damning effects of remote code execution exploits. The idea to use separate processes with socket pairs between them for data exchange can be directly applied to this project. However, this project’s design differs significantly in approach to the paper. Whilst the OpenSSH work focused on reducing the privileges of potentially harmful code, this project’s plan is to protect the RSA private key in a web server, a very small amount of sensitive memory, from the rest of the program by moving the encryption operation away. This makes more sense for this project for two reasons. Firstly, the RSA encryption operation is a significantly smaller region of code to modify when compared to refactoring all of Nginx’s network request code. Secondly, by making the changes to OpenSSL instead of Nginx, it will be easier in the future to port the work, if successful, to other web servers such as Apache [26].

2.2. Wedge

Wedge [27, 28] was a 2008 project designed to separate applications into smaller so-called *sthreads*. An sthread differs from a normal thread in that it is invoked with, by default, an empty set of privileges, but additional privileges can be provided before the thread is launched if needed. For example, an sthread by default will not inherit access to `stdin` from its parent but the Wedge API allows this access to be granted if and only if it is needed.

Sthreads are a powerful idea because they allow code to be written with privilege separation built in from the outset. Programmers do not have to handle the technical process of making `fork()` calls or creating inter-process communication, such as via socket pairs, as the Wedge API will manage all of this. Instead, developers just need to be aware of what data and which specific privileges each part of their program requires to complete a given task and then they can implement it as an sthread.

In order to protect data from attack, each sthread is set up with fresh memory that has a copy-on-write (COW) view of the program's memory prior to the `main()` function running, meaning that any program variables, aside from any initialised globals, will not have been populated yet. However, any initialised `structs` will properly be inherited so that future memory allocation will work as though the sthread were to be a program in its own right. The COW ensures that data does not unnecessarily leak from `main` into the lower privilege sthread. Furthermore, it means that if an adversary is able to inject malicious code into an sthread, it is only able to access and modify its own view of memory, and not the memory of any other sthread. The only way that data can pass between processes is via the Wedge API.

Although sthreads are treated as threads, in actual fact an sthread bears more relation on a technical level to a separate process such as one created via the `fork()` system call. However, the API to create an sthread is deliberately similar to that of the well-known `pthread` [29] library so that experienced C developers can pick up sthreads quickly.

The authors define further the term *callgate*. A callgate is an sthread that has access to some higher privileges, such as access to files owned by root on the host for authentication purposes. The idea is that callgates get invoked by lower privilege parts of an application when a high privilege operation needs to be performed. In the context of this project, if it were to be built with Wedge, the RSA private key operations would be implemented as a synchronous callgate, which would allow the calling sthread to pass plain text to the callgate, wait for the encryption to be performed and then continue servicing a web request from the calling sthread with the cipher text result. Callgates are supposed to be short routines that are given the most attention when audited to ensure that when inputs are passed to them it is not possible to exploit their higher permissions to steal secrets from or seize control of the server.

Wedge also provides a tool known as *Crowbar*, which helps programmers to identify which areas of memory are used where in a legacy (i.e. non-privilege separated) application to assist with migrating to `sthreads`. The paper's authors argue that Crowbar is intentionally offered as a programmer's aid as opposed to an automated tool to generate code with sthreads, as to automate a process like this "could lead to subtle vulnerabilities".

A key downside to the Wedge project is that it require significant refactoring of existing applications to make use of. Sthreads are not a drop-in replacement for existing functions, and although they are powerful they require a lot of additional code to be included for the full Sthreads interface. Bittau privilege separated OpenSSH in a similar manner to Provos *et al.* and concluded that although approximately the same amount of code is trusted in both implementations, sthreads has "perhaps double" the attack surface due to the generalised kernel implementation. That said, Bittau still argues that a generic implementation is better due to the reduction in the amount of code that needs changing to privilege separate a program.

Another potential limitation is that sthreads makes heavy use of a Linux kernel patch to deliver sthreads, which limits portability. Bittau does also deliver a userspace implementation of sthreads but notes "some performance cost" in this approach.

However, despite the overhead of additional code either in the kernel or in userspace along with the sthreads library itself, the overall performance of Wedge is similar to that of using `fork()`, and the authors estimate that for larger amounts of memory using an sthread in place of a traditional `fork()` will actually yield a noticeable performance gain.

2.3. OKWS: The Ok Web Server

OKWS [30] is a web server built similarly to Nginx, in that it is fully event driven so that it can handle thousands of concurrent connections within each process, but privilege separation is included in the architecture from the get go. The paper, similar to this project, takes the view that as bugs are inevitable software developers should work to "limit the effectiveness of attacks when they occur". OKWS became the basis of the online dating website *OkCupid*, of which Krohn, the paper's author, was a co-founder.

Within OKWS every server process is jailed to its own directory via `chroot`, a technique also adopted by Provos *et al.* in OpenSSH. This has the advantage that even if remote code execution in a server process is obtained by an adversary, their impact is limited to the permissions available to the user the process is running under, and only files within the directory the process is jailed to can be accessed. OKWS also defines a structured Remote Procedure Call (RPC) protocol for services to communicate with one another where cooperation is required to service a request. For example, if an Internet-facing web service needs to fetch data from a database, it will send an RPC message to a data service that queries the database for the required data in its behalf, and returns it to the requesting web service via RPC. This means that even if the web server is remotely penetrated, the attacker would not gain database access credentials as no Internet-facing services are provided with them; all database operations must go via a separate database service.

Krohn notably defines a crucial trade-off between security and performance in his design. In order to provide full isolation, in theory it would be necessary to not only isolate services from one another, but also to isolate users from one another by creating many copies of the OKWS services, each of which would serve one user at a time. However, in the case of thousands of users, many thousands of processes would be required to run at once and the context switching between these would be extremely intensive on the server hardware. Therefore, a decision was made to instead optimise OKWS to serve many thousands of users per process so that in the event of remote code execution, data leakage would be restricted to user data handled by that service only, and not data from across other services.

OKWS achieves an impressive privilege separation model that would reliably protect services against vulnerabilities in other services so long as the RPC interface itself does not open the individual services up to attack via a remote code execution exploit. Gaining access to the database as an adversary would require at very least two exploits to be chained together: one to control the Internet-facing web service and another in the database that can be exploited via the RPC interface.

2.4. Titus

Titus [31, 32] is a proof of concept reverse proxy created by Andrew Ayer that is built with privilege separation in mind, created in the wake of the Heartbleed vulnerability. There are several large differences in implementation between Titus and OKWS, as they are built with different aims and priorities in mind. Whilst OKWS aims to be a complete web application server unto itself, with support for many smaller interconnected services working together to build and host a system, Titus does not implement any of the complex HTML templating features or a generic RPC interface; instead, it leaves the workings of the web service to a separate program running behind its reverse proxy.

The privilege separation that Titus provides is twofold. Firstly, every incoming TLS connection spawns a new process for the request to be serviced with. Secondly, and most interestingly for this project, the private key is stored in a separate, non-Internet-facing process which is called upon to perform encryption operations. This idea is excellent because it means that the private key is never stored in memory space that is addressable by any of the Internet facing processes that could in theory be directly hacked and controlled by a bad actor. It also stops vulnerabilities like Heartbleed from being able to access to the memory with the private key, thus preventing its disclosure. Similar to OKWS, multiple vulnerabilities would need to be chained together to compromise the private key; Heartbleed alone, for example, would not be enough.

Another security feature that Titus adds is a new Unix User ID for each TLS process which adds further isolation to the processes. As every TLS request is handled by a new process, and each process is owned by a different, unprivileged User ID, there is no way for them to read each other's memory. Therefore if a bad actor manages to execute code within the process assigned to their connection, they cannot read the memory of any other user's connection, nor can they read the server's private key, as these are all fully isolated.

Even though what Titus can do is impressive, there are a number of issues with using it in production. Firstly, the project has not been updated on GitHub since 2015, meaning that if any vulnerabilities in Titus have been identified over the last four years they have not been addressed. Secondly, by using a new process for every user, a significant amount of processing time is required to context switch between serving each user, and this would have to happen many times per second with thousands of concurrent visitors. The authors of OKWS already stated that such a setup would be impractical at scale due to the huge number of potential site users. Of course, this critical analysis does not take in to account the fact that Titus is a much more recent project than OKWS and processors are more powerful now than ever, so a direct comparison could reveal better performance than one might expect.

Whilst researching Titus, I also came across an article [33] written by Ayer far prior to Titus's release. In this, he explains how one might create a small shim over OpenSSL's RSA private key functions in order to redirect private key operations over a socket pair to a very small request handler, which then sends responses back immediately via a second socket pair for use in the handshake. Although Nginx sets up private keys differently to how Ayer describes due its support for having many private keys loaded in parallel for many parallel websites, this research proves that it is possible to modify OpenSSL to achieve the separation this project is aiming to achieve.

2.5. Related Work Conclusion

The work done to privilege-separate web servers has been approached in several ways already, but so far there appears to be a crucial overlap in all the approaches covered: privilege separation is achieved by using separate memory, usually by splitting a program into multiple processes using `fork()`. Even the `sthread` idea introduced in Wedge which is positioned as a secure alternative to threads has a similar overhead in most use cases to that of `fork()`.

Therefore, there is an opportunity to investigate alternative memory protection measures, such as Intel MPK, which can work within a single process's memory. If MPK can yield acceptable isolation, it would be possible to avoid the overhead of a full context switch to a whole new process, which includes the loading of a new page table, each time sensitive data needs to be used.

There is another more subtle, but very real, difficulty from a Systems Administrator's point of view, and that is that to get the better security and key protection offered by some of these projects, one must deploy a new web server such as Titus. Adopting Wedge with the best performance means taking this one step further and even patching the Linux kernel. The best way to gain adoption of security measures is to make technology *secure by default*, which means building security measures into products that people already use and enabling them by default, rather than building new security features into new systems and expecting users to make a special effort to switch them on. Due to the time constraints of an undergraduate project, and to make this project as applicable as possible to the real world, the focus of this project will be on protecting the private key of a web server that is already well known, of which Nginx is one. Nginx relies on OpenSSL for encryption, meaning that some of what has been learned from the research so far can be applied to this project too. Furthermore, if the results from this project are good, it could be possible in the future to port the project to other web servers that use OpenSSL, such as Apache [26].

3. Project Constraints and Scope

With privilege separation being such a broad topic and with so many ways it can be done, it is important to define the scope of what this project will aim to achieve so that the benefits can be properly benchmarked and evaluated.

3.1. Handshake and Certificate

As this project is designed to advance the current state of web server security, only handshakes using the latest version of Transport Layer Security (TLS), which is 1.3, will be supported. It is defined by the Internet Engineering Task Force (IETF) Request for Comments (RFC) 8446 [17]. Older versions of TLS and its insecure predecessor, Secure Sockets Layer (SSL) will be disabled, and the web server will be configured to prevent the handshake from completing if the client demands any version of TLS below 1.3, or any version of SSL. Paired with TLS 1.3 will be a modern cipher suite as per OpenSSL's documentation [34, 35], and X25519 with Diffie-Hellman key exchange will be expected of all connections to enforce perfect forward secrecy. This is currently the most secure handshake method that OpenSSL supports at the time of this project's publication.

The certificate that will be loaded into Nginx to prove the server's identity will be a 2048-bit self-signed RSA certificate. Although RSA is no longer supported for the generation of TLS 1.3 session keys or protection of the handshake, RSA certificates are still supported for proof of server identity. It is the private key for this RSA certificate that will be protected in this project. In order to keep the project within the constraints of an undergraduate project, only RSA private keys will be supported; other key types such as ECDSA [36, 37] will not be covered.

This project's scope is initially limited to just the server's RSA private key as this is the only long-lived secret that the server requires and should therefore protect. There are also session keys that are generated by the client and server to exchange data securely as part of the TLS 1.3 handshake, but I will not attempt to migrate these to protected memory as part of this project as the specification allows for perfect forward secrecy already. Therefore, if an attack were to be performed, only data from any active sessions at the time of the attack could be stolen by the attack, and without the private key it would still be impossible to *impersonate* the server as part of a Man-in-the-Middle (MITM) attack [19] against any new clients.

3.2. Web Server

Due to the differences in how individual web servers are built, this project will focus on Nginx. The project will not attempt to make code portable or compatible with other web servers such as Apache or Internet Information Services (IIS) on Windows. Given that Nginx is fully open source and has a market share of over 25% of public-facing Web Servers according to Netcraft [21], it is a suitable candidate for this research as it is open source and well-respected.

Furthermore, Nginx relies on OpenSSL for its TLS handshaking and private key management. Nginx and OpenSSL are both written entirely in C, which means that the new extensions to `mprotect` that work with Memory Protection Keys [22] will be directly callable within the code that already exists, without having to create any complex bindings between programming languages.

This project will use the latest mainline version of Nginx at the time of this work (1.15.8) targeting the latest stable code from the 1.1.1 branch of OpenSSL. They will both be built with all debugging enabled for development purposes, and all debug features will be disabled for testing so that the builds mimic production usage as far as possible.

TLS 1.3 has been supported by Nginx since version 1.13.0 [38] built with OpenSSL 1.1.0 [34], so the versions chosen will be compatible with the constraints chosen.

3.3. Operating System

Although Nginx and OpenSSL are supported on Windows, macOS and Linux, this project will be constrained to Linux. There are a number of reasons for this, including the added complexity of programming with portability in mind, but the main reason is that there is no API for Memory Protection Keys available for Windows or macOS, as of the time of writing.

Two distributions of Linux will be used in this project. The first, Ubuntu 18.04 LTS installed from the Windows Store running under Windows Subsystem for Linux (WSL) [39], will be for rapid local development. Although MPK is not supported under the Windows kernel, the code can still be tested and built in this environment. Process-based privilege separation will function as expected under WSL, but without the right hardware and OS support the protection offered by MPK will not actually work; however, the code will still build and run. The Protection Keys API ensures that the behaviour in the absence of MPK support is defined by issuing a dummy `-1` protection key [22, 40] which will be accepted by the protection keys system call interface but not do anything useful.

For testing on real hardware, Ubuntu Server 18.10 will be used as it is the first Ubuntu release after 18.04 that includes OpenSSL 1.1.1 by default in the package repository; at the time of writing OpenSSL 1.1.x has not yet been backported to 18.04 meaning that standard tools like `cURL` that are needed for testing later do not have TLS 1.3 support without building and installing from source.

Note that although development will be done on my local machine under WSL, the versions of the code that are tested and benchmarked will be built on the real target hardware. My local test environment will not be used for statistics.

