Cape Peninsula
University of Technology

DEVELOPMENT OF COMPUTE-INTENSIVE WEB APPLICATIONS WITH NATIVE DESKTOP PERFORMANCE

by

MOEGAMAT ZAHIR TOUFIE

Thesis submitted in fulfilment of the requirements for the degree

Doctor of Information and Communication Technology

in the Faculty of Informatics and Design

at the Cape Peninsula University of Technology

Supervisor: Dr B. Kabaso

District Six
November 2025

DECLARATION

I, Moegamat Zahir Toufie, declare that the contents of this thesis represent my own unaided work, and that the thesis has not previously been submitted for academic examination towards any qualification. Furthermore, it represents my own opinions and not necessarily those of the Cape Peninsula University Of Technology (CPUT).

Signed [illegible] **Date** 1 November 2025

ABSTRACT

A web browser Execution Environment (EE) with the capability of hosting and executing Compute-Intensive Applications (CIAs) with native-like performance, have long been sought after. To that end, various technologies have been developed, the most prominent one being WebAssembly Programming Language (WASM) and to a lesser extent JavaScript Programming Language (JS). However, having CIAs achieve native desktop performance within a web browser EE has been elusive, primarily because it lacks the required systems architecture and design with which to support it.

With that in mind, an empirical hypothesis of: ”A web browser EE that can host and execute CIAs on any device with native desktop performance can be created”, and a research question of: “What software architecture and design does a new web browser EE need to comprise of in order to be able to host and execute CIAs with native desktop performance”, were formed. Methodologically, this study is rooted in Design Science Research (DSR), together with Concept-Knowledge (C-K) theory, and a positivist philosophical position.

Furthermore, a deductive approach, experimental strategy, mono-method quantitative data collection and cross-sectional time horizon were used. The delineation was limited to the web browser EE only. Additionally, all living organisms were excluded from the study, limiting ethical considerations to the development of a new web browser EE. The Structured/Systematic Literature Review (SLR) aided in identifying the primary literature which, as expected, articulates the promising results of WASM, while other technologies such as Rich Internet Application (RIA) was also touched upon.

Further to that, we also introduced problem areas within benchmarking, that being Operating System (OS) noise, after which we provided insights into how one might go about mitigating against OS noise. We then presented our system architecture and design, based on Virtual Machines (VMs) and their lightweight container sibling. To that end a Linux-based prototype called System23 (SYS23) was presented, incorporating components such as Control Groups (Cgroups), namespaces and Secure Computing Mode (Seccomp), which together form the SYS23 enclave.

We then benchmarked the prototype using PolyBench/C, which was liberally used with the literature that was discovered. The findings suggest that the prototype supports our hypothesis and achieves a performance improvement over its WASM counterpart and comes to within 0.45% of its native equivalent. Through that, the singular research question was answered and a foundation for future research to build upon was also provided. Furthermore, our methodological, theoretical and practical contributions moved our field of study forward.

ACKNOWLEDGEMENTS

I wish to thank:

- My supervisor, to whom I would like to extend my heartfelt gratitude for his invaluable patience, ongoing feedback and continual guidance, as well as his willingness to impart his extensive knowledge and expertise to me. This journey would also not have been possible without the help of the defence committees and their members, who generously provided their knowledge, expertise, and guidance.
- My doctoral classmates and cohort members, to whom I am grateful for their help, guidance, late-night feedback sessions, and never-ending moral support. My thanks also extend to the CPUT Centre for Postgraduate Studies (CPGS), research assistants, librarians, and research participants from the university, who supported and inspired me.
- My family, especially my parents, wife, and children, to whom I would be remiss in not mentioning. Their faith in me has kept my motivation and spirits high throughout this journey.
- My cats, for all of the entertainment, head bumps, emotional support and company during those long nights and never-ending days.

Furthermore, no proprietary software was used while pursuing this research, all tools used are open-source and freely downloadable. The experiments and prototype development, together with the generation and collection of data, mainly utilised Linux, Vim, and R together with RStudio, while LaTeX and TeXstudio were used in authoring this thesis.

