École Polytechnique Fédérale de Lausanne

Characterization of the overheads for comprehensive compartmentalization of software

by Andrés Sánchez Marín

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Damian Pfammatter
External Expert

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 25, 2021

Acknowledgments

I want to thank everyone I've got the honor to talk with and who has provided me with some guidance in this journey. Commencing by Prof. Mathias Payer, apart from his amazing insights of the ongoing research, has always shown how vital the well-done work is and demonstrated the value of building research in collaboration. Thanks to Atri Bhattacharyya, whose counseling has helped through the elaboration of this project and the previous ones, I've got the honor to work with him.

My friends and family have given me all their support through this journey, especially my mother and brothers, which have remained strong, even more after all the troubled times we have gotten through the pandemic and my father's death. My father, who I do not forget, who taught me the importance of honesty, integrity, and doing everything (even the most insignificant things) with love, has demonstrated how to take care of our beloved ones regardless of how difficult the situations may seem.

To the HexHive laboratory members, the rest of colleagues from the Systems Security group at IBM Research Zurich, MIT CSAIL, and IMDEA Software Institute. Thanks to them for showing the value of a collaborative environment. Whose members are always willing to help and discuss ideas through the different research projects that I have worked on and accomplished; for their kind help.

Lausanne, June 25, 2021 Andrés Sánchez Marín

Abstract

Our computing systems face an increasing number of potential threats. We can lessen the threat's repercussion by protecting the system components individually. On the software side, approaches like sandboxing apply successfully, but there is an urge for similar techniques for systems aiming for performance. Accelerated compartmentalization shows up as a promising defense technique in different styles, reducing the impact an attacker can generate.

In this project, we explore the different proposed compartmentalization techniques and define a methodology for their evaluation. We prototype the assessment method by applying compartmentalization techniques to Nginx webserver and produce a baseline that bases the separation of its components in OS processes using inter-process-communication for having a similar behavior as the non-compartmentalized version. We demonstrate that securing the boundaries of components comes at a cost that future work shall address. We define how to extend our methodology to be adopted for any compartmentalization technique. With the realization of this project, we aim for a more conscious security-aware design of our systems with long-term repercussions.

Contents

Chapter 1

Introduction

Modern systems tend to have multiple parties' data managed by a single process because of context switching costs in modern architectures and the single-purpose piece of software which controls all the data. Despite isolation at the process level, OS processes are more eager to manage the data in their memory. Notwithstanding the efforts to have data isolation through models where contained data should not be accessible, latent errors in the code infrastructure or the hardware may expose new vulnerabilities. Because of the potential threats, several proposals from the research and industry community have shown up tackling this problem with different approaches, aiming to provide more robust data isolation and restraining the potential attacker impact inside some boundaries.

Different projects from the systems community propose ways to avoid having separate parties accessing the same data layout and its risk. Those proposals go grom the redesign of the interaction between various parties by adding a new abstraction level, passing by through the definition of different trust domains. They have in common the insulation of the process's work domains; we refer to this model as compartmentalization.

All compartmentalization proposals lack a single motive: How better is the proposal related to existing ones, and what is the substantial difference that the proposal has, accompanied by the implementation costs, both in performance and technical cost of its implementation. Then, granting all this, they propose an evaluation method to support their idea, barely noticing the requirements an evaluation system should have in extensive detail.

The problem is intrinsically tricky; the proposals may work at different levels of abstraction (architecture, operating systems, libraries), which evaluation is costly. The approach for a proper assessment of the concepts must consider all the system layers appropriately. Furthermore, all the different layers should be factually measurable in their effect on the whole performance. Finally, we must define a system with security and performance measurement as the primary evaluation goal.

This project's motive is the evaluation systematization of defense-based security systems, focusing in this case on compartmentalization. We cannot measure security quantitatively but qualitatively. All compartmentalization techniques reside in the same security category. Still, we can compare the efficiency of methods offering similar security mechanisms against the same kind of attack.

We propose and prototype an evaluation system for a broad scenario, suggesting the different aspects its design must consider. Moreover, aside from a system that implements compartmentalization for protecting separate working domains inside a process, we must look at the metrics' obtention and their importance.

We propose the components for an efficient evaluation of compartmentalization. The requirements for this evaluation platform to work are: the system presents weaknesses because when lacking from compartmentalization, it uses the whole system stack for its processing (from the hardware passing by the OS utilities), the piece of software to evaluate interacts with the environment, has parameters that affect the system behavior. We test its effectiveness with existing software in a common-used stack of processing over the x86 architecture on the FreeBSD Operating System.

We prototype a system for performing that evaluation both in a virtual environment and in bare-metal. Our prototype implements a tweaked version of Nginx that introduces compartmentalization techniques in its design and is ready to adjust to other existing methods with negligible development costs. The lwC's implementation of its compartmentalization technique in Nginx version inspires our modification. We update the lwC's Nginx definition and instrument the code for evaluating any other compartmentalization model.

In this project, we define the requirements of platforms for comparing different compartmentalization techniques. We prototype an evaluation system based on Nginx for a brief evaluation of other compartmentalization alternatives. Finally, we discuss the security implications of compartmentalization, the different techniques proposed from the literature, and the urge to apply these techniques in our systems. We aspire via this project to contribute to the scientific community by setting the proper evaluation methodology apart from the presented prototype, which can help compare different compartmentalization techniques in the light of more robust security models.

Chapter 2

Background

In this section, we present all the terminology and notions this thesis builds on. Then, we explore the abstract concepts and dive into examples of how those concepts are applied.

2.1 OS abstractions

Operating Systems provide a hardware-independent execution environment to application developers to ease the development process. Furthermore, they give the software a set of abstractions over the hardware; this implies software portability to run on different hardware instances with a similar Operating System and configuration. This section explores the common abstractions an Operating System exposes necessary to understand the development of this project in detail. The OS kernel implements all the functionalities, which labor is to provide all the OS-specific abstractions to the software that will run in the machine.

2.1.1 Processes

The system will organize the software it runs in what will be called processes, each with its own address space to operate. Processes are isolated among them; isolation is granted through disaggregation address spaces, providing non-interference of address spaces when reading or writing. Any unexpected way to communicate between two processes in which the OS does not interfere (either by acting as the channel or allowing straightforward communication) is considered a covert channel.

2.1.2 Virtual memory

The Operating System controls memory through physical addresses (this is the physical memory layout); however, this same layout is not exposed to the processes, which will perceive a fixed memory layout (as a characteristic of memory abstraction), called the virtual memory space. Then, a translation exists from the virtual memory addresses to the physical ones; the OS performs this process. Hardware components can accelerate the translation through the MMU (Memory Management Unit) and TLB (Translation Lookaside Buffer) elements. Virtual memory structures in pages, the minimal transmission unit between the memory and the LLC (Last Level Cache); data is retrieved and backed up with this granularity.

The OS shows a different memory layout for every process; this is the virtual memory, on which an address of the virtual memory of a process will be mapped to an address of the physical memory, delegating that translation to the components in a lower level of abstraction (hardware mechanisms and the OS). The translation is effected within the PID of the process, requesting a concrete memory page.

2.2 FreeBSD

There is all type of flavors of operating systems with different sets of abstractions, among them there exists FreeBSD, an Open Source, Unix-like operating system [10]. We choose this Operating System to systematize our evaluation because of its extensive documentation regarding the OS APIs and the stability of its codebase. Furthermore, it includes all the mainstream features of modern Operating Systems such as Virtual Memory or multi-user facilities in its design.

Although it is not widely used, it is a genuine Operating System for research purposes and evaluation of software because of the user control it provides. In addition, it gives good tooling and consistency across versions and its Open-Source nature and support for a comprehensive set of hardware. [1].

2.2.1 kevent

FreeBSD introduces in its design `kqueue` and `kevent`, a kernel event notification mechanism

[1] `https://freebsdfoundation.org/our-work/research/`

for capturing events meeting certain conditions that the kernel has received. This method is highly efficient in comparison with `epoll` from the Linux kernel, which has to perform different system calls for each event that wants capture. Linux `epoll` system is dedicated to system files, and workarounds are needed for dealing with events not related directly with files (such as timers or semaphores) [2].

`kevent` system call registers events with the queue and returns any pending events to the user. Events get organized in an internal data structure `kevent` which will contain the data of the event.

2.3 Inter-Process Communication

A shared memory model provided by the operating system allows different processes to perform operations over the same domain; this model is dual with a message passing one [18]. This duality implies that whatever model we assume for performing compartmentalization can be transformed into the opposite by applying the necessary changes. Further from the changes in the code structure, we should consider the performance costs because the system design will optimize for either message passing or shared memory, one of the process communication systems.

Because of modern systems design, modeling inter-process communication will use a shared state between the receiver and sender of the data; still, this can be hidden behind the abstractions and only expose the channel to both communication components. A clear example of implementing message passing in modern systems is through UNIX sockets, which will buffer the contents sent through it and allow processes to read and write through the socket descriptor, similar to reading from files.

2.3.1 gRPC

gRPC [3] is the de-facto standard in terms of Remote Procedure Calls; it gets an advantage from Protocol Buffers [30], a way to pass a message with typed-fields. Based on the declared message structure, gRPC automates all messages serialization/deserialization. Moreover, it offers different mainstream communication channels such as UNIX sockets or addresses comprehended in the network stack (attaching the server to a network port of an IP address assigned to the device). Message passing is used both in the call to the RPC and in its response, whose messages might have different structures.

[2] `https://long-zhou.github.io/2012/12/21/epoll-vs-kqueue.html`
[3] `https://grpc.io`

It works in a server/consumer fashion, where a process will attach a procedure to the channel; each time it receives a message, it runs the procedure. Thus, procedures will be a snippet of code that executes for each connection to the gRPC server. gRPC lets to work with sequence streams for continuous communication in both directions. Also, there is the possibility to define asynchronous RPC: synchronous RPC calls will clock until a response arrives from the server, while in the asynchronous model, the client has two different instances to ask the server for beginning an RPC processing and for asking for its termination, leaving room in the meanwhile to the client for performing operations in such a way that the RPC processing by the server can be performed concurrently with procedures from the client.

2.4 Nginx

Nginx [27] is a high-performance HTTP server written in C language [23], Nginx's purpose is modularity and optimization in query processing, including load balancing, traffic management, massively scalable content caching, programmability and automation. It also serves as a reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server.

Nginx HTTP server implementation depends on worker processes (specified ahead of the server runs), which will process the requests performed to the server. Therefore, the load-balancing intent is to leave the workers with similar workloads at processing events. Usually, for better performance, the number of workers will be as much as the number of cores to preserve information locality while intensive processing happens.

Nginx already implements coarse-grained isolation between the workers as they are OS processes and through an internal communication mechanism. The communication only occurs with the Nginx master process, which spawns the worker processes and sets up the channel of communication with them through UNIX domain sockets [4]. The master process will control and maintain all worker processes through the created channels [5].

A user can define several configuration parameters for Nginx execution, which will optimize its efficiency. These parameters are static at runtime, although Nginx allows a hot-reconfiguration to apply a different set of parameters while running. Example parameters are the number of workers spawned, the utilization of a proxy, specific related information of server location (like listening port or server root directory), SSL specifications. Among these parameters, we have the option of tunneling files requested without copying the contents or enabling file caching, which will leave recent accessed files' contents to be faster accessed next time.

Nginx takes advantage from the OS mechanisms for its connection processing methods, this

[4] https://stackoverflow.com/questions/35212208/inter-process-communication-for-apache-server-and-nginx
[5] https://docs.nginx.com/nginx/admin-guide/basic-functionality/runtime-control/

is established in the events module of the source code [6]. For FreeBSD operating system it uses the `kevent` methodology presented in subsection 2.2.1.