3.4. Side Channel Attacks

This project will make no attempts to protect against hardware- or kernel-level side channel attacks that can be launched from user mode, such as Meltdown [41] or Spectre [42]. The project will therefore work on the assumption that any arbitrary code executed in the event of Nginx being vulnerable would be restricted to Nginx's privileges after the setup procedure in which Nginx already relegates itself to the lowest necessary privilege level. We will restrict further to the case of Nginx running on a Linux server with all known speculative execution attack mitigation features, such as retpolines [43], updated and enabled.

In theory, any privilege separation method in which the private key is still stored *somewhere* in the web server's memory could be rendered useless if a side channel attack providing access to all memory could be launched from a user-mode exploit. Therefore, it makes sense in this project to put this possibility aside as it would be otherwise necessary to store the private key on either separate hardware or even a different computer altogether which is outside the scope of this project. Storing a private key in separate secure hardware is already a solved problem in the form of Hardware Security Modules (HSMs) such as Amazon's CloudHSM offering [44].

3.5. Scope and Aim

Privilege separation by migrating the RSA private key of a server certificate to a separate process was already achieved in Titus [45]. However, as this was not done with Nginx, the first aim of this project will be to replicate Titus's private key separation in Nginx. Once this works, I will then use the same versions of Nginx and OpenSSL, but instead keep the private key in one process as per the original Nginx architecture and then migrate all private key operations to a separate thread dedicated to encryption. With usage of the private key isolated to this encryption thread, MPK can then be used to protect the sensitive memory from being accessed by the main, Internet-facing thread.

Once Nginx has been built with both process-based and MPK-based privilege separation, I will build three versions, all of which will have debugging and messages disabled: one "vanilla" control build with no privilege separation functionality, one with process-based separation and one with MPK-based separation. With these three versions, it will be possible to test the overhead of each type of protection and assess whether it is useful and fulfills the requirements. It will also be possible to reason about the security and performance implications of MPK in place of a more traditional process-based approach to privilege separation.

3.6. Test Hardware

In order to be able to fairly compare the three implementations, testing will be performed on identical virtual hardware, with test load generated by a client virtual machine within the same data centre as the server. The test load will be created by a separate virtual machine to the server so that kernel scheduling does not skew the results, and to avoid the problem of load generation consuming resources required by Nginx to service the requests. Furthermore, in real life most web requests come from different machines across the world so it would seem nonsensical to load test a machine by connecting back to `localhost`.

3.6.1. Available Cloud Hardware. The Intel MPK functionality is available, as of the time of writing, only on Intel's Xeon "Scalable" range of Processors from the Skylake-SP family of server chips. I examined and tested offerings from both Microsoft Azure [46] and Amazon Web Services (AWS) [47] to determine which machines would support Intel MPK and reached the following results:

TABLE 1. CLOUD VIRTUAL MACHINES AND SUPPORT FOR INTEL MPK

Provider	Service	Processor	MPK Enabled	MPK Functional
Azure	F2s_v2	Intel Xeon Platinum 8168	Yes	No
AWS	t3.nano	Intel Xeon Platinum 8175M	Yes	Yes
AWS	t3.small	Intel Xeon Platinum 8175M	Yes	Yes

The reason why virtual machines with MPK functionality enabled and built into the kernel on Azure do not have MPK functional remains unclear. Machines from both providers have their default kernels built with the `CONFIG_X86_INTEL_MEMORY_PROTECTION_KEYS`

[22] option enabled, so it is possible that the issue may lie with the Hyper-V hypervisor used by Azure not properly passing the functionality through. However, I did not conduct further testing to fully understand why Azure does not make this kernel functionality available given that the `T3` range of AWS virtual machines do have MPK functional.

It is possible to quickly confirm whether a machine has MPK support enabled by running `cat /proc/cpuinfo` at the terminal of a machine and checking for the CPU features labelled `pku` and `ospke`; the only machines I tested which had these features were available on AWS. These two flags are exposed via the machine's processor's `CPUID` instruction; parsing of this data is handled by `features.h` in the Linux source code [48] and exposed via `/proc/cpuinfo` to allow users to quickly check which processor features are available for use.

3.6.2. Chosen Hardware. In order to reduce the impact of processor scheduling, two separate virtual machines will be used for each test, each of which will have a separate dedicated hyperthread core from an Intel Xeon processor. The instance size to be used is `t3.nano`, an offering from Amazon that was tested to have MPK functional. The first machine will be dedicated to hosting the Nginx build under test and the other will generate varying amounts of load as per the Performance Analysis Plan. In order to reduce the error of network latency, the machines will be set up to be within the same Virtual Private Cloud (VPC) and Availability Zone [49]. While this does not guarantee that the machines will share the same physical host, it does mean that they can at least be placed within the same data centre with a rapid network link between them.

4. System Design

4.1. No Privilege Separation: The Control

The first build of Nginx is a scientific control. Each of the experiments with privilege separation enabled are tested in the same way as the control, and the same versions of the Nginx and OpenSSL source code are used. The purpose of the control is to identify the baseline performance of Nginx on the same test hardware as the privilege separated builds; this means that any performance overhead from the extra security can be evaluated without having to account for hardware or version differences. This control should be considered to be the basis of the next two builds, and therefore every decision made for the control should be assumed to be identical for the next two builds, unless otherwise stated. Each build of Nginx uses the same build options for both debugging and testing to keep the tests consistent, and these are listed in Appendix A.

Additionally, the handshake works exactly as per the TLS 1.3 specification [17] with no changes. Even with privilege separation in place, the handshake from the perspective of the client is identical, with the only changes being how OpenSSL and Nginx internally perform the server identity data signing with the server's RSA Private Key, and this should be transparent to clients.

Finally, to keep Nginx's behaviour from changing across runs, the same configuration file and RSA keypair are used across all three builds. The configuration used is provided in Appendix B.

4.2. Classic Privilege Separation: Two Process Approach

The traditional approach to privilege separation is to ensure that the data to be protected is not available in memory to the program that utilises the data, making all usage of the protected data (in this case the RSA private key) indirect. In this initial design an Nginx Worker thread is split into two processes, via the `fork()` system call, so that we have a process running in parallel ready to receive encryption requests. Only the encryption process has access to the private key, and it does not itself face the Internet or handle client requests. The source process scrubs the RSA private key from memory after the `fork()` so that, by the time the process is ready to respond to requests, the private key is not in the memory of the internet-facing process.

This division means that, should the worker thread be penetrated by a previously unknown remote code execution vulnerability, there would be no way of extracting the private key even if its entire address space were to be scanned by the adversary. Of course, this relies entirely on the adversary not being able to access kernel memory, as the full address space including that of other programs is mapped there, but as Nginx reduces itself to an unprivileged process as part of the startup process this is not a scenario that this project will attempt to protect against. The architecture of Nginx prevents this, provided that no privilege escalation vulnerabilities are present in the Linux kernel at the point at which Nginx is hacked. Furthermore, if Nginx is started as root and the certificate and key files are owned by root, the files will be unreadable even to Nginx once the setup process has been performed and Nginx has reduced itself down to running as a lower privileged user. Therefore, the only place where the private key will be accessible to Nginx is in the second encryption process.

In order to transport encryption requests and cipher text outputs between the two processes, a Remote Prcedure Call (RPC) interface is in place. The two processes can therefore communicate using two POSIX [50] socket pairs, which allows bidirectional sharing of bytes. Every RPC in each direction has a common header that contains a packet type identifier, a unique request identifier and the length of the rest of the packet. As the RPC interface is designed to be as generic as possible, it could be extended in the future to protect other types of data such as session keys.

A timeline of how the new process is set up and which RPC messages are used to fulfil a request is shown in Figure 6, and a key is provided in Table 2. There are two procedures shown in this diagram. The first is the setup procedure, and the second is the procedure by which a TLS handshake request is actually serviced by Nginx. Note that the diagram deliberately leaves out additional processes that Nginx may start, such as caching processes [51], as they are not relevant to this project. Furthermore, we are only considering the first stage of the TLS Handshake in which the `ClientHello` is sent by the client and the `ServerHello` packet is sent back by the server. We do not consider the transmission of data or new TLS 1.3 features such as "Early Data" [17] in this diagram as they are not relevant to the overall architecture.

Note that the steps in the diagram that start with `S` are for the Setup procedure, and tasks that start with `R` are to service a Request. Steps happen in the order defined by the numbers (i.e. Step `S1` happens before `S2`), and any steps that have small letters after can happen in parallel across the two processes. Given that scheduling is unpredictable, it is perfectly acceptable that any permutation of steps S4a, S4b, S5a and S5b happen so long as each process's step `a`

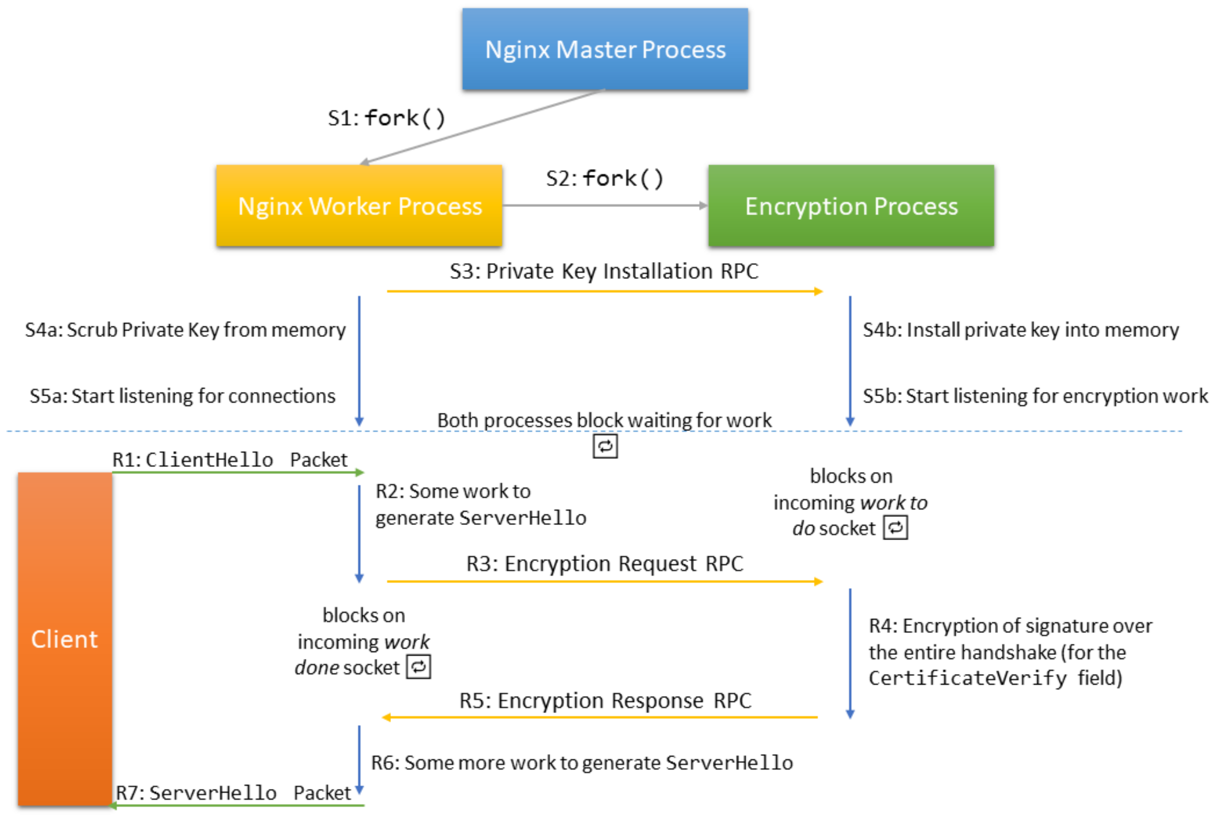

Nginx Master Process
S1: fork()
Nginx Worker Process
S2: fork()
Encryption Process
S3: Private Key Installation RPC
S4a: Scrub Private Key from memory
S5a: Start listening for connections
S4b: Install private key into memory
S5b: Start listening for encryption work
Both processes block waiting for work
R1: ClientHello Packet
Client
R2: Some work to generate ServerHello
blocks on incoming work to do socket
R3: Encryption Request RPC
blocks on incoming work done socket
R4: Encryption of signature over the entire handshake (for the CertificateVerify field)
R5: Encryption Response RPC
R6: Some more work to generate ServerHello
R7: ServerHello Packet

Figure 6. Process-Based Privilege Separation Diagram

TABLE 2. PROCESS-BASED PRIVILEGE SEPARATION DIAGRAM KEY

Feature	**Meaning**
Blue arrows	Time passing; work being done
Orange arrows	RPC message transmission
Dashed blue line	Any amount of time; both processes are waiting for work
🔁	Process blocked by other process doing work
Green arrow	Network packets
S steps	Setup process stages
R steps	Servicing a request (i.e. a handshake with a client)

happens before its step `b`.

The first stage of the setup process is the `fork()` of the worker process, which is handled by Nginx's verbatim startup code that runs after the configuration has been parsed. The change from the norm here, however, is the second `fork()` that occurs from the worker: this becomes the RSA encryption process.

In order to build a proof of concept for this dissertation, the TLS certificate configuration code is left verbatim to set up the RSA certificate and private key as normal. My modifications come after this setup. Firstly, the full RSA structure is dumped into an RPC packet. Once the RSA structure, including the private key, has been completely dumped, the private key is scrubbed from memory but the rest of the RSA structure is left in place so that public key cryptographic operations can still be done in the worker process. Once the `fork()` has happened, the dumped RSA structure is sent via RPC where it is set up into a new `RSA` structure named `priv_sep_rsa` to be used later.

Leaving the rest of the RSA structure in place, but without the private key elements, avoids the further overhead that would be incurred if public key decryption were to be delegated to the encryption process too. As the point of this project is to protect a piece of sensitive data, such as the private key, there is little point in privilege separating a public key as the cryptography system relies on this key being public anyway. It would just add additional overhead to every handshake with no tangible security benefits.

When a connection comes into to Nginx from the Internet it is allocated to the worker process which sets up a state machine to manage the handshake. Once the handshake requires a signature to be encrypted by the private key by calling the `RSA_private_encrypt()` function (and in turn `rsa_ossl_private_encrypt()`, where many of my modifications are), the data to be encrypted is sent via RPC to the encryption process and the worker process is put to sleep until a response is ready. On pushing data to the work socket, the encryption process is woken up via the `select()` system call [52] and encryption is performed using the private key. Once a response has been generated, it is serialised into another RPC packet that is sent back to the worker process to be used in the building of a packet, such as a `ServerHello`, that requires a signature for the handshake.