DEDICATION

For *Mum*, who passed away suddenly in 2020, you innocently bought me my first book on computers and subsequently a Commodore 64, which infinitely piqued my interest in the field. May you rest in peace knowing that those two altruistic acts, changed my life forever.

For *Dad*, with whom I worked casually as a teen, as exciting as it was to earn my own money and learn several useful skills, it also quickly made me realise that working in construction was far less interesting and exciting than working with computers.

PUBLICATIONS

Chapters from this thesis have been accepted to be presented at a conference and/or accepted to be published in a scholarly journal resulting in four published papers, as follows.

Initially, **Chapters 1** and **2**, where **Chapter 2** was the primary focus, were presented at an IEEE international conference in Mauritius, as well as being published in the related IEEE conference proceedings.

> Toufie, Z. & Kabaso, B. 2023. The Next Evolution of Web Browser Execution Environment Performance. In *2023 International Conference on Artificial Intelligence, Big Data, Computing and Data Communication Systems (icABCD)*. icABCD 2023. Durban: Institute of Electrical and Electronics Engineers: 1–7. doi: 10.1109/icABCD59051.2023.10220564.

Then, **Chapters 1** through **3**, where **Chapter 3** was the primary focus, were published in a Springer Nature scholarly journal. This paper also presented our methodological contribution.

> Toufie, Z. & Kabaso, B. 2024a. OS Noise Mitigations for Benchmarking Web Browser Execution Environment Performance. *Discover Computing*, 27(1): 1–29. doi: 10.1007/s10791-024-09471-4. Dataset doi: 10.25381/cput.28595438.

Next, **Chapters 1** through **4**, where **Chapter 4** was the primary focus, were presented at an IEEE international conference in Indonesia, as well as being published in the related IEEE conference proceedings. This paper also presented our theoretical contribution.

> Toufie, Z. & Kabaso, B. 2024b. A Next Generation Web Browser Execution Environment. In *2024 International Conference on Data and Software Engineering (ICoDSE)*. ICoDSE 2024. Indonesia: Institute of Electrical and Electronics Engineers: 1–6. doi: 10.1109/ICoDSE63307.2024.10829873. Dataset doi: 10.25381/cput.28595438.

Lastly, **a summary of the completed thesis** was published in an ACM scholarly journal. This paper presented our completed study together with our practical contribution.

> Toufie, Z. & Kabaso, B. 2025. A High Performance Web Browser Execution Environment for Compute-Intensive Applications. *ACM Transactions on the Web*. 1-30.
> (*acceptance pending*)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ABBREVIATIONS AND ACRONYMS

2D	Two Dimensional
3D	Three Dimensional
ABI	Application Binary Interface
AI	Artificial Intelligence
AOT	Ahead-of-Time
API	Application Programming Interface
BPF	Berkeley Packet Filter
C	C Programming Language
C++	C++ Programming Language
C-K	Concept-Knowledge
Cgroup	Control Group
CIA	Compute-Intensive Application
CPU	Central Processing Unit
CPUT	Cape Peninsula University Of Technology
CSRF	Cross-Site Request Forgery
DAC	Discretionary Access Control
DSR	Design Science Research
EE	Execution Environment
FFI	Foreign Function Interface
GID	Group ID
GPU	Graphics Processing Unit
GWT	Google Web Toolkit

IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-Process Communication
IRQ	interrupt request
ISA	Instruction Set Architecture
JIT	Just-In-Time
JS	JavaScript Programming Language
JSVM	JavaScript Virtual Machine
LLVM	Low-Level Virtual Machine
LLVMIR	Low-Level Virtual Machine Intermediate Representation
LSM	Linux Security Module
MAC	Mandatory Access Control
MIME	Multipurpose Internet Mail Extension
NaCl	Native Client
NCSA	National Center for Supercomputing Applications
NIS	Network Information Service
NSA	National Security Agency
NUMA	Non-Uniform Memory Access
OS	Operating System
PAPI	Performance Application Programming Interface
PID	Process ID
PNaCl	Portable Native Client
POPI	Protection of Personal Information
POSIX	Portable Operating System Interface
RIA	Rich Internet Application
RISC-V	Reduced Instruction Set Computer Version 5