2.5 Novel compartmentalization techniques

There is a broad literature about the introduction of efficient sharing of memory resources without incurring a high penalty imposed through the system abstractions. The systems research community has developed ideas to attenuate the penalty of having several processes in different working domains to let the software do not incur much overhead while defining new working domains and switching to them. Apart from the performance gains offered by these mechanisms, existing pieces of software not securing different agents in themselves can take advantage of the novel designs.

The proposed mechanisms often address memory and the communication with it as the main overhead to overcome as it will be the primary mechanism for transferring data. The case of message-passing-based systems would have an upper limit in the size of the message without going through memory because of the hardware limitations on message passing on the processor bus's size (while XPC tries to avoid the data copy [5]). Still, the message passing mechanism will have the same constraints as the model using shared memory if it stores the message in memory at some point.

2.5.1 lwC's

Light-weight contexts [19] are an OS abstraction that provides independent schemes of protection, privilege, and execution state within a process. Each process may include several lwC's, each with its own virtual memory space, file descriptors, and access capabilities.

lwC's offer new capabilities extremely expensive in previous models:

- It implements a way to roll back efficiently: the process can return to a prior recorded state by discarding the context it is executing and returning to its creation point.
- Isolated address spaces: lwC's within the process may have different perceptions of the memory space, e.g., isolating sensitive network-facing components or isolating different user sessions.

[6] `https://nginx.org/en/docs/events.html`

- Privilege separation: in-process reference monitors can arbitrate and control access

lwC's prototype implementation extends FreeBSD 11.0 with the new abstractions that this compartmentalization method provides through 8 new system calls (`lwccreate`, `lwcsuspendswitch`, `lwcdiscardswitch`, `lwcoverlay`, `lwcrestrict`, `lwcsyscall`, `lwcclose`, `lwcgetlwc`) to be effective, concretely testing Apache, Nginx, PHP and OpenSSL. Both the prototype software compending the FreeBSDg-11 modified kernel [7], the userspace library [8] and the Nginx version implementing lwcs[9] is publicly available [10].

2.5.2 MMP

Mondrian Memory protection [34] proposes a memory model where different protection domains share memory through a segmentation-based protection scheme, contrasting with page-based systems. The segmentation-based model offers a finer granularity of space division. The OS will solely write the permissions table; still, by having a two-level cache table accessing, the overhead provoked by this decision imposes is addressed. MMP provides multiple protection domains within a single address space (regardless it is physical or virtual). It does not need a rework of existing binaries of the ISA as addressing remains the same while changing the OS abstractions.

The permissions to the different protection domains are specified in a compressed permissions table to reduce the space overhead of managing such a fine-grained memory version.

Mondrian Memory Protection exhibits as an enhancement of Linux 2.4.19 kernel version called Mondrix [35]; it enforces isolation between kernel modules, helping to detect bugs in the design as cross-compartment accesses will be observable.

2.5.3 Cheri

Capability Hardware Enhanced RISC Instructions (CHERI) [33, 36] is a hardware extension with additional information per memory object, which indicates the permissions of individual processes to that object. It extends 64-bit MIPS ISA with byte-granularity memory protection. It supplies protection primitives to the compile, language runtime, and operating system. It proposed a capability coprocessor and tagged memory, which allows having accelerated checks for memory accesses. A memory capability is an unforgeable pointer that grants access to a linear range of address space; Capability addressing occurs before virtual-address translation.

[7] `git://jameslitton.net/lwckernel.git`
[8] `git://jameslitton.net/lwclibs.git`
[9] `git://jameslitton.net/lwcningx.git`
[10] https://www.cs.umd.edu/projects/lwc/

2.5.4 XPC

XPC [5] (Cross Process Call) focuses on the problem of having efficient IPC on microkernels; it proposes-assisted OS primitive, enabling direct switch between IPC caller and callee without trapping into the kernel (which is slow). It allows message passing without copying. It implemented the prototype based on RISC-V architecture and ported to sel4 [16] and Zircon [11] microkernels.

2.5.5 Intel-MPK

Memory Protection Keys [12] is a feature found on Intel's Skylake and later architecture CPU's. It is a userspace hardware mechanism to control page table permissions. It works by tagging memory pages with protection keys using four previously unused bits; in other words, we can use up to 16 distinct keys to tag our pages. It is still necessary to perform a system call to tag the pages with a given key. To allocate and free a key, we need to go through the kernel. It adds a new accessible register (`pkru`) and two new instructions (`rdpkru`/`wrpkru`) for reading and writing the new register.

Intel-MPK has its subsequent adoption at the Operating System layer:

- `libmpk` [21] is a software abstraction for MPK that virtualizes the hardware protection keys to eliminate the protection-key-use-after free problem while providing accesses to an unlimited number of virtualized keys. It integrates it on the Linux kernel (which implements MPK support since kernel version 4.6 [13]) through a userspace library providing a software abstraction.
- FreeBSD platform provides a Protection Key Rights for User pages (`pkru`) [14].

Donky

Donky [25] is a hardware-software co-design for strong in-process isolation based on memory protection domains. Domain switches are performed in userspace, minimizing switching overhead and kernel complexity. It demonstrates secure V8 sandboxing, software vaults, and untrusted third-party libraries. The authors provide an evaluation of its security and performance for RISC-V and Intel-MPK.

[11] https://fuchsia.dev/fuchsia-src/concepts/kernel
[12] https://www.kernel.org/doc/html/latest/core-api/protection-keys.html
[13] `http://lkml.iu.edu/hypermail/linux/kernel/1603.2/01145.html`
[14] `https://www.freebsd.org/cgi/man.cgi?query=pkru&sektion=3&manpath=freebsd-release-ports`

ERIM

ERIM [29] is a novel technique providing hardware-enforced isolation with low overhead on x86 CPUs even at a high ratio of context switches. It combines Intel-MPK with a binary inspection. Without compiler changes, it demonstrates its functioning in the Linux kernel.

2.5.6 CODOMs

COde-centric memory DOMains (CODOMs) [32] provides finer grained isolation between software components with effectively zero run-time overhead. A prototype evaluated in a x86 simulator demonstrates the functioning of the proposal. Its composable natura allows to plug program semantics into the hardware.

2.6 Performance Metrics Counters

Performance Metrics Counters (PMC) let to sample through devoted events the efficiency of a system. Measure a system performance is a difficult task from different approaches:

- The metrics taken may be meaningful and give insight into what the system does while it has been measured.
- Measurements should be less invasive with the system primarily because sampling procedures may affect the system performance.
- They must expose the system's genuine efforts while processing.

The "Utilization, Saturation and Errors" (USE) methodology [15] defines how to obtain specialized metrics from any system. It is intended to be used for the early identification of bottlenecks.

The invasiveness of measurement usually offers more detailed and concrete results, still, to force the system to perform statistics collection willtake time of processing and even affect the microarchitecture which state cannot be recovered and is being exploited by the program to evaluate. Across all the system stack we can find facilities for performing measurements:

- Hardware performance counters: Chip vendors include on their latest shipped processors a set of registers with information that keeps updated with the events occurring at the

[15] `https://www.brendangregg.com/usemethod.html`

architectural and microarchitectural level such as with branch prediction, microarchitecture internal operations issued, or cache measurements like evicted data, hits or misses. A complete set of performance counters from the Intel processor architecture is publicly available on Intel's website [16].

Hardware performance metric counters have been proposed too for identifying the misuse of a system by periodically sampling the counters and with the obtained information denote if there are microarchitectural elements forced to have unexpected behavior (high number of misses or frequent contention) [22], which is indeed a non-deterministic helpful method.

- Operating systems may have a similar way of collecting information when the kernel is ruling the system by updating internal data structures related to information. This utility may differ among different operating systems: In FreeBSD `ktrace` tool can retrieve the kernel behavior in a log-based fashion.
- Software running on top of the OS may have its own logging procedures, letting information be recorded. This method may have higher overhead and perform confusion if the system is being measured from a lower-level layer, mainly because it will be measured as a legitimate part of the process despite the fact that its purpose is not information processing.

There are tools that can help to recover all the metrics through a single tool, this is the case of `dtrace`, which has different providers and can be used to get fixed statistics by attaching the tracer process to any process, from which the statistics will be filled up.

[16] https://perfmon-events.intel.com/

Chapter 3

Design

All compartmentalization techniques declared in section 2.5 treat the same issues when focusing on finer-granularity isolation of the system, regardless of where this isolation is applied. The central fact addressed is communication between different compartments that should have as least overhead as possible between different compartments; this is, how to avoid the OS to trap message passing mechanisms, a challenge already addressed by some compartmentalization techniques presented.

We measure the message-passing issue by applying this mechanism over software applying compartmentalization. As mentioned in [18], every model that applies shared memory model can apply message passing and reverse. Then, in software that does not apply compartmentalization, first needs to be defined the information that shall be shared across the different compartments and later perform that information sharing either by message passing or memory sharing. In both cases of information transmission, the program should be easily adapted to use any of both by minimizing the implementation details regarding the technique to the communication process, for which the program design must adapt.

This project directly addresses the security foundations of our systems, as we want to evaluate a system implementing a technique that isolates different components of a system. If the system can be assessed to be free of security vulnerabilities, then the additional protection coming out from compartmentalization would be needless, but no piece of software can be demonstrated as secure because it can suffer from bugs at any layer of the implementation, even more, when the different layers of the system are developed by different parties. Here we would like to aim for a hardware-software codesign [3], focusing on security. This co-design determines security properties across layers from which the more abstract levels will be aware in order to protect the systems; this idea has already been conceived in the aspect of speculative execution [9]. Still, that specifications can change through time like it already happened in the case of hardware-based leakage with Spectre-related attacks, which have been identified way after the deployment of

the hardware [17] or which have been foreseen ahead the hardware has been implemented [28]. This topic is something the security community is aware that needs to be undertaken to secure the development of modern systems [31].

Most of the work presented in section 2.5 claims for the existence of this co-design. The requirement for the co-design to be effective is that hardware admits high-level abstractions for a specific way of processing the information. The use of virtual memory pages (introduced on subsection 2.1.2) unveils this co-design but still at a low level of abstraction that only the hardware can manage. The idea is to expose similar behavior to the high-level software, in an Exokernel manner [6], which minimizes the OS intervention, maximizing the software exercise. These ideas are further explored in chapter 6.

3.1 Comparison of different systems

There is an inherent problem to performance measuring, that is about comparing different systems metrics, existing a substantial problem with measuring time while evaluating our system [20]. This work is not exempt from the same problem. However, by leaving low the bar of differentiation of the systems evaluated, we aim to reduce this problem. We aim to tackle this problem in the study performed during this work by trusting the bare-metal to perform effectively, building our evaluation system over it without any intermediate layer interfering with the measurements. Virtualization, although being useful to prove the software correctness, will not provide a feasible insight of the system performance, mainly because there are parts of the execution that may get overhead from a host processing (e.g., network packets will go first through the host OS processing stack to later be redirected to the guest one).

3.2 Fundamentals for a compartmentalization evaluation platform

We describe a series of requirements for a software piece to be able for a compartmentalization evaluation platform:

- **Compartmentalization adds an extra level of security**, it is not compartmentalized by default, and its design does not give a concrete way of how the program must be compartmentalized. Its design is not planned to harmonize an abstraction concept with its isolation. Still, that concealment can be applied through a compartmentalization technique. As a requisite, the lack of compartmentalization in the system must be a weakness, and applying compartmentalization adds a horizontal security layer across the secured components. It allows sensitive data from different procedences to coexist in the memory layout.