4.3. Thread-Based Privilege Separation: Memory Protection Keys

This design is where the project goes beyond the existing research with respect to web server privilege separation. In previous privilege separation implementations, the only way to ensure that memory was not available to a thread was to move the data to be protected to separate memory altogether; however, with MPK it is now possible to make memory unavailable to specific threads but readable in others. The protection key interface in modern releases of the Linux kernel and `glibc` [22] provides a `pkey_mprotect()` function that allows a thread to control access to memory pages mapped with `mmap`. A memory page can be set either as unavailable, available for reading only or available for reading and writing (the default).

To make use of this, we exploit the fact that Nginx workers are both single-process and single-threaded in nature which rely on `epoll` [53] and a callback interface to handle incoming connections, as it means we can create a second thread using the standard POSIX Threads (`pthread`) API [29, 50] within each worker process to handle encryption. It is then possible to deny all access to the RSA private key elements from within the main worker thread using a memory protection key obtained via `pkey_alloc()`. In the encryption thread we manually enable access to those same memory regions. The reason for this is that when `pthread` instigates a `clone()` on the source thread, the protection keys interface is very careful to ensure that memory is not accidentally made available [40]. Therefore, it is necessary to manually set the access permissions in both threads to ensure that there are no scenarios in which private key memory is either unavailable or only partially available to the encryption thread.

As all threads within the same process live in the same memory space, they can all see all the memory within that process, with the exception of memory that requires root permissions to read (such as kernel memory) and any memory locked with a protection key. This means that the interface between the threads is far simpler than the one between processes as data does not need to be serialised and sent between them; instead, the worker thread just needs to wake the encryption thread up when work is available at a known memory location. Once the encryption thread has done its work, it can then wake up the main worker thread in the same way and go back to sleep itself ready for more work.

4.3.1. Setting Up the MPK Thread. The setup process for the MPK thread is a little more complex at first than for the process-based separation, but once configured it is far simpler to work with as there is no RPC interface required and all the operations happen within the same virtual memory. The setup procedure is as follows:

1) The Nginx Master Process uses `fork()` to spawn a worker process, as per the pre-existing Nginx architecture.
2) Within the worker process, the RSA structure is duplicated into a new global, `priv_sep_rsa`. The new structure is configured in the same way as the original, with one exception: the private key elements are allocated memory using `mmap` instead of `malloc` so that `pkey_mprotect` can be used on them later.
3) The initial RSA structure is scrubbed of the private key, leaving it intact but with only the public key components set up.

4) A mutex and two condition variables (indicating a pending encryption request and a pending encryption response) are initialised.
5) `pthread_create()` is used to create the encryption thread. This thread exclusively handles encryption requests using the private key in `priv_sep_rsa`, and therefore the encryption code is wrapped in a `while(1)` infinite loop. Each request is handled sequentially and synchronously.
6) `pkey_alloc()` and `pkey_mprotect()` are used to protect `priv_sep_rsa`'s private key components from being accessed from the main worker thread, so that only the encryption thread can perform encryption operations.
7) `pkey_alloc()` and `pkey_mprotect()` are used in the encryption thread to clear any inherited protection on the private key components within the dedicated encryption thread only, so that it is able to perform its task.
8) The main worker thread continues to start up as normal and prepares itself to service requests from the Internet.

4.3.2. Servicing Encryption Requests in the MPK Thread. Instead of RPCs and the the `select()` system call, this design uses a shared mutex between the two threads and two condition variables to pass execution between the two threads as work is needed from each one. The design of this handover process between threads is shown in Figure 7. Note that instead of RPC messages being sent over a socket, data to encrypt and subsequent ciphertexts are stored in global variables in memory with the condition variables mediating when each thread should do work. This is essentially a limited, bidirectional implementation of a producer-consumer setup between two threads from a typical Computer Systems course [54].

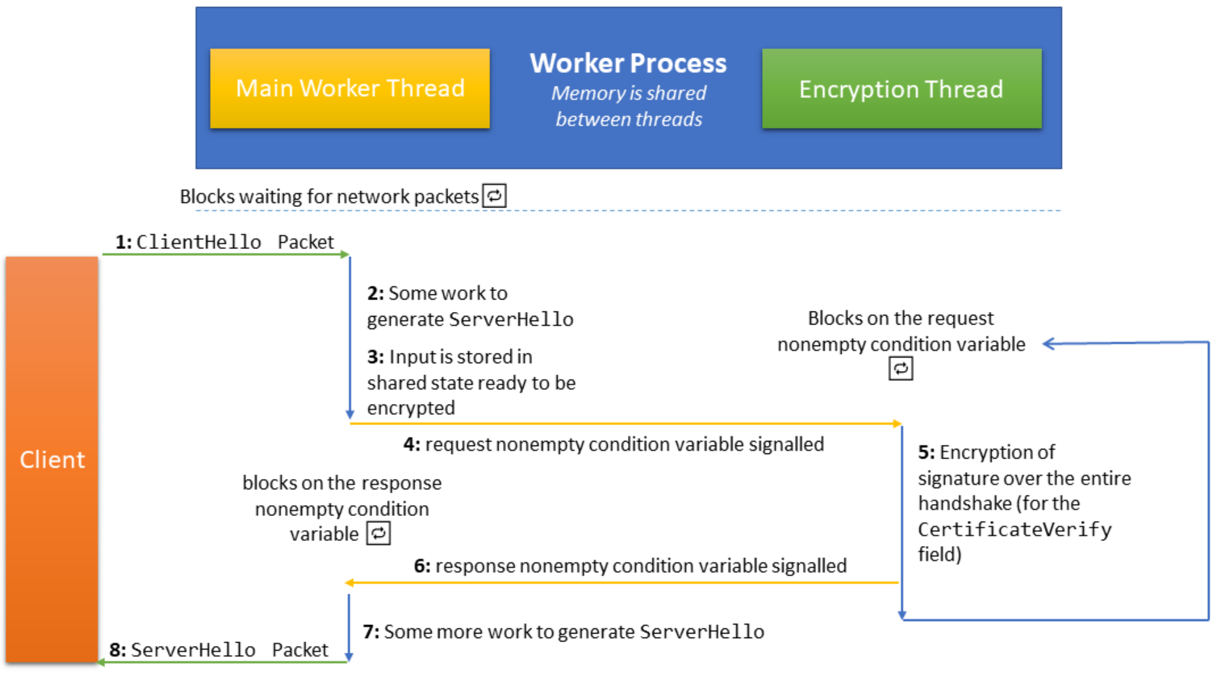

Main Worker Thread
Worker Process
Memory is shared between threads
Encryption Thread
Blocks waiting for network packets
1: ClientHello Packet
Client
2: Some work to generate ServerHello
3: Input is stored in shared state ready to be encrypted
Blocks on the request nonempty condition variable
4: request nonempty condition variable signalled
5: Encryption of signature over the entire handshake (for the CertificateVerify field)
blocks on the response nonempty condition variable
6: response nonempty condition variable signalled
7: Some more work to generate ServerHello
8: ServerHello Packet

Figure 7. Thread-Based Privilege Separation with MPK Diagram

The main thread is barred from giving the encryption thread any more work until the encryption is completed, and the encryption thread is barred from performing any encryption work until the result has been collected by the main worker thread. This avoids race conditions in which data could be lost or corrupted by multiple threads being out of sync with one another. Furthermore, to avoid the undefined behaviour case of two threads writing to the same memory at once, only the main worker thread can write to the variables storing work to be done, and only the encryption thread can write to the memory set aside for the cipher text.

5. Performance Analysis Planning

With two modes of privilege separation and a control in place, the next stage is to gauge how feasible privilege separation of server private keys is, based on how much performance overhead the security measures have. Given that the TLS 1.3 specification[17] uses the certificate only in the initial handshake, and not in subsequent connections due to shared session keys already existing between the server and the client, the performance analysis will focus only on the initial handshake.

Since this project is an experiment in privilege separation methodologies, stated here are some initial hypotheses regarding what I expect to see in the results, and later the actual results are compared to the expectations and evaluated to determine whether MPK is a suitable approach for software privilege separation.

5.1. Initial Hypotheses

1) The control build of Nginx should have the best performance, as there is no thread or process switching involved and the private key is directly available in memory.
2) The MPK build of Nginx should have better performance than the process build, as a context switch from thread-to-thread is less complex than a process-to-process switch. Additionally, there is no need to send the data to be encrypted over a socket pair (and then wait for the ciphertext to be sent back) in the MPK build, unlike in the process build which requires bidrectional Inter-Process Communication.
3) The time for an RSA cryptographic operation itself should remain unchanged, whether the operation runs in the main worker thread, a separate thread under MPK or a separate process.
4) The majority of the time taken to carry out a cryptographic operation will likely be consumed by the RSA operation itself, and any context switching should carry a small percentage of this time as overhead.

The analysis will also take into account that the work done so far is a sub-optimal proof of concept, and there are many ways to improve upon the architecture of both experiments in the future to have a better understanding of how the results might look in production. These optimisation options are discussed in the Future Work section.

5.2. Test Methodology

The test procedure is divided up into two categories: *microbenchmarks* and *macrobenchmarks*. The former are a series of tests of the RSA cryptography code timed with an appropriate highly precise system event timer, and the latter is a series of tests of the system as a whole to determine the overall effect that each of the privilege separation methods have on Nginx's handshake performance. Whilst the microbenchmarks are focused on very small blocks of code, including my changes for privilege separation purposes, the macrobenchmarks aim to provide an indication of the overall web server effects of this code, and how it might affect its ability to handle load. This information can help to estimate the scale of additional investment of resources that would be required in the real world to invest in this privilege separation technology.

In all tests, each virtual machine involved will have no system swap enabled to avoid any possibility of performance being lost to pages being swapped in and out of memory.

5.3. Microbenchmarks

5.3.1. The Time Problem. Amazon Web Services (AWS) Elastic Compute Cloud (EC2) virtual machines have a number of clock sources available, each of which can provide some timing measurements. As the tests will be on a few lines of code at a time, it is very important that the clock is as accurate and precise as possible. Therefore, each available clock source should be checked for accuracy and precision [55, 56]. Each time a source is changed, `clockperf` [57] will be used to ensure that the timer is functioning as expected. The results from successive runs of `clockperf`, along with advice from reputable sources, can then be used to choose a reliable timer.

The reason behind this initial checking step is that there are documented instances of the system clock on EC2 virtual machines not reporting reasonable measurements depending on the hypervisor in use. For example, machines that use the Xen hypervisor can have up to a 77% overhead on the system call to get the current time on EC2 [58]. Whilst the instance type that will be used in these benchmarks, the `t3.nano`, runs on a version of the KVM hypervisor which should have a much lower overall overhead than Xen, the actual hypervisor used is in fact an AWS KVM fork known as *AWS Nitro* [59]. Whilst Nitro is somewhat documented, at the time of writing AWS do not address which event timers can be relied upon [60]. Therefore, it is crucial to confirm that the hardware timers are being appropriately exposed through Nitro/KVM's hardware interface to Linux, and therefore properly functional, before they are actually used for precise benchmarking. We do not know what changes, if any, have been made to KVM's clock functionality in Nitro.

Microbenchmarks will be run from the same virtual machine as the server with a script to kill and relaunch Nginx between every test to ensure that any lingering memory leaks do not impact on the results.

5.3.2. Choosing a Timer. The timers available on a `t3.nano` instances are as follows:

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
kvm-clock tsc acpi_pm
```

The default clock on the instance is set to the `kvm-clock` which is a paravirtualised clock designed to work around the constraints of keeping time in a virtual machine. For example, interrupts are *injected* into virtual machines as, with virtual hardware, there are no physical interrupts to trigger [61]. Other clock options available are `TSC`, the Timestamp Counter, and `acpi_pm`, the timer provided by the power management hardware.

TSC can be unreliable in certain conditions, such as when the processor's clock speed changes, but given that the processor in use has the `constant_tsc` feature in its CPUID, we will not discount TSC off the bat. The ACPI Power Management Timer, on the other hand, has a fixed frequency [62] provided by the power management hadware, and according to *Bharadwaj* is more reliable [63]. However, it takes longer to query than the TSC so the actual measurement itself will add additional overhead.

The full `clockperf` results, along with the criteria for choosing a timer, for each hardware clock option are in Appendix C. Based on these results and criteria, the `hvm-clock` will be used as the system timer and the TSC will be read in code to get time values. The code to gather these timings is provided in Appendix E.

5.3.3. Benchmarks to Perform. The first microbenchmark is the time that an individual RSA encryption calculation takes to perform. This is useful, because if the code is working as expected it should be, within error, a constant across each of the three experiments. This means that by subtracting this time from the time that a full encryption operation takes within privilege separated handshake code, it'll be possible to calculate the overhead to pass the request over to another process or thread, and then get the result back again.

The second microbenchmark will be over the whole encryption operation, from the initial call to `rsa_ossl_private_encrypt` to the moment it is ready to return a response. The relationship between this microbenchmark and the first is shown in Figure 8. The blue

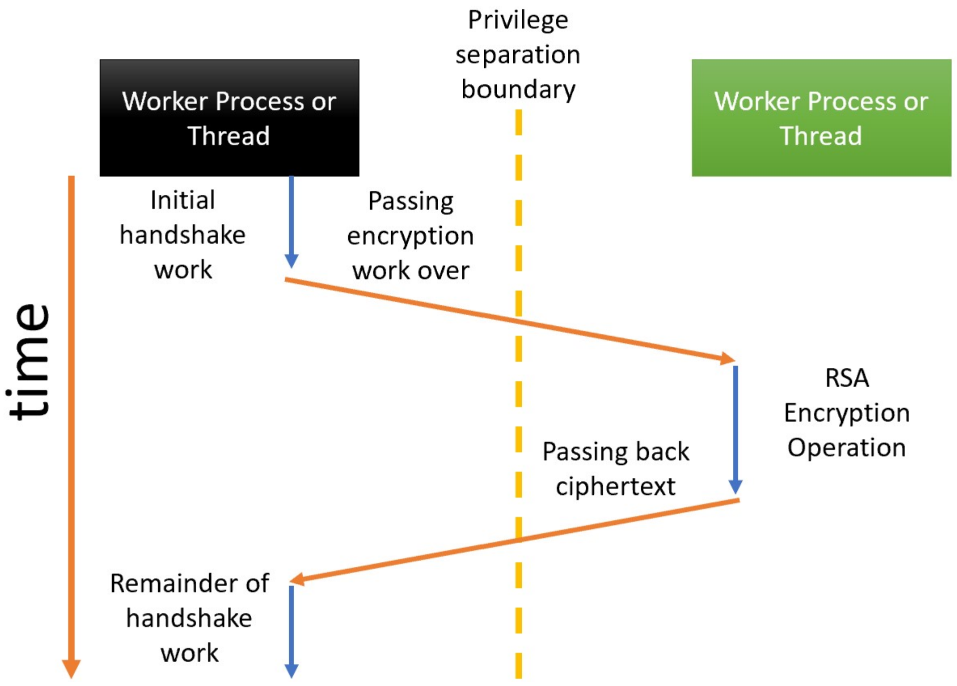


Figure 8. Timing of Privilege Separation Overhead

arrows show timings that are expected to be constants throughout the experiment, and the orange arrows show timings that are expected to differ between the Nginx builds. If the first microbenchmark produces a constant for the encryption operation as expected then the total privilege separation overheads will be defined as follows:

$$Overhead_{Process} = t(encrypt)_{Process} - t(encrypt)_{Control}$$

$$Overhead_{MPK} = t(encrypt)_{MPK} - t(encrypt)_{Control}$$

The TSC value can be accessed by issuing the `RDTSC` processor instruction, then reading the 64-bit value out of two registers [64] as an `unsigned long long` value. However, in order to ensure that out of order execution does not cause the timings to be incorrect, the method described by *Paoloni* in their Intel White Paper will be used to combine the `RDTSC`, `CPUID` and `RDTSCP` instructions. [65]. This is because the `CPUID` and `RDTSCP` instructions force a clear of the processor's pipeline which allows the benchmark to account for all instructions, including those which may under normal circumstances still be in the pipeline at the point at which a modern processor with out-of-order execution support may be taking the TSC reading.

By taking a TSC sample before and after an operation has taken place, it is possible to calculate the time that the code to perform that operation took to run, because the value that is provided by taking the difference of the two TSC readings is the number of processor clock cycles that have executed between them. Therefore, dividing the TSC difference by the processor's default frequency, which is usually the processor's highest non-Turbo Boost frequency, gives the wall clock execution time of a block of code.

The samples are logged to a CSV file named according to the type of privilege separation ("vanilla", "process" or "mpk"), with each row containing two columns of data: whether the measurement is for the RSA cryptographic operation or whether it is for the entire cryptographic operation including the privilege separation handover, and the amount of time the operation took. The *five* microbenchmark binaries are built according to the following specification:

- **Control**: Code compiled once with no privilege separation enabled, and a benchmark taken of the `rsa_ossl_private_encrypt` function.
- **Process**: Code compiled twice. Firstly, the `rsa_ossl_private_encrypt` function in the second process will be benchmarked to ensure that the RSA cryptographic operation takes approximately the same amount of time as the control (as per Initial Hypothesis #3). Secondly, the entire function will be be benchmarked from the perspective of the Nginx worker process. This timing includes includes the RSA operation itself as before, along with the transfer of the input into the encryption process and the fetching of the ciphertext from the completed work socket. The second measurement minus the first measurement will calculate the total overhead of the privilege separation. The measurements are separated to ensure that the time to take the RSA measurement and write it to the CSV file is not taken into account when benchmarking the entire operation (as both processes would be blocked waiting for this operation to happen, skewing the results and making the overhead appear to be much higher as a result of this I/O operation).
- **MPK**: Code compiled twice. Firstly, the RSA measurement will be taken as before, but from within the instance of the `rsa_ossl_private_encrypt` function running within the encryption thread. The result will be written to file immediately before the function returns to the thread's worker loop. Secondly, the overall impact measurement for MPK will be taken from the worker thread, accounting for the entire cryptographic operation as well as the time taken to pass control between the two threads using the mutex and condition variables. Although both measurements could in theory be taken

from within the same binary, the test environment is single-core so it is possible that the calling thread might not get scheduled to resume and take the second measurement until the RSA one has been written to the CSV, again skewing the results and/or making them inconsistent due to the additional I/O operation.

5.3.4. Statistical Approach. Each of the microbenchmarks will be run 500 times. Due to the complexity of taking precise microbenchmarks, and nuances in how code is compiled and instructions are scheduled, a high number of test runs are needed. From these test results, there are two analytical phases. Firstly, the average RSA encryption operation time for each Nginx build is calculated by averaging the values and ensuring that all three experiments have the same RSA operation time within an acceptable degree of error (around 1-2%). This is a sanity check over the whole test methodology to ensure that we see the RSA cryptography operation itself take the same amount of time in each of the three experiments as per Initial Hypothesis #3.

The second phase is the overhead calculation. Every full time measurement for each of the privilege separated experiments will be calculated, and taken away from the averages will be the average RSA calculation time as calculated before. This will give an absolute overhead for each type of privilege separation, and therefore permit concluding which of the two methods has the lowest overhead. From this, it will be possible to suggest whether MPK might be a reasonable privilege separation method when combined with the results of the macrobenchmarks.

5.4. Macrobenchmarks

The macrobenchmarks are designed to test the *overall impact* of the privilege separation on the web server's ability to perform its task well. As mentioned, Nginx have bold claims about the number of requests that can be fulfilled per second [51], but these figures depend on a multitude of factors including the processor in use and the system load. In light of this, a set of overall benchmarks will be performed in which a simple webpage with just one line of text will be served, meaning that the vast majority of the processing time will be the TLS 1.3 handshake including, if enabled, privilege separated RSA signing.

Note that for each of the macrobenchmark operations, all microbenchmarks will be disabled to ensure that the overhead of gathering those metrics does not affect these benchmarks too. It is not necessary to run the microbenchmarks and macrobenchmarks at the same time, so long as the binaries are compiled in the same way and the same test hardware is used.

Macrobenchmarks are performed by making requests remotely from a second AWS EC2 instance with identical specifications to the server. There will be two measurements taken: handshake latency for one connection, and the latency with many parallel connections. The former tests, for each Nginx build, how long a fresh TLS handshake takes with no session caching, as if a new client is connecting for the first time; this is crucial to force the RSA encryption routine to run. Testing one-off, non-parallelised handshakes can be done simply using the `curl` command line tool [66] and the `--trace` option. The value that will be recorded is the time between the test's start timestamp and the timestamp of the first received data after the handshake has completed (which is always the HTTP 200 OK header). This test is done 5000 times for each build of Nginx.

The second measurement will be the 50th and 99th percentile latencies at increasing numbers of connections. This load will be generated via the load-testing program `wrk` [67] which allows multithreaded load testing with full latency output data [68]. The number of connections will start at 200, increasing by 200 each time up to 2000 connections. Each test will be run for 30 seconds so as to stay well under the Amazon's one minute limit within their testing policy [69]. The number of threads assigned to `wrk` will be two, as the `t3.nano` instances being used are both single core instances with Intel Hyperthreading technology enabled giving two logical processor threads.

The 50th percentile latency is an important metric as it shows, in the median case, how long the client has to wait to start receiving a web page, a value known as the "Time to First Byte" (TTFB). The TTFB is a key metric in internet performance measuring, and to give it some context Content Delivery Network *KeyCDN* states that any TTFB over one second "should likely be investigated" [70]. High TTFB values can also negatively affect a website's search engine ranking [71] as slow load times contribute to a significantly hindered user experience when using a website [72]. Analysing the 99th percentile too will give an indication of the extent to which the web server is struggling under high load with large numbers of connections queuing up for a slice of processing time.

Given that the test configuration for the Nginx experiments is set to serve a single static file, it is reasonable to suggest that any latency over 100ms is cause for concern given that no background processing or reverse proxying has to be done by Nginx. Furthermore, there is sub-1ms latency between the two virtual machines (server and client) due to them being situated in the same data centre; this is a scenario that is extremely unlikely in the real world where hundreds of milliseconds of latency can be seen in remote locations or when, for example, a client might be using in-flight wireless Internet. However, as the servers are set up in well controlled conditions, it is reasonable to expect significantly lower latencies than KeyCDN's $< 1s$ recommendation.

5.5. MPK Memory Protection Test

To ensure that Intel Memory Protection Keys are actually working as expected, a single test build of Nginx will be created that attempts to dump the RSA private key to `stdout` from the main thread after the protection has been put in place. The expected result is that the main thread within the Nginx worker process receives a *segmentation fault* signal, which is the signal the Linux kernel sends to a program that tries to access memory it does not have access to. Figure 9 depicts the expected behaviour.

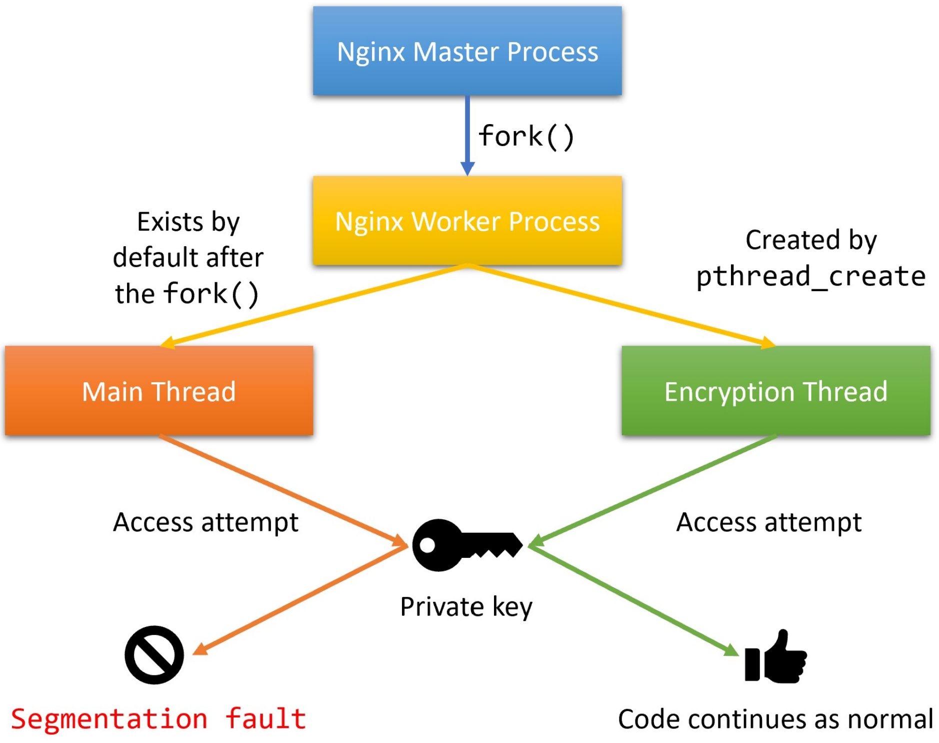


Figure 9. Expected Segmentation Fault with MPK

This test is required because, unlike in process-based privilege separation where the sensitive data is in separate memory, the thread-based approach keeps the private key in the same memory. This means that if MPK is not functioning properly or has been improperly configured then the protection may not work and the privilege separation attempt will be useless.

Checking for a segmentation fault of this nature can be difficult in programs like Nginx that use `fork()` and tamper with signals, and this is made even more complex by the use of multiple threads. Therefore, the segmentation fault will be checked for using `gdb`, the GNU debugger, with `follow-fork-mode` set to `child` [73], which will ensure that the debugger is watching the Nginx *worker* process as opposed to the *master* process that gets started first.

6. Performance Analysis Results

6.1. Microbenchmark Results

6.1.1. RSA Cryptographic Operation. As suggested in Initial Hypothesis #4, in both Nginx builds with privilege separation enabled the majority of the time for the code to perform an RSA encryption was taken up by the cryptographic operation itself, as opposed to the extra overhead of privilege separation. As shown in Table 3, the RSA cryptographic operation took almost the same amount of time in each of the three experiments. However, the $< 4\%$ overhead in each case is higher than expected, but there are possible technical explanations for this including that the privilege separation methods require either an extra process or an extra thread to be running and scheduled, as well as code to watch for the operation to be completed running in parallel.

TABLE 3. RSA CRYPTOGRAPHIC OPERATION TIMINGS

Experiment	RSA Operation Time	Difference to Baseline
Vanilla Nginx [Baseline]	1.040ms	-
Process Privilege Separation	1.072ms	3.092%
MPK Privilege Separation	1.080ms	3.828%

As one of this project's aims is to determine what extra resources an Nginx administrator may need to allocate to a service to add in this extra security measure, the worst case scenario results are provided here. They are based on the assumption that, despite every effort to fully isolate the cryptographic code for the timings in Table 3 being made, there is still some overhead added to the RSA cryptographic operation due to the wider context of the changes to the server's code and architecture. Therefore, the calculated overhead values in Table 4 include the effects of the different RSA cryptographic timings results, and this is done by taking the *vanilla* RSA processing time away from the total processing time for each of the privilege separated builds of Nginx.

The initial plan was to take an average of all 1500 total RSA cryptographic timings and use this value to calculate the overhead, but given the variations shown in Table 3, it now does not make sense to calculate the overhead in that way.

TABLE 4. FULL PRIVILEGE SEPARATION OVERHEAD

Experiment	Total Time to Run	Overhead	Overhead %
Vanilla Nginx [Baseline]	1.040ms	-	-
Process Privilege Separation	1.140ms	0.1001ms	9.621%
MPK Privilege Separation	1.105ms	0.0644ms	6.189%

These results demonstrate that, as expected, both methods of privilege separation do indeed add a small overhead but using MPK and threads is significantly more efficient than having two separate processes; in fact, the **process-based implementation is over 55% more demanding on processor time than the MPK-based implementation**. Of course, these experiments have been run on prototype code. There are some suggestions on how to improve the prototypes and experiments in the Future Work section, and this could indeed affect the performance gap.

6.2. Macrobenchmark Results

The macrobenchmarks were run from a second virtual machine hosted with Amazon's AWS environment. Both virtual machines were in the `eu-west-2b` availability zone in London, with typically under 0.3ms of latency between the instances. Each virtual machine was provisioned as a `t3.nano` instance with 0.5GB of RAM and two virtual processor threads.

6.2.1. Handshake Time. 500 sequential connections were timed for each of the three Nginx builds, and the time between the beginning of the test and the arrival of the first line of result data (`HTTP/1.1 200 OK`) calculated. The results are below in Table 5. The scripts to generate these results are provided in Appendix D.

TABLE 5. AVERAGE HANDSHAKE TIMES AND OVERHEAD PERCENTAGES (REMOTE CONNECTIONS)

Experiment	Time Per Handshake	Overhead %
Vanilla Nginx [Baseline]	3.330ms	-
Process Privilege Separation	3.527ms	5.916%
MPK Privilege Separation	3.382ms	1.562%

These results show that, whilst it adds over 1.5% to the handshake time, **MPK is nearly four times as efficient as the process-based build of Nginx.**

6.2.2. Load Testing. The load testing results are not particularly helpful, and indeed show mostly junk data. If anything, this is a testament to the efficiency of Nginx and OpenSSL, as the overhead is mostly unnoticable, but some suggestions on how to better test this are provided later in Future Work.

Figure 10 shows the 50% latency as the number of parallel connections increases from 200 to 2000. As expected, the latency does increase with the number of connections due to the extra workload placed on the server and the necessity to queue incoming requests, but in the general case, with some minor outliers aside, Nginx handles the load well. There is no indication that at this level of load any particular Nginx build is any better; indeed, the chart actually shows if anything that the the vanilla build performed the worst which does not make logical sense given that it has less processing to do per handshake.

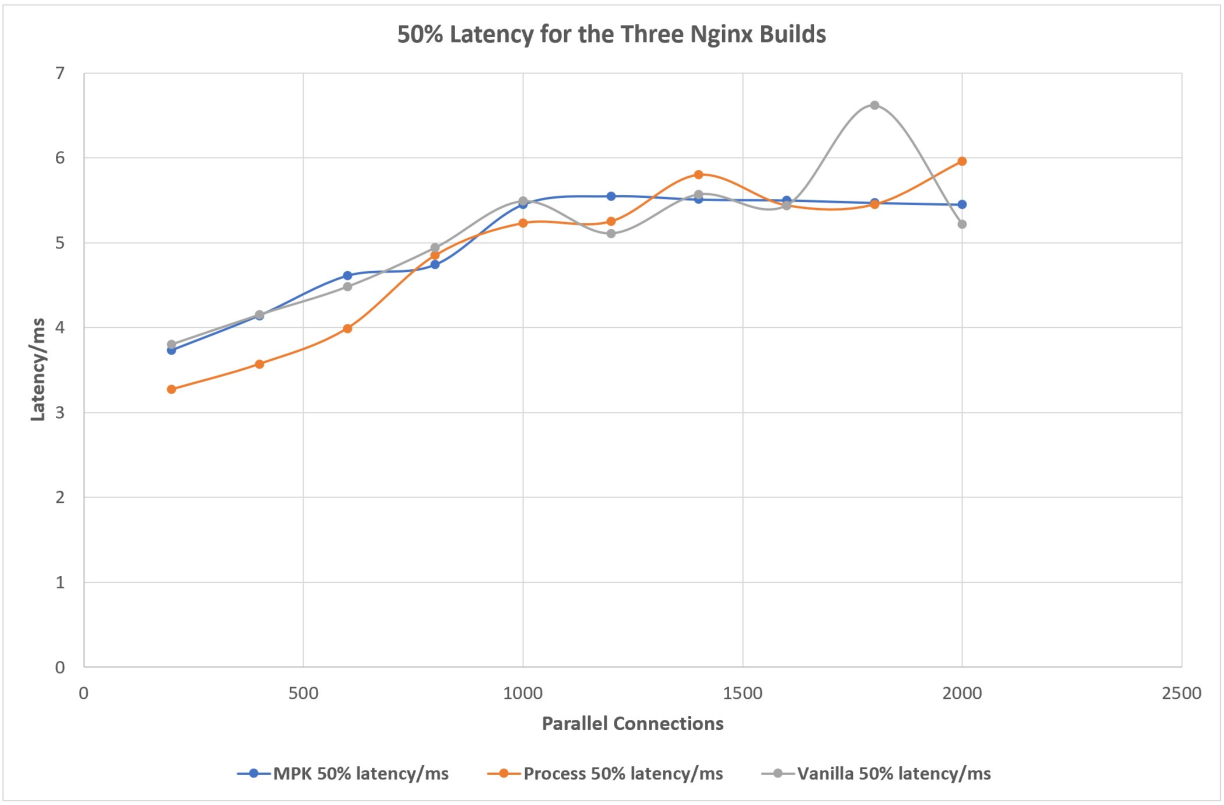


Figure 10. 50% Latency Graph

Figure 11 shows the 99% latency under the same conditions. The results here look strange at first, but given that it is the chart for the worst 1% for each connection, it is possible for a small number of outliers to dramatically affect the results. Therefore, this graph also does not yield a useful indicator of how the builds perform under high load.

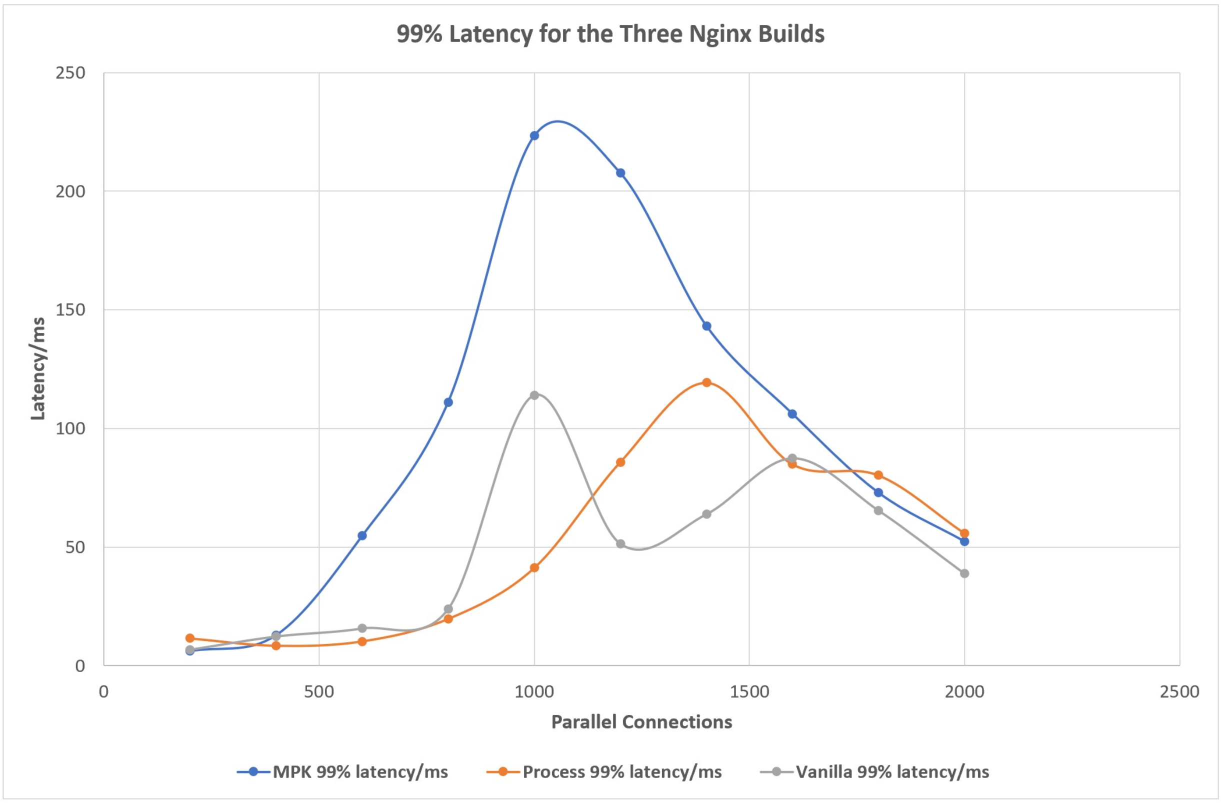
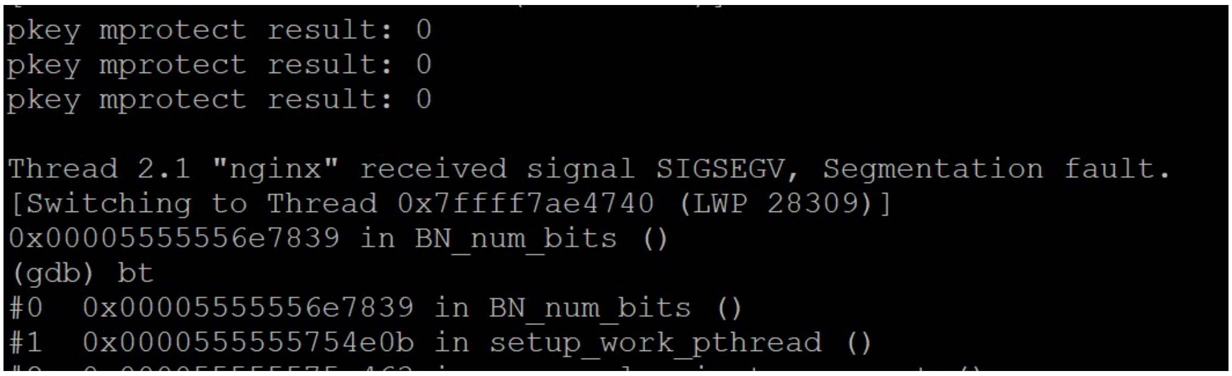


Figure 11. 99% Latency Graph

6.3. MPK Memory Protection Test Result

A test build was produced which attempts to dump the private elements of the RSA structure from memory. The result was, as expected, a segmentation fault. Proof that this code worked was twofold:

1) When the program was run without a debugger, the web server ceased to operate properly due to the failure of the main thread. Furthermore, the private key components were not written to `stdout`, indicating that the code did not manage to execute that far.