- **Is an userspace program**, so any possible overhead from process context-switch or actions that require from the OS intervention, will impose a performance overhead.
- **It interacts with its environment**: it is not a processing-intensive program. It produces responses to events it receives, so it is measurable in the range of states it can be. We can consider this as the dynamic response.
- **It admits tuning** through different substantial parameters that will directly affect the program's processing of the results. We consider this as the static response.

For choosing a system that works for our evaluation criteria, we can go for either a process that already introduces some compartmentalization in its design, like sandboxing of browser tabs. Alternatively, enforce compartmentalization to an abstraction in the design of a piece of software. This project uses the latter option because it allows better measurement of the overheads of the lack of compartmentalization. Moreover, its introduction to a widely used software can benefit in the long term, whereas this technique is efficient.

3.2.1 Nginx serving as an evaluation criteria

For the prototype of our design, we decide to use Nginx as it meets the prerequisites we define for a piece of software that allows different compartmentalization techniques evaluation and, therefore, will be our yardstick for systems compartmentalization. In our case, the abstraction that can be compartmentalized and not considered in the original design is the multiple connections that the software manages. `Connection` is a HTTP packet header that controls if the network connection stays open or not after the current transaction finishes [1]; if the value sent is `keep-alive`, the connection is persisted an not closed. The webserver implements its method for handling the transactions corresponding to a single connection for this header. For Nginx, the connections will have separate access of resources at has its method for managing the packets of a single connection [2]. This header is used only for measurement purposes of our platform as it is deprecated from HTTP/2 onwards [3].

The purpose of the worker processes and their isolation is for performance and organizational matters, while security is assumed to exist through the defined abstractions in Nginx design. Specifically, the requests served by Nginx will fall in the same trust domain, sharing the same memory resources. Because of this fact, any latent memory-related vulnerability in the system stack (it may happen to be architectural, in the OS, or the code itself) will expose information from one connection workspace to another. Such a variety of vulnerabilities exist and allow

[1] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Connection`
[2] `https://www.nginx.com/blog/http-keepalives-and-web-performance/`
[3] `https://datatracker.ietf.org/doc/html/rfc7540` - section-8.1.2.2

illegal memory accesses; an example of this software is the very recent vulnerability CVE-2021-23017 [4], which allows an attacker to do a memory overwrite. Another vulnerability in Nginx is CVE-2018-16845 [5] which allows process memory disclosure, while if compartmentalization is applied, the disclosure only affects the connection-related information.

We perform our evaluation over the x86 architecture. In contrast, we can readjust the evaluation system to another architecture as ARM, RISC-V, or MIPS processor in which other hardware-based compartmentalization are outlined.

3.2.2 FreeBSD for the base operating system

We argue in section 2.2 the benefits of choosing FreeBSD as the Operating System where to evaluate because of its stability and commitment to research, determining, even more, the decision process that we expose in continuation.

Because of the existence of a version of Nginx using the lwC's compartmentalization technique we discuss in section 2.5, it reassures our decision to use FreeBSD as the operating system for performing the evaluation. However, the MMP compartmentalization technique leaves without being evaluated in this project because of the lack of support for FreeBSD from this platform, even though Mondrix is presented as a solid model for testing compartmentalization benefits. In the same way, we can not test lwC's using a Linux-based system for our evaluation. Mondrix base Linux kernel dates from 2.4.19 kernel version dates from 2002 [6] (Mondrix was published in 2005)), while FreeBSD-11 dates from 2015 [7] (lwC's was published on 2016). Then, it strongly supports our decision to use such an Operating System as the criterion for the evaluation system (considering that the Mondrix project is not open sourced).

In the case of other projects aiming for modifications in the hardware components, we see that CHERI opts out for a FreeBSD-based system too, and at the point of this project, it is still in active development. Still, the evaluation of CHERI in this project is out of the scope due to the additional work it needs to set up a platform that runs the system and the subsequent modifications of Nginx for getting it working.

We perform our evaluation over the x86 architecture. In contrast, we can readjust the evaluation system to another architecture as ARM, RISC-V, or MIPS processor in which other hardware-based compartmentalization are outlined.

[4] https://www.cvedetails.com/cve/CVE-2021-23017/
[5] https://www.cvedetails.com/cve/CVE-2018-16845/
[6] https://lkml.org/lkml/2002/8/4/154 - https://en.wikipedia.org/wiki/Linux_kernel_version_history
[7] https://www.freebsd.org/releases/11.0R/schedule/

3.2.3 Software needing from compartmentalization

Other software to consider for evaluating compartmentalization is web browsers, already implementing sandboxing tabs by websites (as in Chromium engine [8])). Web browsers can benefit from finer-grained compartmentalization, impacting their abstractions in the low level by getting further benefits from acceleration.

Finally, we must consider other servers like mail servers (Qmail [9]) or other HTTP servers (like Apache2 hhtpd web server [10]). Operating systems may be excellent software to apply hardware-accelerated compartmentalization techniques on a set of abstractions they propose. However, they lack the characteristic of being a userspace program, so any compartmentalization technique pertinent to a layer over the hardware will not be appropriately applied (the kernel itself resides on a low level of abstraction in the system). We will further discuss the development of compartmentalization evaluation techniques for these software pieces in chapter 6.

3.3 Applying compartmentalization in Nginx design

The existing version of Nginx for lwC's introduces compartmentalization through its mechanism. However, it does not apply an OS-based compartmentalization method for it (fork) due to the software complexity. This project approaches the technical debt issue that existing software may have to introduce security mechanisms to make the program inherently secure against domain-crossing attacks.

Compartmentalization projects use for their evaluation either mature projects or microbenchmarks. In the case of mature projects, we can find projects like Apache2 `httpd` webserver, V8 engine, or PHP evaluation software. Microbenchmarks focus on evaluating the performance of the compartmentalization-related mechanisms. However, Nginx codebase is slightly more complex than other presented projects: Nginx server contains 136202 lines of C code, while Apache2 httpd last server version contains 45532 lines of C code [11]. In addition to supporting Nginx election for the evaluation system, during the realization of this project, Nginx turned to be the webserver with highest usage statistic in the whole internet [12]. The requirements of modern web applications may support and motivate this trend [13].

Methods for evaluating system's performance such as SPEC 2006 [13] are not suitable for this evaluation mechanism because it already defines the workloads and configuration parameters

[8] `https://www.chromium.org/developers/design-documents/site-isolation`
[9] `https://cr.yp.to/qmail.html`
[10] `https://httpd.apache.org/`
[11] These measurements have been performed using `cloc` software utility
[12] `https://w3techs.com/technologies/history_overview/web_server` - `https://news.netcraft.com/archives/category/web-server-survey/`
[13] `https://www.nginx.com/blog/nginx-vs-apache-our-view/`

of the programs to be run, giving more an overview of raw performance. The system must include some compartmentalization to be evaluated, while the programs do not admit such modification. We want to demonstrate the additional security that the piece of software will gain through the mechanism implemented at negligible performance costs. If the program is inherently secure and doesn't need from compartmentalization, in its design as there are no different parties making use of it, there will be no room to perform compartmentalization.

We will perform two types of measurements during the evaluation, using HTTP keepalive connections and performing a new connection per new HTTP request sent (one per packet). Keepalive connections will set a flag in the HTTP packet noting that the connection must be kept active until the client notifies to close it (or it falls dead due to a long interval of time without receiving packets from the connection it corresponds to). Keepalive has a direct effect over Nginx compartmentalization, as the compartmentalization in Nginx is performed per-connection (as we state in subsection 3.2.1), this means that for each new HTTP packet indicating a new connection will cause the creation of a compartment in the Nginx server-side, with the costs that it implies and the additional context switches. The experiments in chapter 5 show performance differences from using one or another method.

A fact we already have mentioned is the cost of adopting new techniques on existing infrastructure, while the cost has an objective (measurable) dimension and a subjective one (not measurable), and comes in 3 different flavors:

- **Technical cost**: The cost of having the new system deployed, measured in which system's components must change. It is also the cost in terms of performance gains/loss of embracing the technology.
- **Time cost**: The cost in adapting the system to use the new feature is commonly a matter of human cost for adaption. It also affects the evolution and maintenance that the system must receive.
- **Social cost**: The concrete adaptation of the technology is necessary, but it also needs to be adopted by society. This adoption can be present in the way of software updates or behavioral changes of the user.

We aim to evaluate those three costs in the platform we design; mainly, the technical cost will prime, but it is wortk discussing the other types of costs.

The more different a system is from its predecessor, the higher the overall transition cost will be. Although the features this system implements beat the state-of-the-art ones, the standard ones will likely be deployed because the new system would need to be adopted and fit in the existing scenario. Nonetheless, in our case study, there is a lack of a predominantly deployed system that introduces in-process compartmentalization techniques. Then, we can diminish the social costs not to be considered and focus on the technical costs while assuming the time costs

inherent to the technology that cannot be lessened. However, we will discuss the generalization of the usage of compartmentalization techniques in chapter 6.

3.4 Message passing for compartments communication

Microkernels and other more secure alternatives are not adopted widely because their design compels for an efficient message passing mechanism, not standardized. We introduce in our case study gRPC (presented in subsection 2.3.1) as the communication mechanism between compartments. Nonetheless, we can use any other message-passing method for efficient compartmentalization, with a particular focus on the promising hardware acceleration techniques for this method presented in section 2.5.

For implementing gRPC in any piece of software as a compartmentalization technique, we should change the program layout to perform separation through procedure execution. The program structure will likely organize through procedures; if not, slight modifications in the code will enable it. The more modular the software is, the easier it is to implement compartmentalization. We still need to pay special attention to the memory that can be shared and only read; its declaration and use from different compartments will boost performance. We need those memory sections to exist and not be written read before creating the compartment. The Operating System will control when a compartment writes to a shared memory region by deduplicating it.

The gRPC messages should be defined with their typed fields. Those fields should be adequated to the program requirements for the necessary data transmission between compartments when applying compartmentalization. As explained, the definition of the message structure automatically generates the serialization/deserialization operations.

Chapter 4

Implementation

In chapter 3 we point out the methodology for achieving a well-defined compartmentalization methods evaluation. In this section, we call attention to the details for performing that concrete evaluation which we will able to carry on and later show some of its results in chapter 5. As specified, our platform will focus on introducing OS processes-based compartmentalization in Nginx, using gRPC as the communication mechanism between the compartments in a FreeBSD Operating System. Having separate gRPC and Nginx systems working separate in FreeBSD is pretty straightforward [1], but their combination requires a bit more endeavor.

4.1 Applying gRPC to Nginx

We perform 2-step modification over an existing version of Nginx, which depends on lwC's to introduce a way of compartmentalization in the codebase. We aim to transform this compartmentalization implementation from being based in lwC's to be done through OS processes. Before implementing gRPC utilities, we transition from version customized 1.9.15 [2] of the codebase (on which the modification introducing lwC's is based) to release 1.19.8 [3], such modification is to ensure an optimal behavior and that the software keeps up-to-date. This transition is trivial, as the significant code, which is the events processing through `kqueue` (we present this fact in section 2.4), remains unchanged. The changes to make this version of the code work under lwC's are related to wrappers' use instead of the original functions in the new code that calls the

[1] `https://nginx.org/en/docs/configure.html` - `https://grpc.io/docs/languages/cpp/quickstart/`

[2] based on tagged as release-1.9.15 and with the commit ID 7a719646fafaeea6e087a9ab531f8e23c4e24935 an with commit ID d5abe38e12e8a32ea913bb361c61f2eaa1608ee0 in `lwcnginx` repository

[3] tagged as release-1.19.8 and with the commit ID 1b8771ddcbc53dee494aeae1449f3b6403e7299d

original variants. Concretely, those changes are the substitution of `ngx_http_close_connection` calls by `ngx_http_close_connection_later` and the change of the references of the variable `ngx_event_timer_rbtree` to the dereference of itself. With this version update change to the compartmentalized version of Nginx, we ensure later projects can establish their foundations and code on the work presented on this project.