```
pkey mprotect result: 0
pkey mprotect result: 0
pkey mprotect result: 0

Thread 2.1 "nginx" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7ffff7ae4740 (LWP 28309)]
0x00005555556e7839 in BN_num_bits ()
(gdb) bt
#0  0x00005555556e7839 in BN_num_bits ()
#1  0x0000555555754e0b in setup_work_pthread ()
```

Figure 12. GDB Segmentation Fault Output after attempting to illegally access memory protected with MPK

2) When the program was run with a debugger as per the test plan, a segmentation fault was caught by `gdb` when the main thread attempted to access a private key component. This is shown in Figure 12.

7. Conclusion

This project has demonstrated the potential for a novel use of Intel Memory Protection Keys (MPK) to isolate confidential memory within a process that could be exposed to hostile code. All the previous work in this field has involved the overhead of either a `fork()`, or a similar action, to fully isolate the program into separate memory regions. This project takes this work further by using MPK and `pthread` to avoid much of that overhead and to eradicate the previously-required inter-process communication via socket pairs. Three prototypes were built: a "vanilla" build of Nginx with only benchmarking code added to assess the current state of the web server, a build of Nginx with process-based privilege separation in line with the current state of the research in this area, and a third build with threads and MPK used in place of a new process to protect the critical memory – in this case, an RSA private key.

The project assumes that in the future the web server might be hacked by a powerful adversary capable of injecting code into the remote server. Assuming that arbitrary code can be run, both methods of privilege separation should make the adversary's task to steal the private key significantly more difficult. In the case of the process-based approach, the key will not exist in the web server's worker process's memory at all. In the case of the MPK-based approach, the key will be in addressable memory but attempts to access it will result in a segmentation fault [22] as only the encryption thread will have permission to read those memory pages.

The prototypes demonstrate with metrics that MPK-based protection with threads is more efficient than the current state of the research, which suggests that perhaps further research in this area should be pursued. Adding security on a technical level has a computational cost attached, and in industry this would be considered against the probability and cost of a breach. By reducing the cost of additional security by minimising the computational resources required to implement it, there is a higher chance that work with similar aims to this project could be deployed to production web servers in the near future.

MPK is still new technology that does not yet have widespread adoption, so it is still difficult to know how this technology might be shaped in the future by Intel, or whether it will even be kept in the architecture at all if uptake is slow. Having said that, the current generation of Intel Xeon Scalable processors do make it possible to add protection to a web server's private key using MPK. There are a number of caveats and important future research avenues to consider before this work could be used to protect production workloads, and many of these are discussed in Future Work. Indeed, the code written for this project is certainly no more than a prototype and it is not ready for deployment in a production environment, nor could such a deployment be recommended in good conscience as the code has not been audited. However, as testing and deploying the code in a production environment is out of this project's scope, I would deem the project overall to be a data-supported success.

8. Future Work

8.1. Multi-Process Web Servers

This project only considers the case of Nginx running with a single worker process on a single core web server. The reality is that multiple processes can be enabled in Nginx to help it to handle high load [74] by running a worker process per core, but neither of the experimental builds are tested with this in mind. Whilst the MPK-based implementation would likely work in this mode, this is deliberately untested as the architecture of the process-based experiment is not conducive to being spun out to multiple processes, so it would not be possible to fully draw a comparison.

A potential architecture for multi-core support in the process-based isolation version would be to set up, within the encryption process, two socket pairs *per worker process* for bidirectional communication. The encryption process could then process work generated by the worker processes in a round-robin fashion using the `select()` system call to look for work to do. Completed work would then be sent back to worker processes on their respective return sockets. These worker processes would use `select()` within their event loops to trigger callback events when encryption work has been completed. This is a far more complex architecture, but it would allow benchmarking of a process-based approach against an MPK-based approach on a multi-core system with multiple Nginx processes in a fair way.

Note that such an architecture would involve deep integration between Nginx and OpenSSL, and significantly affect portability of this new security measure to other web servers.

8.2. Event-Driven Programming

In both implementations of privilege separation, a naive approach to parallelism was used. In each case, work was set up for the encryption worker or thread and the main thread was blocked until the RSA calculation had been done. This was a caricature of the architecture from before, as the vanilla code does *not* perform the RSA calculations asynchronously. However, given that threads and processes are scheduled separately, making use of Nginx's event interface could increase throughput by servicing other requests on the main thread whilst waiting for the RSA encryption to be done elsewhere, perhaps on another processor core in a multi-core environment. This could make use of spare computational resources on another core to perform encryption operations whilst the first core handles other requests that do not need RSA cryptography operations.

Furthermore, as blocking I/O system calls (such as `write()` and `read()`) were used in the process-based approach to Privilege Separation, the event loop was completely paused until the encryption operations had been completed. By moving to a combination of asynchronous I/O calls and `epoll` [53], the thread would not waste time waiting for these bytes to be copied between processes via sockets by the kernel.

Whilst the current system is likely more efficient in microbenchmarks as they were performed with one request at a time utilising very little extra code, switching to non-blocking I/O would likely improve the overall performance in cases of high load by more equally sharing the computation time between active connections. Although the implementation would be more computationally complex, it would mean ensuring that a sudden influx of handshakes would be less likely to affect the experience of clients making secondary requests (*e.g.* for extra resources) in which an RSA operation is not needed due to session key caching [17].

8.3. Auditing

Code written for security purposes should be properly audited for vulnerabilities and issues that have not come up in limited testing, such as memory leaks. Without being fully audited, this privilege separation code should not be launched into production as it may, ironically, introduce new bugs that could be exploited by hackers. Only after passing an independent evaluation and being certified as memory bug free by tools such as Valgrind's Memcheck [75] and Google's AddressSanitizer (ASan) [76] should the code be released into limited beta testing to ensure that professionals round the world are comfortable with these significant architectural changes to the security model.

8.4. Support for Other Operating Systems

At the moment this project is limited to Linux. However, both OpenSSL and Nginx are fully compatible with other platforms including Windows. I firmly believe that all security features should be available to users and systems administrators, regardless of their choice of platform.

There are some issues that would need to be addressed, however, to ensure that this code is portable across platforms. One of them is that, as of the time of writing, there is no MPK API available for Windows. Another is that the experiments are written using Linux system calls, such as `socketpair()` and `fork()`. These would need to be converted to use more generic functions that would be mapped back to an appropriate system call depending on the operating system the build is targeted at.

8.5. Protection of the PKRU Register

The protection of memory in MPK is dependent on the `PKRU` register that holds the protection keys not being modified by malicious code. If a bad actor could inject any amount of malicious code into a remote Nginx web server, including system calls or assembly, the `PKRU` register in the worker thread could be rewritten using the `WRPKRU` processor instruction [40, 77] to no longer disable read access to the protected regions of memory, thus removing the protection on the memory pages containing the RSA private key from the main worker thread.

Protecting against an attack this significant could be an entire project in its own right. The vulnerability stems from what is possibly a critical oversight on Intel's part in the design of MPK: the processor does not need to be in supervisor mode to update the `PKRU` register. Arguably, if this technology was simply aimed at assisting developers in writing correct code by making assertions about which memory should be accessible when, MPK is a perfectly adequate design. However, if it was always intended to be used for security features such as what this project proposes, Intel could potentially have done better. By restricting the modification of the `PKRU` register to supervisor mode, it would be possible to require that Nginx be run as root initially, allow it to set up the memory protection in this mode, then protect the memory pages by setting the `PKRU` register. Therefore, once Nginx has demoted itself to an unprivileged user, those memory pages would be protected for the duration of the thread being alive.

An alternative design idea is if memory could be protected and unprotected at will without supervisor mode (as is the case now), with an extra instruction that “locks” the `PKRU` register until the end of the program’s execution. Although jumping into the kernel to request such memory protection via a system call requires some extra computational time, this would only apply to the initial configuration stage. Subsequent requests would not incur any additional overhead or system calls, as is the case now. Locking processor registers so that they can only be changed in supervisor mode is not a new or novel idea; in fact, the Control Registers (such as `CR4`) already possess this attribute [77].

Given that changing an Intel processor feature design post-launch is not realistic future work without being able to convince Intel to make this architecture change, a more realistic suggestion is to add additional protection to the software so that undesired instructions, such as a malicious `WRPKRU` call, cannot be executed by an attacker should they have an exploit for the web server. One approach to this is called Control-Flow Integrity (CFI) [78]. The object of CFI is to ensure that the instructions being executed by a program are in line with a pre-computed Control Flow Graph which describes which functions call which other functions and when. If code were to be remotely injected by an adversary into a running Nginx process to rewrite the `PKRU` register, CFI should in theory detect that the `WRPKRU` instruction is being called at an unexpected time and therefore block it.

CFI is not just a theoretical research concept, and it is in use now. For example, the Windows kernel is now compiled with CFI enabled, and modern Windows installations come with a Microsoft proprietary implementation of CFI called Control Flow Guard available as an API to programs that need protection against exploits [4, 79]. The main issue with a solution like CFI is that it has its own noticeable performance overhead; some “strong” CFI implementations have quoted up to 10% [80]. Comparing the MPK- and process-based implementations with this in mind begins to remove credence from the MPK solution due to the additional performance cost of CFI to keep it secure.

An alternative approach is to employ an MPK-based privilege separation solution as a middle ground between no privilege separation and separate processes. Whilst it may be easier as an attacker to defeat MPK than full process separation, it also has a significantly lower performance overhead and this compromise might still make it worth considering. Extracting the private key from memory with MPK protection in place is still more difficult than in the vanilla scenario where it is not necessary to circumvent any memory protection at all.

8.6. Systems Without MPK Support

At the time of writing the only processors with MPK support are the Intel Xeon Scalable server chips. This means that the vast majority of server chips currently deployed do not support MPK at all. Combine this with the lack of MPK support on Windows and Azure means that, even on chips with MPK hardware support, the feature still may not be available for use. Some future work could therefore be to use the `CPUID` instruction to determine whether `PKU` (Protection Keys in Userspace; the CPUID identifier for this feature) is available. If so, MPK could be used, and for the cases where it not, a classic privilege separation method such as a process-based one as described in this project could be set up in its place. This would bring privilege separation to everybody, and for those machines with MPK available the protection would have better performance.

Deciding whether to enable privilege separation at all could be programmed to be administrator-configurable in, for example, the Nginx configuration file.

8.7. Multiple Private Keys and Key Types

A key limitation in this project is that only one private key is supported and it must be an RSA key. Ideally, certificate and key types other than RSA that Nginx supports, such as ECDSA certificates, should be supported. Furthermore, privilege separation for multiple keys should be supported, as Nginx supports hosting many websites on the same server which can have different certificates and private keys [81]. The experiments built in this project do not consider how how to handle multiple private keys. One solution is to have a separate process or thread for each key to separate them from one another, and another is to have them together in the isolated environment and program the worker process/thread to supply an index number with the encryption request. The encryption process/thread could then use this index number to decide which private key to use to fulfil each request.

After this project was already underway, a paper by Park *et al.* [82] which explores this idea in the wider context of building a wrapper library for MPK was released. In their application of MPK-based security, instead of performing privilege separation, the team surrounds accesses to OpenSSL private key container objects such as `EVP_PKEY` with their own code to unlock and subsequently re-lock the memory. Whilst this idea is interesting as it can reduce the window for attack and no extra thread is needed, if an attacker is able to craft a malicious request that exploits the handshake code they may still be able to extract the private key at the point at which it has been unlocked. The paper's approach also adds additional computational overhead by requiring constant system calls to unlock and re-lock the memory every time a request comes in. In fact, the authors state in the paper that this method "affects performance and programmability such that this paper does not choose that approach".

This project's approach, on the other hand, has a thread with a very short loop body that is asked to do work simply by signalling a condition variable; no MPK-specific code has to be run to service a given request as all the MPK initialisation is done just once per worker process. Another potential source of future research, however, would be to benchmark the thread switching approach in this project against the locking and unlocking of memory as per Park *et al.*'s paper.

8.8. Better Latency Benchmarking

The environment to benchmark Nginx against high load was very limited and contrived, and it became obvious that a single `t3.nano` instance was not enough to saturate the server as the number of connections it could handle plateaued quite early in the testing. Later on, similar benchmarks were attempted, but not recorded, with much higher specification AWS instances used as the client including one with four cores and 16GB of RAM. However, it was extremely difficult to saturate the load on the server, indicating that perhaps there are some restrictions on AWS networking that plateau the number of outbound connections from a server.

Therefore, a future benchmark should have traffic coming from multiple sources to fully saturate Nginx and better show the impact of Process- or MPK-based privilege separation on the server's ability to serve large numbers of parallel requests. Such load testing could not be done in this project because it would violate AWS's load testing policy without obtaining prior approval. Obtaining approval for this level of testing was outside of the project's scope and time constraints, and it likely would have incurred additional costs such as dedicated tenancy to ensure that it would not impact on other users of the EC2 service. A more ideal, albeit costly, solution to this problem is to obtain physical hardware with MPK support at the University and run the tests on internal networks between virtual machines with dedicated processor cores.

8.9. Throughput Benchmarking

Whilst latency benchmarking was attempted, another beneficial test method would be a measurement of throughput. This could be done by increasing a constant amount of load on the server with a tool like `wrk2` [83], and analysing how many responses per second are received. As load goes up, we would expect to see the number of responses per second plateau at a maximum throughput level, at which point queuing would happen on the target server. Theoretically, the higher the overhead of the privilege separation method, the lower the number of responses per second we could expect to see due to a higher amount of time to process each handshake.

8.10. Final Word

As explained above, there are many ways in which this approach could be extended in the future. With a new processor architecture has come a new possibility to revisit privilege separation, and I am curious to see the future work of computer scientists in this area.

References

[1] E. Chien, "Vbs.loveletter.var," 2002 (accessed April 20th, 2019). [Online]. Available: https://www.symantec.com/security-center/writeup/2000-121815-2258-99

[2] CrowdStrike, "2018 global threat report," 2018 (accessed April 6th, 2019). [Online]. Available: https://go.crowdstrike.com/rs/281-OBQ-266/images/Report2018GlobalThreatReport.pdf

[3] Government Communications Headquarters (GCHQ), "The equities process," 2019 (accessed April 28th, 2019). [Online]. Available: https://www.gchq.gov.uk/information/equities-process

[4] V. Brange, "Analysis of the shadow brokers release and mitigation with windows 10 virtualization-based security," 2017 (accessed April 20th, 2019). [Online]. Available: https://www.microsoft.com/security/blog/2017/06/16/analysis-of-the-shadow-brokers-release-and-mitigation-with-windows-10-virtualization-based-security/

[5] M. Field, "Wannacry cyber attack cost the nhs £92m as 19,000 appointments cancelled," 2018 (accessed April 28th, 2019). [Online]. Available: https://www.telegraph.co.uk/technology/2018/10/11/wannacry-cyber-attack-cost-nhs-92m-19000-appointments-cancelled/

[6] I. Thomson, "Notpetya ransomware attack cost us $300m shipping giant maersk," 2017 (accessed April 28th, 2019). [Online]. Available: https://www.theregister.co.uk/2017/08/16/notpetya_ransomware_attack_cost_us_300m_says_shipping_giant_maersk/

[7] T. Brewster, "How hackers broke equifax: Exploiting a patchable vulnerability," 2017 (accessed March 27, 2019). [Online]. Available: https://www.forbes.com/sites/thomasbrewster/2017/09/14/equifax-hack-the-result-of-patched-vulnerability/#283a96465cda

[8] B. Strom, "Att&ck 101," 2018 (accessed April 20th, 2019). [Online]. Available: https://medium.com/mitre-attack/att-ck-101-17074d3bc62

[9] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste, "The cost of the "s" in https," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. New York, NY, USA: ACM, 2014, pp. 133–140. [Online]. Available: http://doi.acm.org/10.1145/2674005.2674991

[10] Google, "Https encryption on the web," 2019 (accessed April 20th, 2019). [Online]. Available: https://transparencyreport.google.com/https/overview?hl=en_GB

[11] T. Foltn, "Majority of the worlds top million websites now use https," 2018 (accessed April 20th, 2019). [Online]. Available: https://www.welivesecurity.com/2018/09/03/majority-worlds-top-websites-https/

[12] Z. Whittaker, "Nsa's use of 'traffic shaping' allows unrestrained spying on americans," 2017 (accessed April 20th, 2019). [Online]. Available: https://www.zdnet.com/article/legal-loopholes-unrestrained-nsa-surveillance-on-americans/

[13] A. Langley, N. Modadugu, and W.-T. Chang, "Overclocking ssl," 2010 (accessed April 20th, 2019). [Online]. Available: https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html

[14] S. Khandelwal, "Stolen d-link certificate used to digitally sign spying malware," 2018 (accessed April 20th, 2019). [Online]. Available: https://thehackernews.com/2018/07/digital-certificate-malware.html

[15] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 16–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251369

[16] O. Deutsch, "Myheritage statement about a cybersecurity incident," 2018 (accessed April 6th, 2019). [Online]. Available: https://blog.myheritage.com/2018/06/myheritage-statement-about-a-cybersecurity-incident/

[17] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8446.txt

[18] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[19] S. Gangan, "A review of man-in-the-middle attacks," *CoRR*, vol. abs/1504.02115, 2015. [Online]. Available: http://arxiv.org/abs/1504.02115

[20] A. Rawdat, "Testing the performance of nginx and nginx plus web servers," 2017 (accessed April 13th, 2019). [Online]. Available: https://www.nginx.com/blog/testing-the-performance-of-nginx-and-nginx-plus-web-servers/

[21] Netcraft, "March 2019 web server survey," 2019 (accessed April 9th, 2019). [Online]. Available: https://news.netcraft.com/archives/2019/03/28/march-2019-web-server-survey.html

[22] M. Kerrisk, "pkeys - overview of memory protection keys," 2019 (accessed April 12th, 2019). [Online]. Available: http://man7.org/linux/man-pages/man7/pkeys.7.html

[23] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14. New York, NY, USA: ACM, 2014, pp. 475–488. [Online]. Available: http://doi.acm.org/10.1145/2663716.2663755

[24] M. Zalewski, "Ssh 1.2.x - crc-32 compensation attack detector," 2001 (accessed April 28th, 2019). [Online]. Available: https://www.exploit-db.com/exploits/20617

[25] Sun Microsystems Inc., "XDR: External Data Representation Standard," RFC 1014, Jun. 1987. [Online]. Available: https://rfc-editor.org/rfc/rfc1014.txt

[26] The Apache Software Foundation, "Apache ssl/tls encryption," 2019 (accessed April 20th, 2019). [Online]. Available: https://httpd.apache.org/docs/2.4/ssl/

[27] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting applications into reduced-privilege compartments," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 309–322. [Online]. Available: http://dl.acm.org/citation.cfm?id=1387589.1387611

[28] A. Bittau, "Toward least-privilege isolation for software," Ph.D. dissertation, University College London, 11 2009.

[29] M. Kerrisk, "pthreads - posix threads," 2019 (accessed April 13th, 2019). [Online]. Available: http://man7.org/linux/man-pages/man7/pthreads.7.html

[30] M. Krohn, "Building secure high-performance web services with okws," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '04. Berkeley, CA, USA: USENIX Association, 2004, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247415.1247430

[31] A. Ayer, "Totally isolated tls unwrapping server," 2019 (accessed April 8th, 2019). [Online]. Available: https://www.opsmate.com/titus/

[32] ——, "Totally isolated tls unwrapping server - github," 2019 (accessed April 8th, 2019). [Online]. Available: https://github.com/AGWA/titus

[33] ——, "Protecting the openssl private key in a separate process," 2019 (accessed April 8th, 2019). [Online]. Available: https://www.agwa.name/blog/post/protecting_the_openssl_private_key_in_a_separate_process

[34] M. Caswell and K. Roeckx, "Tls1.3," 2018 (accessed April 7th, 2019). [Online]. Available: https://wiki.openssl.org/index.php/TLS1.3

[35] M. Caswell, "Using tls1.3 with openssl," 2017 (accessed April 9th, 2019). [Online]. Available: https://www.openssl.org/blog/blog/2017/05/04/tlsv1.3/

[36] N. Sullivan, "Ecdsa: The digital signature algorithm of a better internet," 2014 (accessed April 29th, 2019). [Online]. Available: https://blog.cloudflare.com/ecdsa-the-digital-signature-algorithm-of-a-better-internet/

[37] P. Timteo, "How to set up an nginx https website with an ecdsa certificate (and get an a+ rating on ssl labs)," 2017 (accessed April 20th, 2019). [Online]. Available: https://zurgl.com/how-to-set-up-an-nginx-https-website-with-an-ecdsa-certificate-and-get-an-a-rating-on-ssl-labs/

[38] Nginx Team, "Nginx changelog," 2019 (accessed April 7th, 2019). [Online]. Available: https://nginx.org/en/CHANGES

[39] Microsoft Open Source, "Frequently asked questions about windows subsystem for linux," 2019 (accessed April 28th, 2019). [Online]. Available: https://docs.microsoft.com/en-us/windows/wsl/faq

[40] T. Gleixner, "Protection keys syscall interface," 2016 (accessed April 20th, 2019). [Online]. Available: https://lore.kernel.org/patchwork/patch/724116/

[41] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *CoRR*, vol. abs/1801.01207, 2018. [Online]. Available: http://arxiv.org/abs/1801.01207

[42] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *CoRR*, vol. abs/1801.01203, 2018. [Online]. Available: http://arxiv.org/abs/1801.01203

[43] Intel Corporation, "Deep dive: Retpoline: A branch target injection mitigation," 2018 (accessed April 13th, 2019). [Online]. Available: https://software.intel.com/security-software-guidance/insights/deep-dive-retpoline-branch-target-injection-mitigation

[44] Amazon Web Services, "Aws cloudhsm," 2019 (accessed April 13th, 2019). [Online]. Available: https://aws.amazon.com/cloudhsm/

[45] A. Ayer, "Titus isolation techniques, continued," 2019 (accessed April 8th, 2019). [Online]. Available: https://www.agwa.name/blog/post/titus_isolation_techniques_continued

[46] C. Sanders, "Fv2 vms are now available, the fastest vms on azure," 2017 (accessed April 20th, 2019). [Online]. Available: https://azure.microsoft.com/en-gb/blog/fv2-vms-are-now-available-the-fastest-vms-on-azure/

[47] Amazon Web Services, "Introducing amazon ec2 t3 instances," 2018 (accessed April 20th, 2019). [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2018/08/introducing-amazon-ec2-t3-instances/

[48] Linus Torvalds et al., "Linux kernel cpuid features header," 2019 (accessed April 12th, 2019). [Online]. Available: https://github.com/torvalds/linux/blob/52f64909409c17adf54fcf5f9751e0544ca3a6b4/arch/x86/include/asm/cpufeatures.h#L323

[49] Amazon Web Services, "Regions and availability zones," 2019 (accessed April 12th, 2019). [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.RegionsAndAvailabilityZones.html

[50] "Ieee standard for information technology– portable operating system interface (posix(r)) base specifications, issue 7," *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008) - Redline*, pp. 1–6900, Jan 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8372834

[51] O. Garrett, "Inside nginx: How we designed for performance & scale," 2015 (accessed April 12th, 2019). [Online]. Available: https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/

[52] M. Kerrisk, "select()," 2019 (accessed April 13th, 2019). [Online]. Available: http://man7.org/linux/man-pages/man2/select.2.html

[53] ——, "epoll - i/o event notification facility," 2019 (accessed April 13th, 2019). [Online]. Available: http://man7.org/linux/man-pages/man7/epoll.7.html

[54] B. Karp, E. Kohler, D. Mazires, P. Gibbons, D. OHallaron, and R. Bryant, "Concurrency with threads," 2018 (accessed April 13th, 2019). [Online]. Available: http://www0.cs.ucl.ac.uk/staff/B.Karp/3007/s2018/lectures/3007-lecture16-threads-sync.pdf

[55] Amazon Web Services, "How do i manage the clock source for ec2 instances running linux?" 2019 (accessed April 16th, 2019). [Online]. Available: https://aws.amazon.com/premiumsupport/knowledge-center/manage-ec2-linux-clock-source/

[56] Red Hat, "Chapter 15. timestamping," Unknown (accessed April 16th, 2019). [Online]. Available: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_MRG/2/html/Realtime_Reference_Guide/chap-Timestamping.html

[57] S. Noonan, "Clocksource behavior microbenchmarks," 2019 (accessed April 16th, 2019). [Online]. Available: https://github.com/tycho/clockperf

[58] Packagecloud, "Two frequently used system calls are $\tilde{7}7\%$ slower on aws ec2," 2017 (accessed April 16th, 2019). [Online]. Available: https://blog.packagecloud.io/eng/2017/03/08/system-calls-are-much-slower-on-ec2/

[59] J. Barr, “Amazon ec2 update additional instance types, nitro system, and cpu options,” 2018 (accessed April 16, 2019). [Online]. Available: https://aws.amazon.com/blogs/aws/amazon-ec2-update-additional-instance-types-nitro-system-and-cpu-options/

[60] Amazon Web Services, “Amazon ec2 faqs,” 2019 (accessed April 16th, 2019). [Online]. Available: https://aws.amazon.com/ec2/faqs/

[61] Red Hat, “Chapter 14. kvm guest timing management,” Unknown (accessed April 16, 2019). [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/virtualization_host_configuration_and_guest_installation_guide/chap-virtualization_host_configuration_and_guest_installation_guide-kvm_guest_timing_management

[62] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[63] R. Bharadwaj, *Mastering Linux Kernel Development*. Packt Publishing, 2017.

[64] GNU, “Constraints for particular machines,” Unknown (accessed April 17th, 2019). [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Machine-Constraints.html

[65] G. Paoloni, “How to benchmark code execution times on intelia-32 and ia-64 instruction set architectures,” 2010 (accessed April 17th, 2019). [Online]. Available: https://www.intel.de/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf

[66] Stenberg, Daniel *et al.*, “curl.1 the man page,” 2019 (accessed April 20th, 2019). [Online]. Available: https://curl.haxx.se/docs/manpage.html

[67] W. Glozer, “wrk - a http benchmarking tool,” 2019 (accessed April 20th, 2019). [Online]. Available: https://github.com/wg/wrk

[68] B. Boucheron, “An introduction to load testing,” 2017 (accessed April 16th, 2019). [Online]. Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-load-testing

[69] Amazon Web Services, “Amazon ec2 testing policy,” 2019 (accessed April 20th, 2019). [Online]. Available: https://aws.amazon.com/ec2/testing/

[70] S. Baumgartner, “A slow website - time to first byte (ttfb),” 2016 (accessed April 16th, 2019). [Online]. Available: https://www.keycdn.com/blog/a-slow-website-time-to-first-byte-ttfb

[71] D. Escardo, “Time to first byte: The critical seo metric you aren’t measuring,” 2018 (accessed April 16th, 2019). [Online]. Available: https://www.impactbnd.com/blog/time-to-first-byte-seo

[72] Kissmetrics, “How loading time affects your bottom line,” 2011 (accessed April 16th, 2019). [Online]. Available: https://blog.kissmetrics.com/wp-content/uploads/2011/04/loading-time.pdf

[73] GNU, “Debugging forks,” 2019 (accessed April 28th, 2019). [Online]. Available: https://sourceware.org/gdb/onlinedocs/gdb/Forks.html

[74] R. Nelson, “Tuning nginx for performance,” 2014 (accessed April 20th, 2019). [Online]. Available: https://www.nginx.com/blog/tuning-nginx/

[75] Valgrind Developers, “Memcheck: a memory error detector,” 2010 (accessed April 20th, 2019). [Online]. Available: http://valgrind.org/docs/manual/mc-manual.html

[76] Google, “Addresssanitizer,” 2018 (accessed April 20th, 2019). [Online]. Available: https://github.com/google/sanitizers/wiki/AddressSanitizer

[77] Intel Corporation, “System programming guide,” 2019 (accessed April 20th, 2019). [Online]. Available: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

[78] M. Abadi, M. Budiu, and . Erlingsson, “Control-flow integrity,” Tech. Rep., November 2005, aCM Conference on Computer and Communication Security (CCS). [Online]. Available: https://www.microsoft.com/en-us/research/publication/control-flow-integrity/

[79] J. Kennedy and M. Satran, “Control flow guard,” 2018 (accessed April 20th, 2019). [Online]. Available: https://docs.microsoft.com/en-gb/windows/desktop/SecBP/control-flow-guard

[80] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, “Preventing memory error exploits with wit,” in *Proceedings of the*

IEEE Symposium on Security and Privacy. IEEE, May 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/preventing-memory-error-exploits-with-wit/

[81] A. Braun, “Use nginx to host multiple ssl secured websites with one external ip address,” 2017 (accessed April 20th, 2019). [Online]. Available: https://progressive-code.com/post/8/Use-Nginx-to-host-multiple-SSL-secured-websites-with-one-external-IP-address

[82] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, “libmpk: Software abstraction for intel memory protection keys,” *CoRR*, vol. abs/1811.07276, 2018. [Online]. Available: http://arxiv.org/abs/1811.07276

[83] G. Tene and W. Glozer, “wrk2: a http benchmarking tool based mostly on wrk,” 2018 (accessed April 16th, 2019). [Online]. Available: https://github.com/giltene/wrk2

Appendix A.
Build Commands

OpenSSL is not be configured directly, but is instead configured via the default Nginx build scripts. These are the commands that are used to enable Nginx and OpenSSL debugging, whilst forcing Nginx to use the OpenSSL code downloaded from GitHub into a separate directory (in this case `../openssl-1.1.1` relative to the `configure` script). The changes to the code in both builds of Nginx and OpenSSL with privilege separation enabled were also placed into the `openssl-1.1.1` directory, and these build commands ensure that these are picked up.

A.1. With Debugging Enabled

Debugging was enabled only for debugging purposes. It was disabled for all the performance tests.