To transition to introduce gRPC functionality in Nginx `kqueue` event module we must transition the codebase to C++ due to gRPC offers a C++ library [4] and there's no gRPC library version in C language. We choose C++ as the language from the offer of lgRPC libraries due to the object files link compatibility of C and C++. Then, we must rewrite in C++ all the files that we want to implement the interface to interact with gRPC libraries. In our case we only perform the transition to the file `src/events/modules/ngx_kqueue_module.cpp`. That transition needs from type specifications in the code, for example the assignation of value to `kcf` variable from the `ngx_palloc` function needs to be casted to `(ngx_kqueue_conf_t *)` because C++ doesn't admit the disambiguations from `void*` type.

To introduce the process-based compartmentalization, we take the existing approach from lwC's. Each of the code connections is assigned to a different compartment (a light-weight context in this case) and port it to use another kind of compartments (OS processes). As explained, lwC's will share the same PID but different lwC's IDs within a process. Only a single lwC's can run per process at a time, and then, switch to another process is declared explicitly (using the ID of the lwC's to be switched to), idling in this process the lwC's that performs the switch. Within the switch operation, each lwC's can perform message passing.

There are two main aspects to work on when introducing this functionality to mainstream OS-based compartmentalization: how to perform efficient message passing and define an explicit context change.

In the lwC's case, both context switch and message passing are part of the same primitive. We can have similar behavior by using a general message passing system, as when the receiver reads from a channel will be locked until the sender has written that channel.

In the concrete implementation of Nginx with lwC's, all the switches happen between the kevents' handler light-weight context and the lwC's it creates (no other mechanism spawns lwC's).. Then the spawned processes do not perform any switch between them, only to the kevents' handler lightweight context. The information passed in the switch contains the information about an event to be processed, which only knows the process that calls `kevent` (kevents' handler). All the processing information is included in the function `ngx_kqueue_process_events`.

Although the OS processes cannot designate the process to switch (as it would break the process isolation), we aim to produce a similar behavior through message passing, which first needs a shared channel between sender and receiver. Then is impossible to have any way to perform process communication within the context switch only and no shared elements.

[4] `https://grpc.io/docs/languages/cpp/`

Unix-like operating systems offer the chance of signaling another process by using its PID; we can explore this method as an alternative to the message passing mechanism in combination with memory sharing, a mechanism introduced in chapter 3. Its implementation requires more developer work and can introduce concurrency bugs that in our model are avoided through the abstraction gRPC provides.

As all the code that introduces compartmentalization resides in the same piece of software (Nginx), this task does not require any generalization using a global mechanism in the system (such as a file that serves to indicate a communication channel or a lock-free mechanism).

As we introduce in section 3.4, adding a message passing mechanism in the code layout needs some modularization of the code itself. In Nginx codebase, we bring in the concrete action handling and execution from retrieving the actions to perform. In the code snippet handling events, we also set up the compartments' creation and the channel for their communication. If there is more than a single simultaneous connection, it will provoke the spawn of a compartment.

Because we are creating OS processes, one of the two processes resulting from the `fork` call may try to communicate with the other process before it starts. In gRPC, communication happens in a server/client fashion, so if the process that tries the communication is the server, it will only be accepting requests. If the process trying to communicate is the client, then it will fail when contacting the server. We control this error by specifying that if the client cannot communicate with the server, it should wait for the server to be ready (as a directive in the communication context); another alternative would be synchronizing the server with the server process before establishing the connection.

We can conceptualize the code architecture to introduce an extra level of abstraction with the dissemination of connections. We produce a semantic relation of the HTTP connection with its abstraction in Nginx code, with the compartment that the compartmentalization technique will manage. In Figure 4.1 we show the organization of Nginx introducing generic compartmentalization mechanisms. Even though non-compartmentalized Nginx handled all the events in the same address space, we delegate to the receiver of the event to dispatch new compartments and send the events to process to the compartment it assigns. Here we observe a division between the control plane and the data pane of the program, as the elements will flow into the memory of the program. However, the information they contain will not affect or propagate until they reach the compartment they should belong. Different compartments from the same worker will not process simultaneously because of the constraint imposed by the context switching model that only the execution inside a compartment is allowed if called from the event assignation.

We contemplate a similar construction in the connection between Nginx master and worker processes have and the events delegation and the events handler. Both receive a stream of packets that will reassign between the processes they have created. Worker processes are longer-lived and only terminate with the whole Nginx server, while a connection will shut down the process corresponding to the events handler of its compartment.

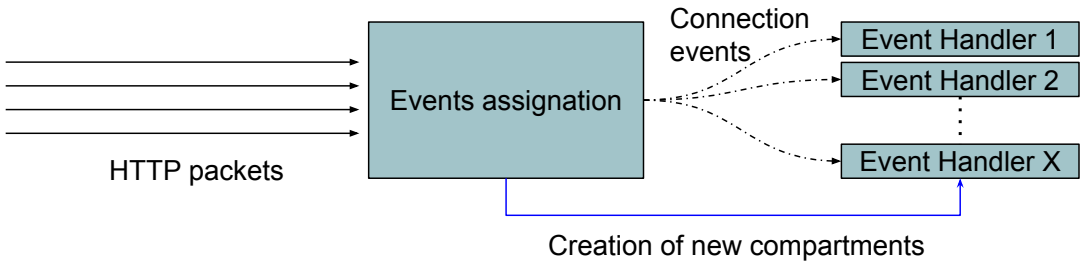


Figure 4.1: Internal architecture for compartmentalization based Nginx

Last, in the configuration build system we must introduce all the changes for linking gRPC utils and make the code C++ compliant. We do this by adding all the libraries from gRPC and `stdlib++`, additionaly to common utils needed for gRPC such as `protobuf`, `ssl` and `crypto` among others. Also we need to define and give the Nginx processes rights to write and read from the folder where the UNIX sockets used for communication in gRPC will reside (in our case we do it in a temporal folder called `/tmp/nginx_com`). In the resulting `Makefile` generated by the configuration of the build system we introduce the compilation of the gRPC message declaration for our case to be compiled before our new C++ file.

Code versions

For the experiment's environment, we set up systems with the necessary Operating systems: For lwC's evaluation, we use FreeBSD11 implementing all the lwC's related system calls; for the rest of the systems, we use a FreeBSD13 without any further modification.

We base our gRPC implementation on the version `v1.35.0` [5], which depends on Protocol Buffers version `3.14.0` [6] at the sime this project is realized.

The hardware utilized for evaluation variates and is specified in chapter 5. As well the Nginx versions are mentioned in section 4.1.

4.2 Measurement methodology

We already talked about Performance Metric Counters and the USE method in section 2.6. We implement a system that measures gRPC in its version (with and without compartmentalization)

[5] `https://github.com/grpc/grpc/tree/v1.35.0`
[6] https://github.com/protocolbuffers/protobuf/tree/v3.14.0

during execution. We select hardware counters concerning the microarchitectural events like cache events and branch prediction; we also select the number of instructions committed. We measure as well scheduling events happening at the OS level through the `dtrace` tool. We use as a measurement the rate of processed events by Nginx with the time it takes to generate a response for a per request.

We tune the parameters in our experiments that let to decide the number of processes. We discuss in chapter 5 the entanglement between these parameters, the obtained metrics, and the design decisions. We automate all the sampling for obtaining the metrics and plots with minimal interaction for ease of reproducibility of results.

4.2.1 Plotting results

We use two different methods that let us descriptively visualize the results for plotting results.

Flamegraph [8] software [7], which lets to represent performance genuinely: The performance metrics have recorded the success of events with a particular frequency, from those events, we can aggregate which ones fall in the same piece of code. A graph outlines the resulting aggregation, where there are stacked horizontal bars representing functions. The vertical stacking illustrates the calling stack (a function calls the one above it). The length of a bar means the number of times a function execution was interrupted through sampling.

The other system for displaying results is `matplotlib` [14], a widely known python library that implements methods for plotting data in various ways. It is necessary to use a python script that processes and aggregates the data for its correct visualization.

[7] `https://github.com/brendangregg/FlameGraph`

Chapter 5

Evaluation

In this section, we present the results of experimenting with the system proposed in chapter 3 with the defined implementation in chapter 4. Previously to obtaining the results that we present, we test the correct functionality in the software side of the whole stack (gRPC, Nginx, and FreeBSD with and without lwC's) in a virtualized controlled environment. We are going to expose those prior results in section 5.3 and then jump into the results from running the stack in real hardware in section 5.4.

From now on, we will refer to the Nginx version implementing OS-based process communication with gRPC internal communication between the processes as "gRPC Nginx". The same applies for the entire Nginx (the one we find on the official repositories), which we refer to as "baseline Nginx" and the same for lwC's with "lwC's Nginx". Also, some notation as `Xw` and `Xc` will mean that we are using `X` workers and clients respectively.

5.1 Metrics collection

To evaluate and compare the different compartmentalization techniques, we must define the metrics we want to collect under the system functionality.

The main problem of how to evaluate security is that security is a qualitative characteristic of a system. At the same time, the other metrics we can measure concerning performance are quantitative. As we expose in , there is a set of metrics across the processing stack, belonging to hardware, Operating System, and software response. Over the collection of that metrics, we perform consecutive experiments and interpret the results.

We directly collect software response statistics with the Apache HTTP server benchmarking tool [1] (refer to it as ApacheBench), which we use to stress out the Nginx webserver with multiple HTTP petitions. We run ApacheBench and collect all the metrics in a separate machine than where Nginx is running for the minor interference with the processing for our experiment. Apachebench lets to have different parameters for forcing the server with requests; among these parameters, we can choose the HTTP headers for Keepalive connections, the number of concurrent clients, the number of petitions performed, and the upper limit in the running time of the experiment.

5.2 Tooling

We develop a methodology that automates the retrieval of the wanted metrics. In the case of hardware performance counters we use `pmcstat` [2], which is in some terms the equivalent to `perf` tool in Linux Operating System. For performing the measurements, logically, they must be run inside the same machine, so we begin the intensive stressing of the Nginx webserver under the desired parameters, and while it is running, we measure the events we wish. For the stressing and statistics collection of Nginx response, we use Apachebench, presented in section 5.1.

5.3 Results using virtualization

We obtain all the experiments in virtualization from a running `qemu` instance virtualizing `x86_64` architecture with 10 CPUs and 8GB of memory. We evaluated both the performance of a system that implements lwC's (the requirements of that system to run were presented in subsection 2.5.1) and in one running the latest stable version of FreeBSD with gRPC (defined in section 4.1).

The results from the testing in the virtualized environment presented in Table 5.1 and Table 5.2 show that compartmentalization based on OS processes has a high overhead. The overhead has around one order of magnitude when transitioning from lwC's to gRPC compartmentalization model in Keepalive connections. While the overhead grows even more to 2 orders of magnitude in the case of transitioning from lwC's to gRPC based compartmentalization with a new compartment per request received.

Increasing the number of worker processes shows a more remarkable improvement in the gRPC case, although the worse overall results. In lwC's, increasing the number of workers from 1

[1] `https://httpd.apache.org/docs/2.4/programs/ab.html`
[2] `https://www.freebsd.org/cgi/man.cgi?query=pmcstat`

Experiment iteration	gRPC Nginx 1w	gRPC Nginx 4w	lwC's Nginx 1w	lwC's Nginx 4w
1	5724	8869	34760	43000
2	5009	9665	41528	44914
3	5010	10901	37802	50000
4	5002	5911	28934	47916
5	5008	11196	32703	44130

Table 5.1: Nginx requests served throughput during a 60s workload in a virtualized environment with 50 concurrent clients and Keepalive set, asking for a 612 bytes document.

Experiment iteration	gRPC Nginx 1w	gRPC Nginx 4w	lwC's Nginx 1w	lwC's Nginx 4w
1	111	805	6014	18215
2	54	707	5594	16777
3	54	1430	5637	15403
4	57	593	5780	16294
5	55	639	6123	17912

Table 5.2: Nginx requests served throughput during a 60s workload in a virtualized environment with 50 concurrent clients and new connection per packet, asking for a 612 bytes document.

to 4 supposes a throughput improvement from 1.03x to 1.72x with Keepalive connections and from 2.51x to 3.26x for connections without Keepalive header. In gRPC, we get those factors to go from 1.03x to 2.24x and from 5.34x to 26.48x, respectively, being even higher than the factor of workers' growth. We hypothesize that gRPC has a bottleneck in the side of gRPC OS processes management and context switches, which gets alleviated with a higher number of workers, which will reduce the number of connections assigned per worker, causing the creation of fewer compartments thanks to load balancing.

As an observation, we see some anomalies in the gRPC metrics in which the throughput falls into ranges with low variation; we hypothesize that may be the consequence of the similarity between all the requests (they ask for the same file) and the OS schedules the processes to answer them similarly, producing that several requests are resolved in a brief period of time. All the hypotheses fall out of this project's scope to be tested; still, they can be considered for future work.

5.4 Results using real hardware

With the methods described in section 5.1 we make a comparison of Nginx introducing process-based compartmentalization and baseline Nginx.

For comparing gRPC Nginx with baseline Nginx using Intel performance metrics counters which we presented at section 2.6 we use a machine with processor intel i7-8700 (Coffee Lake microarchitecture, 6 cores and 12 threads), 16GB of RAM.

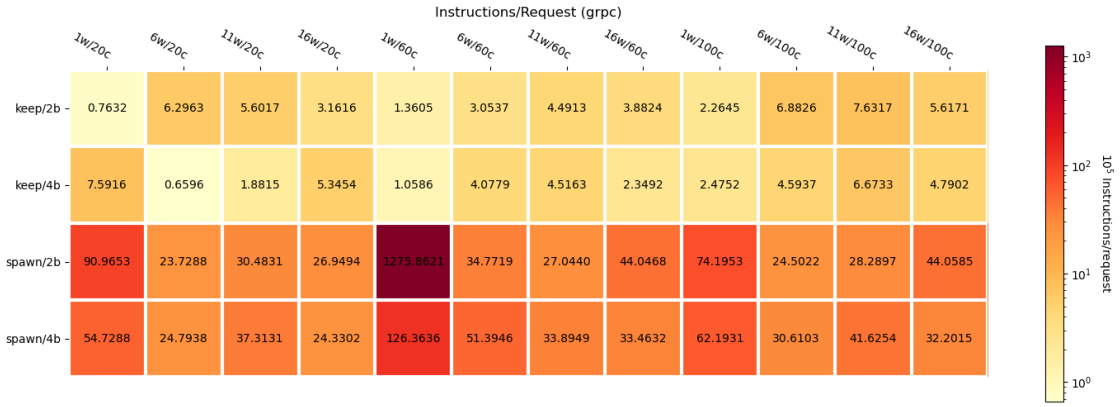
We define a set of parameters over we will perform the experiments as static for a run of the measurement, those parameters are:

- Number of clients: Taking the values of 20, 60 and 100
- Number of worker processes: Taking the values of 1, 6, 11 and 16
- Size of requested file: 100 bytes and 1000 bytes (represented by `2b` and `4b` respectively)
- HTTP packet header indicates Keepalive or not.

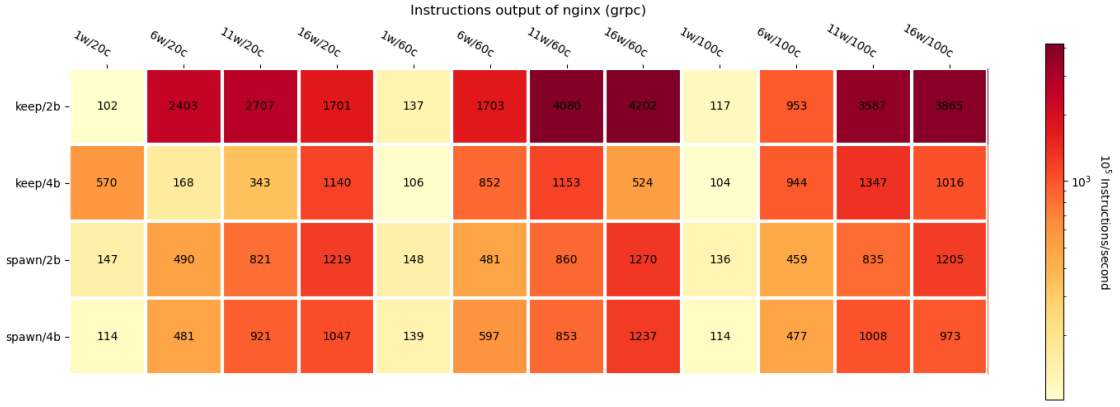
We obtain the next 3 metrics: $\frac{Instructions\ issued}{second}$, $\frac{Requests\ served}{second}$ and $\frac{Instructions\ issued}{Request\ served}$ and we plot in all the parameters combination for gRPC Nginx, baseline Nginx and the proportion of the results of gRPC and baseline Nginx ($\frac{gRPC\ Nginx}{baseline\ Nginx}$). The results are presented in Figure 5.1 Figure 5.2 Figure 5.3 . With these results we obtain a more descriptive insight of what's happening inside the system:

- In most of the cases, gRPC Nginx issues more instructions than baseline Nginx (Figure 5.3b), still that is not reflected in the proportion of requests served from the server(Figure 5.3c). In general, this shows a worse performance of gRPC Nginx over baseline Nginx.
- There are a couple of exceptions in which the request served by gRPC Nginx is moresignificant than in baseline Nginx; those are the cases with Keepalive connections and 1000 bytes of filesize transferred (Figure 5.1cFigure 5.2cFigure 5.3c). Still, the number of instructions issued per request answered is higher in mostly all the cases (Figure 5.3a), which means that baseline Nginx is spending more time in an idle state during processing.
- With a single worker in gRPC Nginx the number of instructions is significantly lower than with multiple ones, while differende is not oustanding with baseline Nginx (Figure 5.1bFigure 5.2bFigure 5.3b).

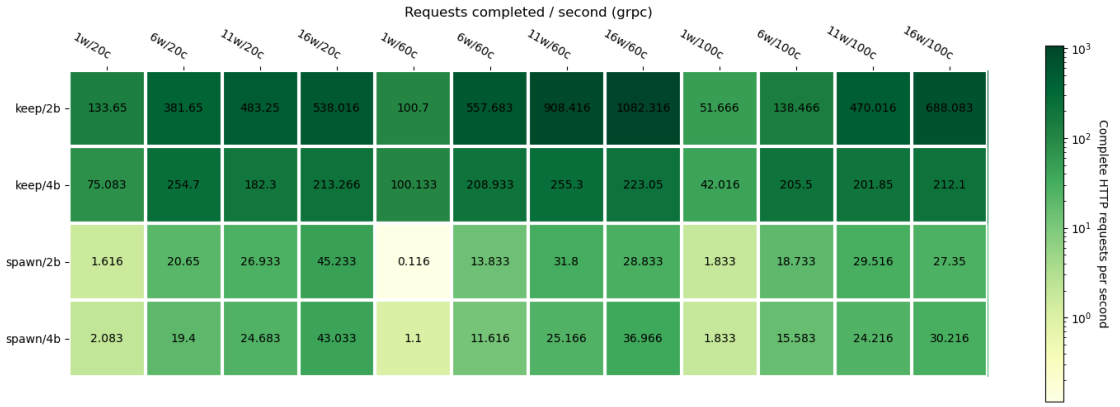
If we take a look in more of the Flamegraph based on instructions sampling (Figure 5.4 Figure 5.5 **?? ??** Figure 5.8 Figure 5.9), we will observe a curious effect happening. The total number of samples taken is much higher for gRPC Nginx than baseline Nginx with 11 workers processes and keepalive, but lower for the case with 1 worker process and no keepalive. In comparison, it is almost similar with 11 workers and not keepalive. If we take a look where does the processing spend cycles, we will find that userspace functions and `trap` by the kernel do take significant time for gRPC Nginx, while is a shred of available evidence that the cycles spent in `amd64_syscall` and `fork_exit` is big. Nevertheless, the userspace function is higher in having keepalive, which we