```
./configure --with-http_ssl_module --with-http_realip_module --with-
    http_auth_request_module --with-stream_ssl_module --with-openssl=../
    openssl-1.1.1 --prefix= --with-debug --with-openssl-opt=-g
```

A.2. With Debugging Disabled

This is the build configuration used when performance testing was done, so that the overhead of the debugging options in Nginx and OpenSSL would not affect the results.

```
./configure --with-http_ssl_module --with-http_realip_module --with-
    http_auth_request_module --with-stream_ssl_module --with-openssl=../
    openssl-1.1.1 --prefix=
```

Appendix B.
Nginx Configuration

To reduce the variables in the test procedure, the configuration was kept consistent across all debugging and testing. The file used is as follows, with annotations to explain why each option was chosen.

```
# We only have one process enabled so that we can judge performance on a
    single core.
# The code is also a proof of concept that is not built with multi-core
    processing in mind.
worker_processes  1;

# Debug level logging is enabled.
# No further log entries are added in the privilege separated code, so this
    setting can be left as-is on all builds.
error_log  logs/error.log  debug;

# Standard pid file setting
pid        logs/nginx.pid;

# Run in the foreground for ease of management and killing
daemon off;

# High number of connections for testing
events {
    worker_connections  1024;
}

http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;

    # HTTPS server
    # Only HTTPS is enabled; we do not allow non-HTTPS
    server {
        # Listen on a high port number so we don't have to run as root
        listen       8443 ssl;

        # Listen on all IP addresses
        server_name  0.0.0.0;

        # Use our self-signed RSA certificate
        ssl_certificate       cert/selfsigned.crt;
        ssl_certificate_key   cert/selfsigned.key;

        ssl_session_cache     shared:SSL:1m;
        ssl_session_timeout   5m;

        # For this project, only enable TLS v1.3
        ssl_protocols TLSv1.3;

        # Disable TLS early data so that we don't have to worry about this
            variable
        ssl_early_data off;

        # Enable all TLS 1.3 ciphers
        ssl_ciphers TLS13-CHACHA20-POLY1305-SHA256:TLS13-AES-256-GCM-SHA384:
            TLS13-AES-128-GCM-SHA256:EECD+CHACHA20:EECDH+AESGCM:EECDH+AES;

        # Serve up a static HTML file on successful connections
        location / {
            root    www;
            index   index.html index.htm;
        }
    }

}
```

Appendix C. Clockperf Results

Provided here are the full outputs from `clockperf` [57] when run with each of the three system timers available: `kvm-clock`, `tsc` and `acpi_pm`. The samples were taken on an Amazon AWS `t3.nano` instance hosted on an Amazon Nitro KVM hypervisor.

Each output has a number of attributes. The choice of which time functionality to use was based on the following criteria:

- `Mono` must be `Yes`, meaning that the timer result is monotonic.
- `Resol` should be as high as reasonably possible, to show a high degree of timer resolution. All hyphen output means that the tick rate is too precise to be estimated.
- `Fail`, `Warp`, `Stal` and `Regr` must all be 0 to indicate that the clock source always advances, it never jumps by an inexplicably high number, it never stalls and it never regresses (goes backwards).
- The time to take a measurement (the `Cost`) should be as low as possible. Given the range of results and large selection of timing methods available, 100ns will be set as the absolute cost ceiling.

With a clock source chosen, the method to obtain the time is chosen by selecting the one with the least error; this is the method which yields the lowest deviation (+/-).

C.1. **`kvm-clock`**

```
clockperf v2.0.1-7-ga014d90

== Reported Clock Frequencies ==

tsc                  2501MHz
realtime             1000MHz
realtime_crs         250Hz
monotonic            1000MHz
monotonic_crs        250Hz
monotonic_raw        1000MHz
boottime             1000MHz
process              1000MHz
thread               1000MHz
clock                1000KHz
```

```
time                    1Hz


== Clock Behavior Tests ==

Name                 Cost(ns)     +/-      Resol   Mono  Fail  Warp  Stal  Regr
tsc                     18.01   0.63%     ——       Yes      0     0     0     0
gettimeofday            33.28   0.31%  1000KHz      No      0     0   999     0
realtime                40.69   1.25%     ——       Yes      0     0     0     0
realtime_crs            23.91   1.52%    250Hz      No    998     1     1     0
                        23.35  12.14%
monotonic               42.24   1.31%     ——       Yes      0     0     0     0
monotonic_crs           23.50   0.73%    250Hz      No    998     1     1     0
                        23.35  12.15%
monotonic_raw          346.28   0.31%     ——       Yes      0     0     0     0
boottime               352.74   0.39%     ——       Yes      0     0     0     0
process                426.31   0.37%     ——       Yes      0     0     0     0
thread                 420.03   1.89%     ——       Yes      0     0     0     0
clock                  437.76   0.84%  1000KHz      No      0     0   337     0
getrusage              530.49   0.27%  1000KHz      No      0     0   104     0
ftime                   42.42   0.66%   1000Hz      No    991     0     8     0
time                    11.93   2.31%      1Hz      No   1000     0     0     0
```

Valid candidate time methods based on results above:

```
Name                 Cost(ns)     +/-      Resol   Mono  Fail  Warp  Stal  Regr
tsc                     18.01   0.63%     ——       Yes      0     0     0     0
realtime                40.69   1.25%     ——       Yes      0     0     0     0
monotonic               42.24   1.31%     ——       Yes      0     0     0     0
```

C.2. tsc

```
clockperf v2.0.1-7-ga014d90

== Reported Clock Frequencies ==

tsc                     2500MHz
realtime                1000MHz
realtime_crs            250Hz
monotonic               1000MHz
monotonic_crs           250Hz
```

```
monotonic_raw           1000MHz
boottime                1000MHz
process                 1000MHz
thread                  1000MHz
clock                   1000KHz
time                    1Hz


== Clock Behavior Tests ==

Name                     Cost(ns)      +/-      Resol   Mono   Fail   Warp   Stal   Regr
tsc                         18.08    0.69%     ——       Yes      0      0      0      0
gettimeofday                28.70    0.35%   1000KHz    No       0      0    999      0
realtime                    38.99    0.76%     ——       Yes      0      0      0      0
realtime_crs                23.74    2.50%    250Hz     No     998      1      1      0
                            23.35   14.77%
monotonic                   38.94    0.49%     ——       Yes      0      0      0      0
monotonic_crs               23.66    0.62%    250Hz     No     998      1      1      0
                            23.35   12.14%
monotonic_raw              333.80    0.38%     ——       Yes      0      0      0      0
boottime                   338.03    0.35%     ——       Yes      0      0      0      0
process                    426.26    0.32%     ——       Yes      0      0      0      0
thread                     417.86    1.44%     ——       Yes      0      0      0      0
clock                      429.08    0.33%   1000KHz    No       0      0    367      0
getrusage                  542.83    1.30%   1000KHz    No       0      0    152      0
ftime                       40.74    0.53%   1000Hz     No     991      0      8      0
time                        11.94    2.30%      1Hz     No    1000      0      0      0
```

Valid candidate time methods based on results above:

```
tsc                         18.08    0.69%     ——       Yes      0      0      0      0
realtime                    38.99    0.76%     ——       Yes      0      0      0      0
monotonic                   38.94    0.49%     ——       Yes      0      0      0      0
```

C.3. acpi_pm

```
clockperf v2.0.1-7-ga014d90


== Reported Clock Frequencies ==


tsc                     2510MHz
```

```
realtime            1000MHz
realtime_crs        250Hz
monotonic           1000MHz
monotonic_crs       250Hz
monotonic_raw       1000MHz
boottime            1000MHz
process             1000MHz
thread              1000MHz
clock               1000KHz
time                1Hz


== Clock Behavior Tests ==

Name                  Cost(ns)      +/-      Resol   Mono  Fail   Warp   Stal   Regr
tsc                      22.83    1.90%    ------    Yes     0      0      0      0
gettimeofday           9394.71    6.54%    ------     No     0      0      0      0
realtime               9102.22    0.32%    ------    Yes     0      0      0      0
realtime_crs             23.80    4.43%     250Hz     No   998      1      1      0
                         23.35   14.76%
monotonic              9107.45    0.50%    ------    Yes     0      0      0      0
monotonic_crs            24.05    0.56%     250Hz     No   998      1      1      0
                         24.02   12.67%
monotonic_raw          9206.60    1.34%    ------     No     0      0      0      0
boottime               9045.66    0.38%    ------    Yes     0      0      0      0
process                 448.23    2.77%    ------    Yes     0      0      0      0
thread                  422.81    0.43%    ------    Yes     0      0      0      0
clock                   435.81    0.42%   1000KHz     No     0      0    367      0
getrusage               541.09    0.42%   1000KHz     No     0      0    101      0
ftime                  9072.10    0.26%    1000Hz     No     0      0   1000      0
time                     12.20    2.17%       1Hz     No  1000      0      0      0
```

Valid candidate time methods based on results above:

```
tsc                      22.83    1.90%    ------    Yes     0      0      0      0
```

Appendix D. Handshake Timing Scripts

These are the two scripts used to generate the first set of Macrobenchmarks. The first is a shell script launched from the client virtual machine to perform five hundred successive handshakes using `cURL` and log the results to files, and the second is a Python 3 script to parse these results and generate the metrics provided in the report.

D.1. Shell Script to Generate Handshakes

```
#!/bin/bash
RUNS=500
URL="https://3.8.145.72:8446/"
echo "Test URL: $URL"

for i in $(seq -f "%03g" 1 $RUNS)
do
    echo "Test $i of $RUNS"
    OUTFILE="results/$1_$i.txt"
    curl -k --no-sessionid --verbose $URL --trace $OUTFILE
done
```

D.2. Python 3 Script to Parse Results

```
#!/usr/env/bin python3
import os
import sys

from datetime import datetime, date, time

# vanilla, process or mpk, passed on the command line
test_name = sys.argv[1]

files = []
for file in os.listdir("results"):
    if file.startswith(test_name):
        files.append(os.path.join("results", file))

print("Analysing files: {} ...".format(" ".join(files[:10])))

results = []

for file in files:
    with open(file, 'rt') as f:
        data = f.readlines()
    initial_timestamp = time.fromisoformat(data[0].split(' ')[0])
    http_ok_header = [s for s in data if "Recv header, 17 bytes (0x11)" in
     ↪  s][0]
    ttfb_timestamp = time.fromisoformat(http_ok_header.split(' ')[0])
    time_diff = datetime.combine(date.min, ttfb_timestamp) - \
        datetime.combine(date.min, initial_timestamp)
    secs = time_diff.total_seconds()
    assert(secs > 0)

    results.append(time_diff.total_seconds())

total_time_taken = sum(results)
print("Parsed {} files".format(len(files)))
print("Average: {}s".format(total_time_taken / len(files)))
print("Average: {}ms".format(total_time_taken / len(files) * 1000))
```

Appendix E. Microbenchmark Code

The code provided here is an excerpt from `rpc.h` within the modifications I made to OpenSSL for this project. It is a modified version of the code provided by Intel in their white paper on benchmarking code execution with the Timestamp Counter [65].

The functions are forced to be inline to avoid the computational overhead of jumping into a function and returning. Instead, the inline assembly is set up to protect the registers that get clobbered by the instructions being run.

```
unsigned long long inline rdtsc_before() __attribute__((always_inline));
unsigned long long inline rdtsc_before()
{
    unsigned long long ticksl, ticksh;
    __asm__ __volatile__(
        "CPUID\n\t"
        "RDTSC\n\t"
        "mov %%rdx, %0\n\t"
        "mov %%rax, %1\n\t": "=r" (ticksh), "=r" (ticksl):: "%rax", "%rbx",
          ↪  "%rcx", "%rdx"
    );

    return (ticksh << 32) | ticksl;
}

unsigned long long inline rdtsc_after() __attribute__((always_inline));
unsigned long long inline rdtsc_after()
{
    unsigned long long ticksl, ticksh;
    __asm__ __volatile__(
        "RDTSCP\n\t"
        "mov %%rdx, %0\n\t"
        "mov %%rax, %1\n\t"
        "CPUID\n\t": "=r"(ticksh), "=r"(ticksl)::"%rax", "%rbx", "%rcx",
          ↪  "%rdx"
    );

    return (ticksh << 32) | ticksl;
}
```

Appendix F.
Enclosed Project Files

Enclosed within the zip file attached to this project are a number of files and directories.

F.1. Code Directory

Within the `Code` directory is a clean copy of all the Nginx and OpenSSL code generated throughout the project. It should build with one of the build commands in Appendix A.

The directory has a full copy of the Git history, right from the initial commit with none of my code changes through to the latest code used for benchmarking.

The final Git commit is GPG signed with my private signing key (`D73B45E2973633B59A2C77112ABB1755FD93244C`) to prove its authenticity.

F.2. PrivSep.diff

The `PrivSep.diff` file is a patch that contains all my code changes from the initial commit (commit hash `a7f3...`) through to the latest version of the code included in the `Code` directory. The patch can be applied to the following code:

- **Nginx** 1.15.8 downloaded and extracted to the directory `nginx/`
- **OpenSSL 1.1.1 stable**, commit hash `ed48d20` cloned using `Git` into the directory `openssl-1.1.1/`.

F.3. Data Analysis.xlsx

`Data Analysis.xlsx` is an Excel spreadsheet that contains all the data for the data analysis, including raw values and formulae.

F.4. Diagrams.pptx

`Diagrams.pptx` is a PowerPoint file that contains the source for all the diagrams created for this report.

F.5. HandshakeResultsParser.py

`HandshakeResultsParser.py` is the parser for the first stage of the macrobenchmark results, written in Python 3. This code is the same as in Appendix D. It is designed to be used with files such as those contained in `handshake_results_final.tar.gz`.

F.6. handshake_results_final.tar.gz

`handshake_results_final.tar.gz` is a highly compressed copy of all the `cURL` outputs created for the first macrobenchmarks. These files are parsed with `HandshakeResultsParser.py`.

Appendix G.
rpc.h Configuration Options

Configuring the build is done by modifying `openssl-1.1.1/crypto/rpc/rpc.h`. There are a number of `#define` compiler macros there that switch features on and off.

A copy of those settings is listed here with their explanations for convenience. Some templates are also provided below for quick testing of the three main configurations.

```
// Global switch to enable or disable Privilege Separation
#define PRIV_SEP_FUNC_ENABLED

// Define PRIV_SEP_BENCHMARK to enable microbenchmarking
#define PRIV_SEP_BENCHMARK

// Set the benchmark phase
// Phase 1: RSA operation only
// Phase 2: Entire rsa_ossl_private_encrypt, including the crypto
#define PRIV_SEP_BENCHMARK_PHASE 2

// ID of each Privilege Separation mode
#define PRIV_SEP_MODE_PROCESS 1
#define PRIV_SEP_MODE_MPK 2

// Which Privilege Separation mode to use (based on the above)
#define PRIV_SEP_MODE PRIV_SEP_MODE_MPK

// Define PRIV_SEP_DEBUG to enable textual output from the priv sep code
#define PRIV_SEP_DEBUG

// Define PRIV_SEP_MODE_MPK_TEST_SEGFAULT to ensure accessing memory locked
 ↪  by MPK results in a segmentation fault.
// This is for feature testing only, and must be disabled for Nginx to work!
#define PRIV_SEP_MODE_MPK_TEST_SEGFAULT
```

G.1. Vanilla

```
// Global switch to enable or disable Privilege Separation
// #define PRIV_SEP_FUNC_ENABLED

// Define PRIV_SEP_BENCHMARK to enable microbenchmarking
// #define PRIV_SEP_BENCHMARK

// Set the benchmark phase
// Phase 1: RSA operation only
// Phase 2: Entire rsa_ossl_private_encrypt, including the crypto
#define PRIV_SEP_BENCHMARK_PHASE 1

// ID of each Privilege Separation mode
#define PRIV_SEP_MODE_PROCESS 1
#define PRIV_SEP_MODE_MPK 2

// Which Privilege Separation mode to use (based on the above)
#define PRIV_SEP_MODE PRIV_SEP_MODE_MPK

// Define PRIV_SEP_DEBUG to enable textual output from the priv sep code
// #define PRIV_SEP_DEBUG

// Define PRIV_SEP_MODE_MPK_TEST_SEGFAULT to ensure accessing memory locked
 ↪  by MPK results in a segmentation fault.
// This is for feature testing only, and must be disabled for Nginx to work!
// #define PRIV_SEP_MODE_MPK_TEST_SEGFAULT
```

G.2. Process-Based Privilege Separation

```
// Global switch to enable or disable Privilege Separation
#define PRIV_SEP_FUNC_ENABLED

// Define PRIV_SEP_BENCHMARK to enable microbenchmarking
// #define PRIV_SEP_BENCHMARK

// Set the benchmark phase
// Phase 1: RSA operation only
// Phase 2: Entire rsa_ossl_private_encrypt, including the crypto
#define PRIV_SEP_BENCHMARK_PHASE 1

// ID of each Privilege Separation mode
#define PRIV_SEP_MODE_PROCESS 1
#define PRIV_SEP_MODE_MPK 2

// Which Privilege Separation mode to use (based on the above)
#define PRIV_SEP_MODE PRIV_SEP_MODE_PROCESS

// Define PRIV_SEP_DEBUG to enable textual output from the priv sep code
// #define PRIV_SEP_DEBUG

// Define PRIV_SEP_MODE_MPK_TEST_SEGFAULT to ensure accessing memory locked
 ↪  by MPK results in a segmentation fault.
// This is for feature testing only, and must be disabled for Nginx to work!
// #define PRIV_SEP_MODE_MPK_TEST_SEGFAULT
```

G.3. MPK-Based Privilege Separation

```
// Global switch to enable or disable Privilege Separation
#define PRIV_SEP_FUNC_ENABLED

// Define PRIV_SEP_BENCHMARK to enable microbenchmarking
// #define PRIV_SEP_BENCHMARK

// Set the benchmark phase
// Phase 1: RSA operation only
// Phase 2: Entire rsa_ossl_private_encrypt, including the crypto
#define PRIV_SEP_BENCHMARK_PHASE 1

// ID of each Privilege Separation mode
#define PRIV_SEP_MODE_PROCESS 1
#define PRIV_SEP_MODE_MPK 2

// Which Privilege Separation mode to use (based on the above)
#define PRIV_SEP_MODE PRIV_SEP_MODE_MPK

// Define PRIV_SEP_DEBUG to enable textual output from the priv sep code
// #define PRIV_SEP_DEBUG

// Define PRIV_SEP_MODE_MPK_TEST_SEGFAULT to ensure accessing memory locked
 ↪  by MPK results in a segmentation fault.
// This is for feature testing only, and must be disabled for Nginx to work!
// #define PRIV_SEP_MODE_MPK_TEST_SEGFAULT
```