(a) Proportion of instructions issued regarding each compled request

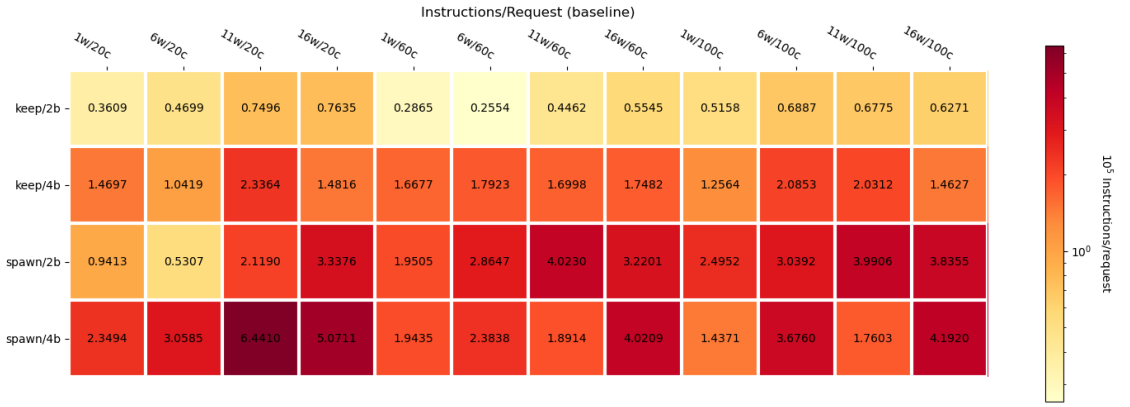


(b) Instruction emmited

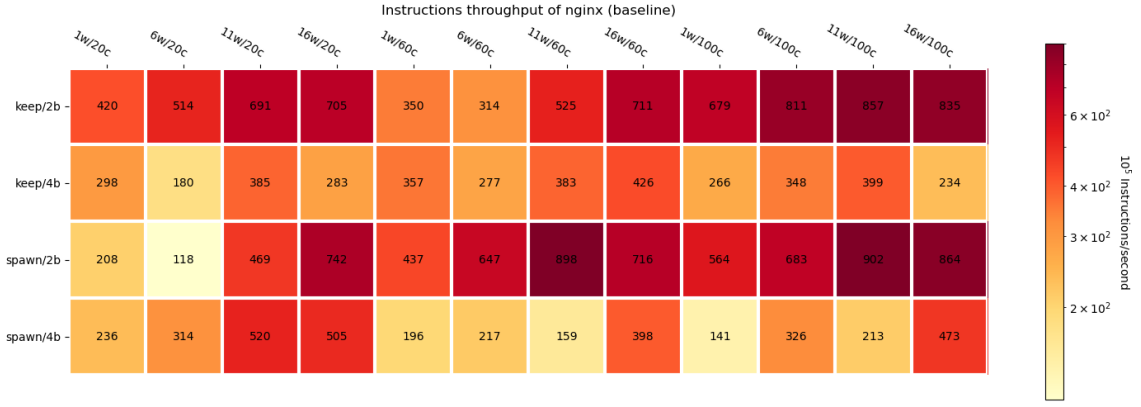


(c) Requests served throughput

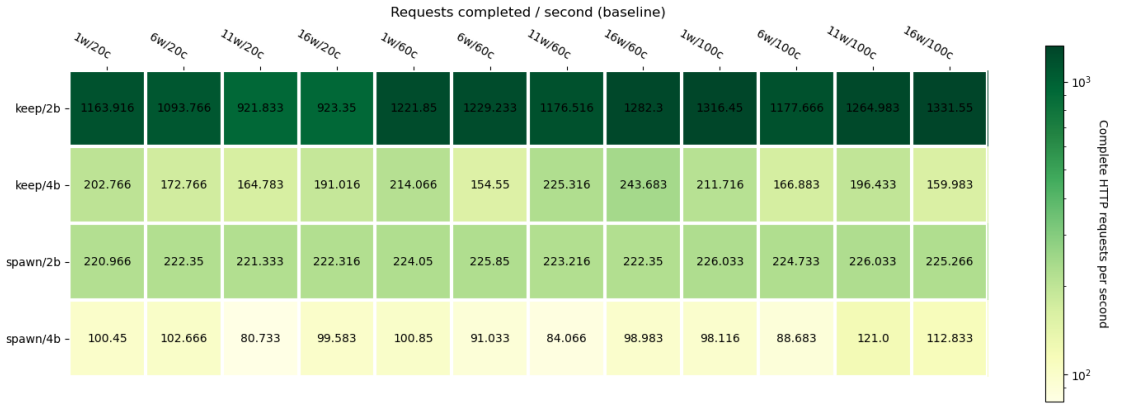
Figure 5.1: gRPC Nginx results



(a) Proportion of instructions issued regarding each compled request

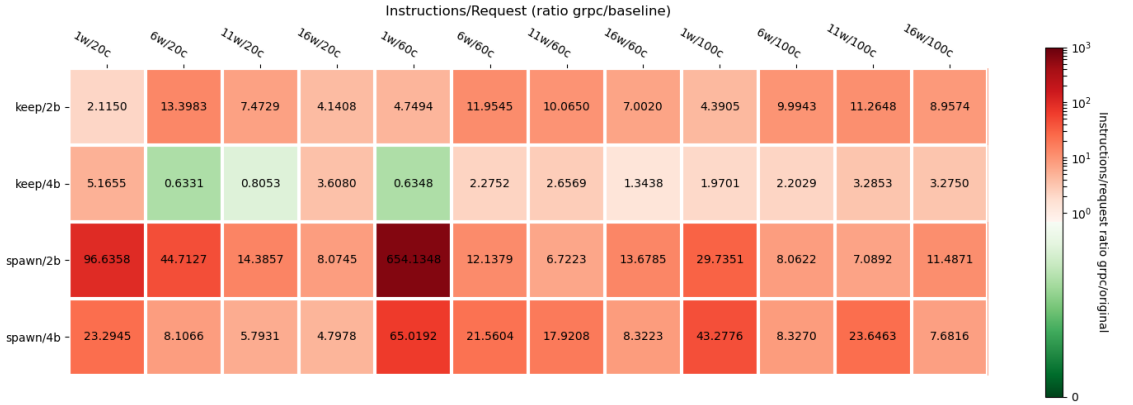


(b) Instruction emmited

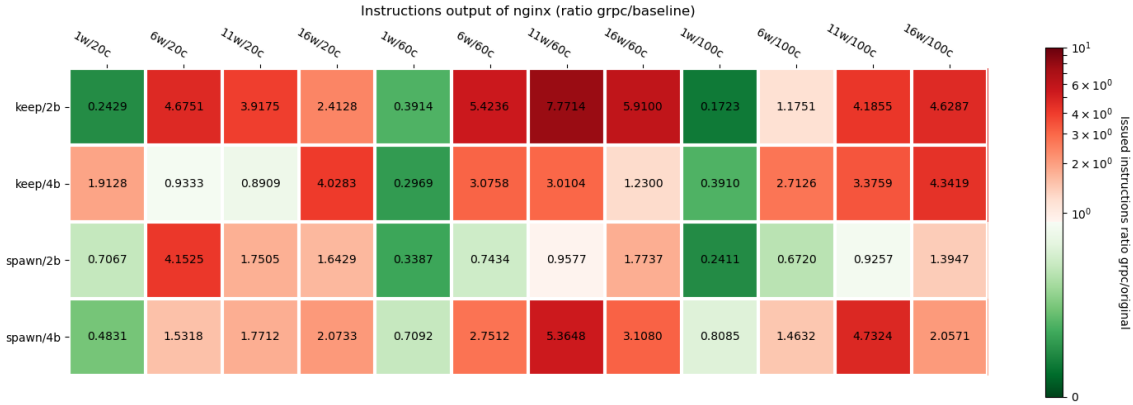


(c) Requests served throughput

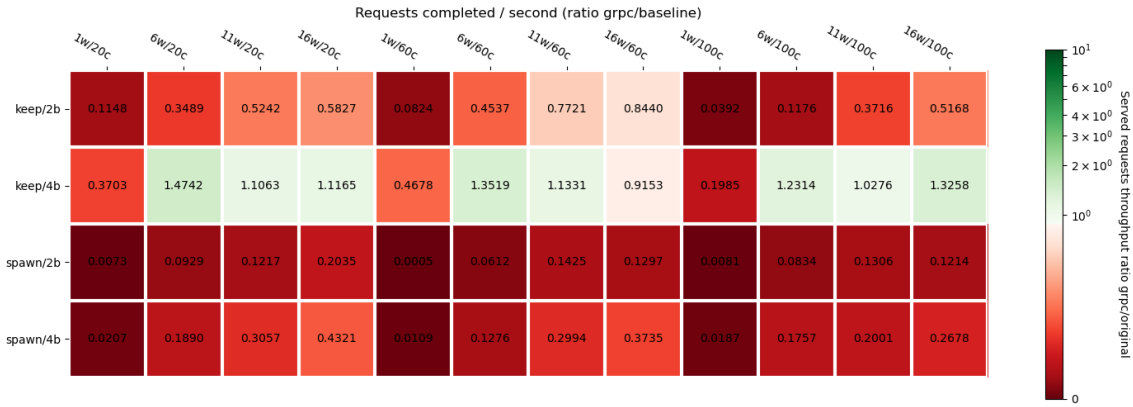
Figure 5.2: baseline Nginx results



(a) Proportion of instructions issued regarding each compled request



(b) Instruction emmited



(c) Requests served throughput

Figure 5.3: Proportion between gRPC Nginx and baseline Nginx results

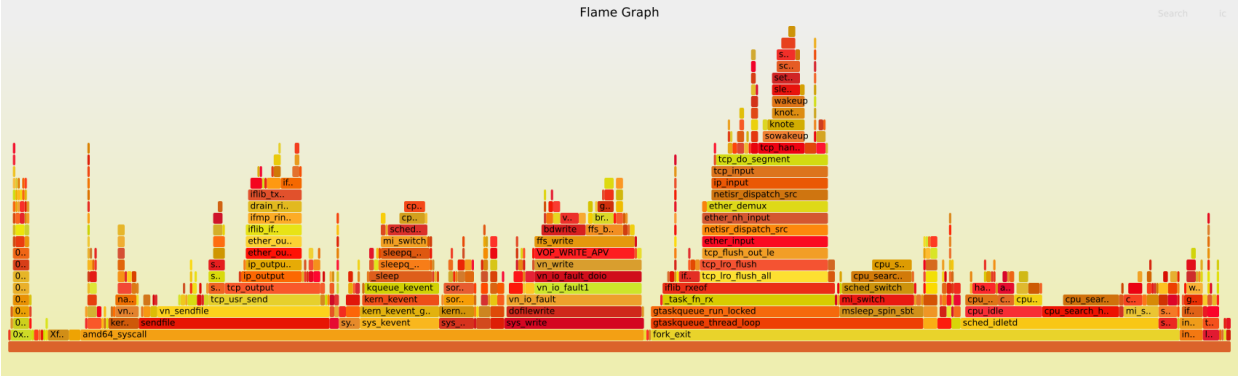


Figure 5.4: Flamegraph — Time concentration of operations in baseline Nginx, 11 worker processes, 60 client processes, filesize 100B, keepalive, 525 event sampled

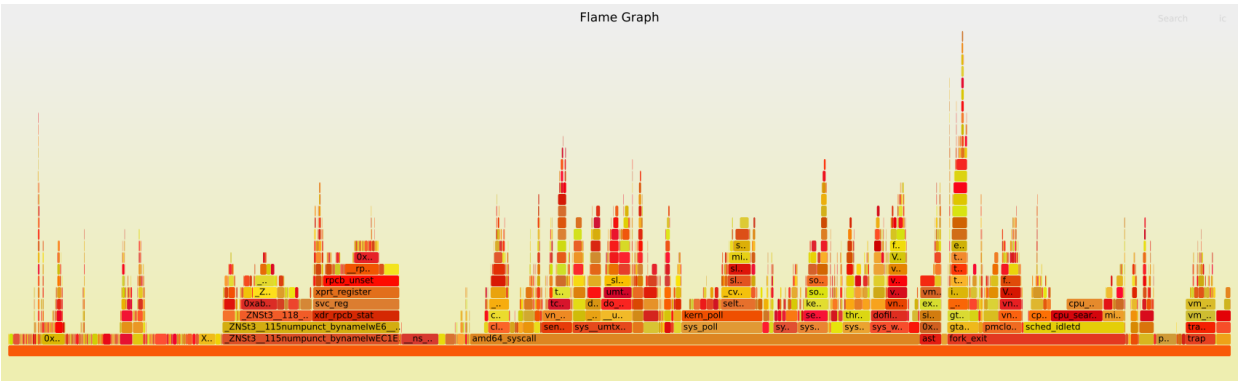


Figure 5.5: Flamegraph — Time concentration of operations in gRPC Nginx, 11 worker processes, 60 client processes, filesize 100B, keepalive, 4080 events sampled

can indicate to be directly related to the communication process of gRPC. In contrast, the presence of trap is much higher with no-keepalive, which may be due to the different process creation that will provoke more `vm_fault` when dealing with memory elements. The `amd64_syscall` usage may be due to the HTTP packet retrieval and files reading coming out from Nginx functionality.

For performing a throughput comparison of original Nginx with lwC's Nginx and gRPC Nginx we do our tests on a machine with processor intel i7-10700 (Comet Lake microarchitecture, 16 cores and 32 threads), 32GB of RAM.

In this case, we measure the time response of lwC's Nginx, gRPC Nginx, and baseline Nginx against incremental workloads (our X-axis). The resulting plots in Figure 5.10 show what we already confirmed in previous experiments: with a low number of compartments created in gRPC Nginx, the performance does not differ drastically from the baseline Nginx and lwC's Nginx, but this scenario will only happen in rare cases. About the scalability concerning the number of workers, the behavior is similar as in section 5.3 with greater scalability for the gRPC Nginx version, a less likely scenario, even more, when the stress comes from the number of clients.

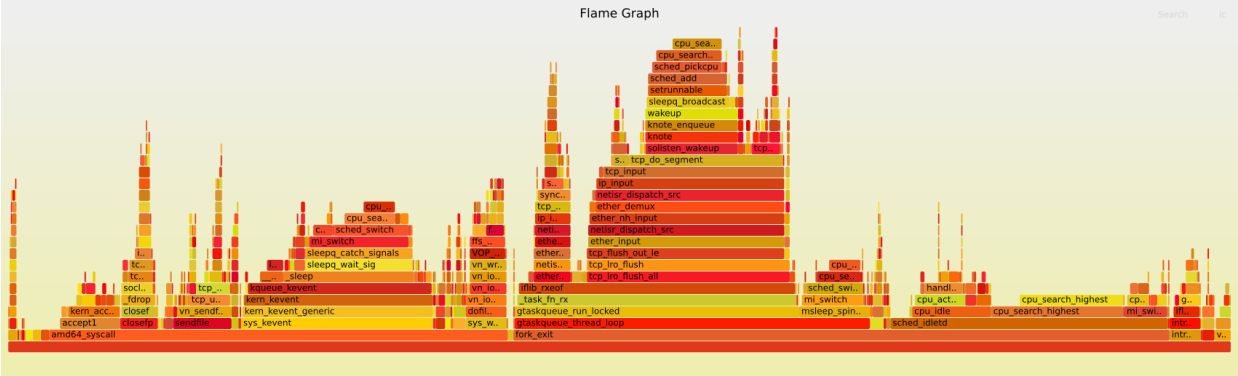


Figure 5.6: Flamegraph — Time concentration of operations in baseline Nginx, 1 worker processes, 60 client processes, filesize 100B, not keepalive, 898 event sampled

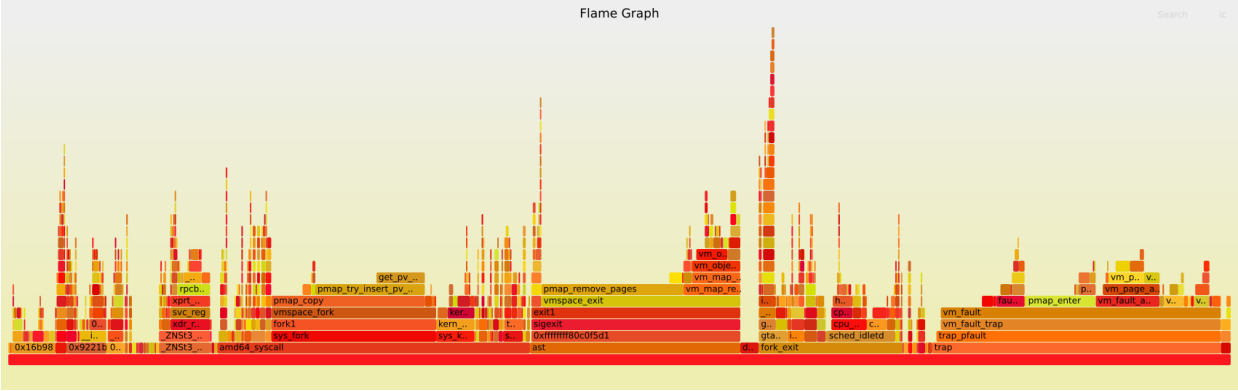


Figure 5.7: Flamegraph — Time concentration of operations in gRPC Nginx, 1 worker processes, 60 client processes, filesize 100B, not keepalive, 860 events sampled

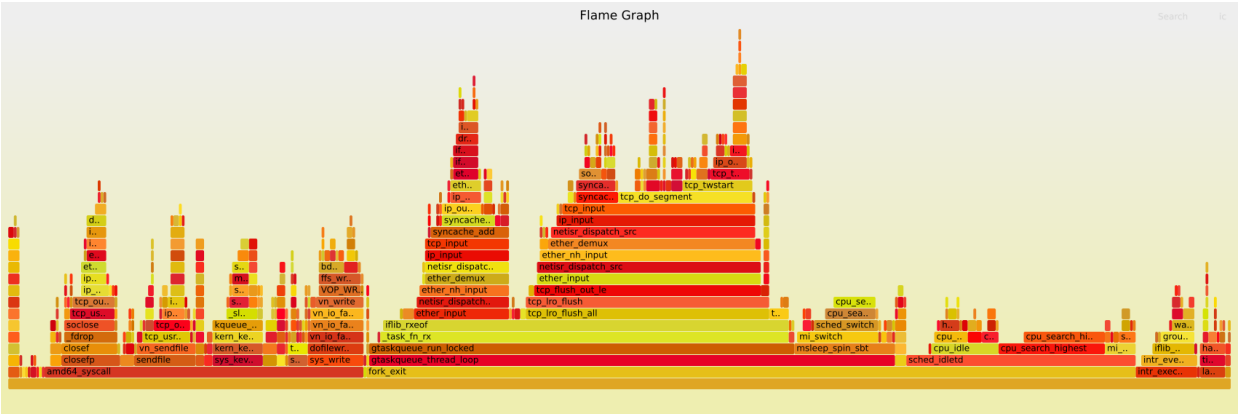


Figure 5.8: Flamegraph — Time concentration of operations in baseline Nginx, 1 worker process, 60 client processes, filesize 100B, not keepalive, 437 events sampled

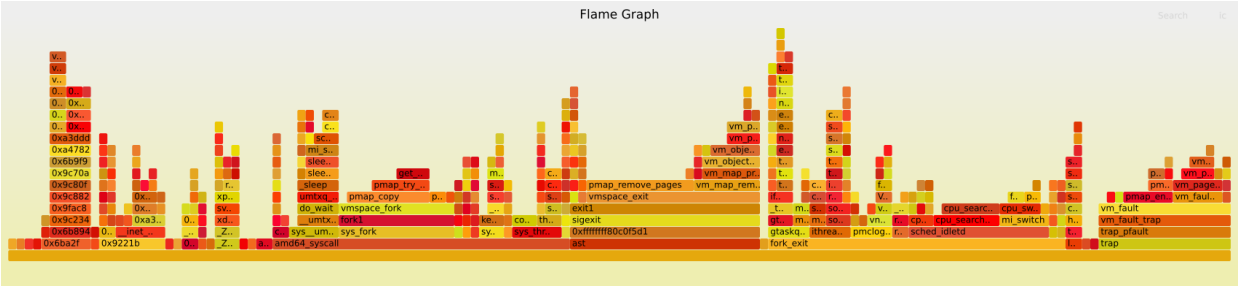
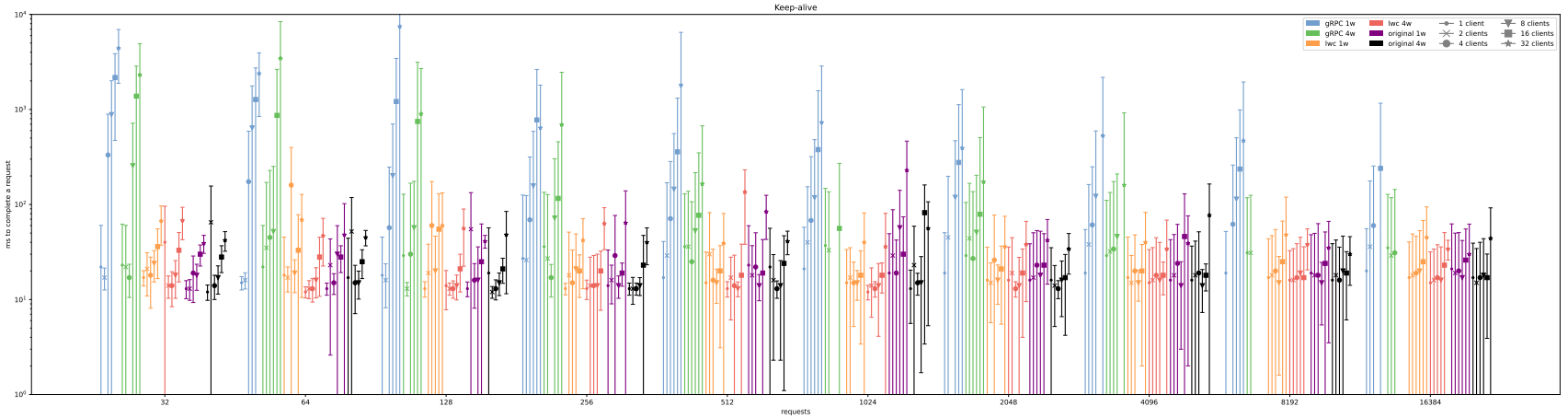
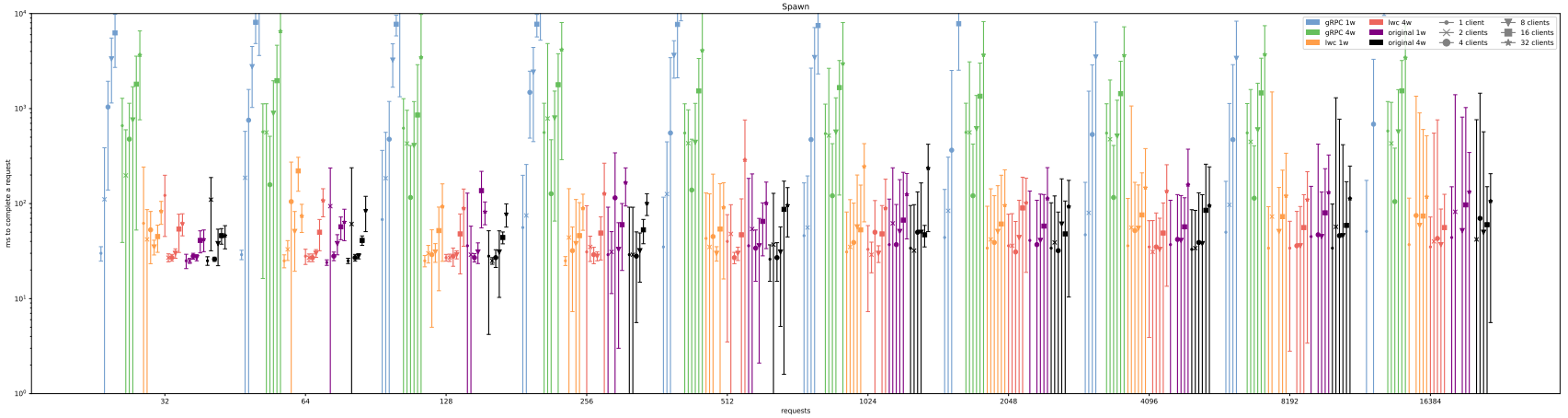


Figure 5.9: Flamegraph — Time concentration of operations in gRPC Nginx, 1 worker process, 60 client processes, filesize 100B, not keepalive, 148 events sampled

It is important to denote that the compartmentalization method we implement in gRPC Nginx is a generic mechanism that can be applied without modifying any of the rest of the stack elements (application-based). When we run in a lower level of specification for implementing compartmentalization, we will obtain superior results, primarily since a more significant part of the system stack will be designed for easing compartmentalization.



(a) HTTP requests in keepalive mode



(b) HTTP requests spawning new connections

Figure 5.10: Requests throughput comparison between gRPC Nginx, baseline Nginx and lwC's Nginx

Chapter 6

Related Work

6.1 Evaluation of hardware-based compartmentalization

Compartmentalization systems based on hardware-design modifications have been mentioned previously section 2.5. As exposed, no compartmentalization technique has been broadly adapted yet, leaving room for improvement in the evaluation of these systems.

The compartmentalization systems that propose hardware changes for efficient compartmentalization need cycle-aware simulators to be correctly compared against those that do not need new hardware elements. FPGA's implementation is an alternative, too, but the performance will be lower in this case and only can be used for testing the functionality. For cycle-aware simulation, it can be used zsim [24] or McSimA+ [1]. This simulation would help simulate hardware-based compartmentalization techniques like MMP [34], CHERI [36] or XPC [5]. However, the cycle-aware simulation needs exhaustive resources and takes a long time to produce results; the more complex the processing stack is, the longer it will take. This issue is difficult to solve, but there are works in the direction of making this simulation lighter while weeping it accurate [11, 37].

To compare the performance through different architecture platforms is intrinsically difficult because the implementation of different processors will have different way of interpret the metrics. For example, RISC and CISC architectures will have different meanings for the number of completed instructions: a CISC processor instruction may have a direct translation to several RISC instructions. In any case, there is no concrete translation from the instructions of onw model to instructions of the other model. The same applies for hardware specialized elements which may happen to exist in a concrete architecture, potentially reducing the performance

for the cases it is not used: an example of this would be with the usage from CHERI extending a RISC-V processor which implements XPC, at the time of getting the metrics for the baseline (the program being compartmentalized with through OS processes and a standard IPC system), those metrics are not going to reflect the performance of the platform if XPC extension was not added.

For this, it is essential to consider hardware design and how optimal it should be when introducing new primitives to it.

6.2 Memory protection

There are other ideas regarding the improvement of the program's use security for memory-related issues. However, their foundations are more focused on another type of enforcement of invalid access through semantic invalidity. All the techniques presented address orthogonal points where memory safety can be enforced and therefore can be combined. Still, any logic bug in the code as it happens with the exposed compartmentalization techniques at section 2.5 will potentially break the trust zones created by these ideas. The idea of plugging concepts semantically from the software into the hardware is not new and already mentioned in [32]. At the same time, its development is critical to introduce and substantial for the hardware-software co-design that we mention in chapter 3.

Hardbound [4] is an implementation of a *hardware bounded pointer* as an architectural primitive for enforcing spatial memory safety. Software defines bounds of concrete regions of memory when created and adheres to the dedicated pointer for checking the access validity at the time it happens.

High-level languages focused on low-level performance like Rust [15] fundamental purpose is a secure memory model layout, controlled by the programmer. The language definition itself imposes restrictions on how the memory elements should be controlled through program execution, limiting the propagation of memory values and distinguishing the usage of references and values of the objects used. This fact eliminates the possibility of accessing other memory elements when using the mechanism for accessing a concrete object. The concepts of memory management apply as well for high-level languages. However, the ideas of Rust are exciting because it keeps letting define a very fine-granularity of the program memory's layout as in C or C++.

Sanitization techniques focused on invalid memory accesses will crash the program whenever invalid access happens; AddressSanitizer (ASan) [26] presents a clear example of this technique. Memory sanitization works by when accessing alien memory contents, forcing the program to crash. It works by introducing a set of canaries between memory objects that will execute a crash routine when modified or accessed. Still, the case of direct access to the alien element without going through the canaries may happen. The technique is mainly used for detecting bugs in

the program and can be used to identify the logic bugs of the program, but can be applied as a defense mechanism too. However, it is not the best method due to the probabilistic nature of accessing a canary (even more if the program's memory layout is known) and the consequent bypass that may follow it.

In the same domain of probabilistic defense mechanisms approaches, we should consider a compartmentalization system with fewer compartments than the necessary ones. In this model, different untrusted agents reside in the same compartment. The idea resembles ASLR; without protecting the whole memory layout, it relies on the low probability of revealing an address. Although finer isolation is not provided, the probability that an attacker falls in the desired compartment is low; depending on the threat model, this could be a cheaper defense mechanism.

6.3 Comparison of metrics

Replication of results from existing projects in literature is a problem that the research community should address [1]. In this case, we are aware of the problem; this project is not different and, in the long-term, will suffer from the same issues. The implementation of our prototype depends on concrete pieces of software section 4.1, but the leading proposal of the design will remain unchanged (chapter 3). The more dependencies a project has, the harder it will be to maintain it up-to-date. Still, we believe that gRPC is a mature enough project for being easily reproducible. Depending on concrete versions, it can be easily updated because it is widespread among different projects that would suffer from legacy code problems the same way ours would.

6.4 Benchmarking systems

SPEC CPU benchmarks [2, 12, 13] propose a set of programs with computational-intensive characteristics that serve for evaluating the system's performance. It has been broadly explored in the systems community but needs the additional security dimension to consider it for an evaluation mechanism as the one we propose.

However, compartmentalization is not a technique that ensures total protection. Compartmentalization would be useless if the program or system suffers from any logic bug that will allow a privilege-scalation or leakage-related security issue. A compartmentalization technique reduces the consequences of a bug by limiting its scope.

[1] https://www.sigarch.org/a-checklist-manifesto-for-empirical-evaluation-a-preemptive-strike-against-a-replication-crisis-in-computer-science

Chapter 7

Conclusion

Compartmentalization acceleration has been around for a long and has received intense development in the last decade (as we present in section 2.5). One of the motivations of this emergency is the aiming for hardware-software co-design, which is happening incrementally. More room exists for specialized hardware elements because of Dennard scaling breakdown, while Moore's law continues in the form of dark silicon [7]. Facing this tendency, we propose the usage of a ground-truth comparison mechanism to evaluate several compartmentalization methods. Our study demonstrates the importance of a criterion to evaluate those systems and differentiate the type of metrics that should be measured.

From the measurements of the evaluation platform (shown in chapter 5),we observe that the compartmentalization purely performed at a high level of the stack (the userspace application) will undermine the whole system performance and introduce additional operations. However, the platform results from the design that lets us introduce arbitrary compartmentalization into a piece of software; in our case, we prototype it over Nginx webserver. We emphasize in the importance of correctly selecting the stack for evaluation to keep the evaluation system portable and with the fewer overheads.

Because of the hardware design is offering more room to specialized components, the research community is contemplating security-focused units [1]. However, the latest considerations from the research community focus on attacks Hardware-based novel techniques are presented in two ways: as an accelerator for a common practice (as it is context switch) or as a new dimension in which software can work (as with Memory Protection Keys). This introduction in the hardware dimension means that new security alternatives can be applied directly to existing software. In this aspect, we aim for a hardware-software co-design in which software abstractions can be lowered into the hardware to introduce an additional level of security. As a community, we

[1] https://www.sigarch.org/a-primer-on-security-threats-for-computer-architects/
https://www.sigarch.org/highlights-of-2020-security-conferences-for-computer-architects/
https://www.sigarch.org/highlights-of-2020-security-conferences-for-computer-architects-part-ii/
https://www.sigarch.org/architecture-and-hardware-security-research-early-2021/

should transition as quickly as possible from the lack of security mechanisms not implemented because of performance loss to eliminate all potential problems from cross-boundaries attacks.

A secure software framework would ease securing existing platforms and may be looked at by the research community. Its purpose would be to facilitate the developers the task of applying novel securing in their programs without incurring a high time cost.

In the existing literature of compartmentalization methods, their evaluation mechanism, albeit it is not standardized, is open source, which at least lets to make a comparison between the proposed systems by using the original work. Similarly, all the work produced during this project will be publicly open for its usage in possible future research. We advocate for this open science and research model and believe it is essential for scientific progress to be as honest and accessible as possible. Supporting a model with greater importance of artifact evaluation will help us to have better-designed software and results. This project sets an evaluation platform that will work for any compartmentalization accelerator and will facilitate obtaining metrics for the following projects. At the same time, it will institute the technique directly in a compartmentalized system project making it more likely to be adopted for future research.

Bibliography

[1] Jung Ho Ahn, Sheng Li, O Seongil, and Norman P Jouppi. “McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling”. In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2013, pp. 74–85.

[2] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. “SPEC CPU2017: Next-generation compute benchmark”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 2018, pp. 41–42.

[3] G De Michell and Rajesh K Gupta. “Hardware/software co-design”. In: *Proceedings of the IEEE* 85.3 (1997), pp. 349–365.

[4] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. “Hardbound: Architectural support for spatial safety of the C programming language”. In: *ACM SIGOPS Operating Systems Review* 42.2 (2008), pp. 103–114.

[5] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. “Xpc: Architectural support for secure and efficient cross process call”. In: *Proceedings of the 46th International Symposium on Computer Architecture*. 2019, pp. 671–684.

[6] Dawson R Engler, M Frans Kaashoek, and James O’Toole Jr. “Exokernel: An operating system architecture for application-level resource management”. In: *ACM SIGOPS Operating Systems Review* 29.5 (1995), pp. 251–266.

[7] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. “Dark silicon and the end of multicore scaling”. In: *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE. 2011, pp. 365–376.

[8] Brendan Gregg. “The flame graph”. In: *Communications of the ACM* 59.6 (2016), pp. 48–57.

[9] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. “Hardware-software contracts for secure speculation”. In: *arXiv preprint arXiv:2006.03841* (2020).

[10] FreeBSD Handbook. *The freebsd documentation project*. 2004.

[11] Sina Hassani, Gabriel Southern, and Jose Renau. “LiveSim: Going live with microarchitecture simulation”. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2016, pp. 606–617.

[12] John L Henning. “SPEC CPU2000: Measuring CPU performance in the new millennium”. In: *Computer* 33.7 (2000), pp. 28–35.

[13] John L Henning. “SPEC CPU2006 benchmark descriptions”. In: *ACM SIGARCH Computer Architecture News* 34.4 (2006), pp. 1–17.

[14] John D Hunter. “Matplotlib: A 2D graphics environment”. In: *IEEE Annals of the History of Computing* 9.03 (2007), pp. 90–95.

[15] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[16] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. “seL4: Formal verification of an OS kernel”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 207–220.

[17] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *arXiv preprint arXiv:1801.01203* (2018).

[18] Hugh C Lauer and Roger M Needham. “On the duality of operating system structures”. In: *ACM SIGOPS Operating Systems Review* 13.2 (1979), pp. 3–19.

[19] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. “Light-weight contexts: An {OS} abstraction for safety and performance”. In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 49–64.

[20] Ali Najafi, Amy Tai, and Michael Wei. “Systems research is running out of time”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2021, pp. 65–71.

[21] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. “libmpk: Software abstraction for intel memory protection keys (intel {MPK})”. In: *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 2019, pp. 241–254.

[22] Mathias Payer. “HexPADS: a platform to detect stealth attacks”. In: *International Symposium on Engineering Secure Software and Systems*. Springer. 2016, pp. 138–154.

[23] Dennis M Ritchie. “The development of the C language”. In: *ACM Sigplan Notices* 28.3 (1993), pp. 201–208.

[24] Daniel Sanchez and Christos Kozyrakis. “ZSim: Fast and accurate microarchitectural simulation of thousand-core systems”. In: *ACM SIGARCH Computer architecture news* 41.3 (2013), pp. 475–486.

[25] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. “Donky: Domain keys–efficient in-process isolation for RISC-V and x86”. In: *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2020, pp. 1677–1694.

[26] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. “AddressSanitizer: A fast address sanity checker”. In: *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*. 2012, pp. 309–318.

[27] Rahul Soni. *Nginx*. Springer, 2016.

[28] Po-An Tsai, Andres Sanchez, Christopher W Fletcher, and Daniel Sanchez. “Safecracker: Leaking secrets through compressed caches”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 1125–1140.

[29] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. “{ERIM}: Secure, efficient in-process isolation with protection keys ({MPK})”. In: *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 2019, pp. 1221–1238.

[30] Kenton Varda. *Protocol Buffers: Google’s Data Interchange Format*. Tech. rep. Google, June 2008. URL: http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html.

[31] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W Fletcher. “Opening Pandora’s Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data”. In: ().

[32] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. “CODOMs: Protecting software with code-centric memory domains”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 469–480.

[33] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. “Cheri: A hybrid capability-system architecture for scalable software compartmentalization”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 20–37.

[34] Emmett Witchel, Josh Cates, and Krste Asanović. “Mondrian memory protection”. In: *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. 2002, pp. 304–316.

[35] Emmett Witchel, Junghwan Rhee, and Krste Asanović. “Mondrix: Memory isolation for Linux using Mondriaan memory protection”. In: *Proceedings of the twentieth ACM symposium on Operating systems principles*. 2005, pp. 31–44.

[36] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. “The CHERI capability model: Revisiting RISC in an age of risk”. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE. 2014, pp. 457–468.

[37] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. “SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling”. In: *Proceedings of the 30th annual international symposium on Computer architecture*. 2003, pp. 84–97.