SCoP	Static Control Parts
Seccomp	Secure Computing Mode
SELinux	Security-Enhanced Linux
SLR	Structured/Systematic Literature Review
SPA	Single Page Application
SYS23	System23
TCP/IP	Transmission Control Protocol/Internet Protocol
UID	User ID
UTS	Unix Time-Sharing
UX	User Experience
VM	Virtual Machine
W3C	World Wide Web Consortium
WASI	WebAssembly System Interface
WASM	WebAssembly Programming Language
WWW	World Wide Web
XML	Extensible Markup Language
XSS	Cross-Site Scripting
ZTA	Zero Trust Architecture

GLOSSARY

BogoMIPS

Refers to the Bogus Millions of Instructions Per Second measurement, which is a rough unscientific gauge of processor performance that was developed by Linus Torvalds, the creator of Linux.

Execution Environment

An Execution Environment is a computer software system that can execute computer code, the most common ones are Operating Systems and Virtual Machines.

Hyperthreading

Intel Corporation’s proprietary simultaneous multithreading implementation.

Kendall Rank Correlation Coefficient

Kendall’s Rank Correlation Coefficient or simply Kendall’s Tau, is a nonparametric measure of the strength and direction of association that exists between two variables measured on at least an ordinal scale.

OS Noise

Refers to the duration during which a CPU executes instructions unrelated to a specific application task assigned to that CPU, even when the task is ready to run.

Pearson Correlation Coefficient

Pearson’s Correlation Coefficient is the most common way of measuring a linear correlation. It is a number between –1 and 1 that measures the strength and direction of the relationship between two variables.

Portable Operating System Interface

Is a set of standards established by the IEEE Computer Society to ensure compatibility across operating systems. These standards define the application programming interface (API), as well as command-line shells and utility interfaces to maintain software compatibility with Unix variants and other operating systems.

Simultaneous Multithreading

Is the process of executing multiple threads simultaneously within the context of physical CPU cores that are split into two or more logical cores or threads.

Single Page Application

A Single Page Application is an application made up of a single web page, refreshed via JavaScript.

Spearman Rank Correlation Coefficient

Spearman's Rank Correlation Coefficient measures the strength and direction of association between two ranked variables. It is simply the Pearson Correlation Coefficient of the rankings of the raw data.

Symbolic Expression

Symbolic Expression or s-expression, is a human-readable representation of semistructured data, which is composed of atoms (symbols), lists and strings. Symbolic expressions have their roots in the Lisp computer programming languages.

System V

System V (Five) Unix, developed by AT&T Bell Labs, is one of the earliest and most influential versions of the Unix OS. It played a critical role in shaping the development of modern Unix computer systems.

System23

System23 is the Execution Environment that this study seeks to develop.

Zero Trust Architecture

A Zero Trust Architecture operates on the principle of trusting no one and nothing by default. Traditional IT network security follows the castle-and-moat model, where it is difficult to gain access from outside the network (castle), but once inside (crossed the moat), everyone is automatically trusted.

CHAPTER 1

PREFACE

The research discussed within this thesis proposes the development of a new web browser Execution Environment (EE), specifically one with native desktop performance. This research was initiated subsequent to performing a Structured/Systematic Literature Review (SLR), as detailed throughout chapter 2, on what web browser EEs currently exist together with and focusing on their current state in relation to performance. Also subsequent to the SLR, this research identified the lack of performance by web browser EEs, as the key problem hindering Compute-Intensive Applications (CIAs) from being widely developed for the Web.

This chapter examines the foundational proposal of this research by briefly discussing the following:

Section 1.1. *Introduction*

Section 1.2. *Research problem*

Section 1.3. *Objectives and research question*

Section 1.4. *Theoretical background and related work*

Section 1.5. *Design, methodology and ethics*

Section 1.6. *Delineation*

Section 1.7. *Outcomes, contribution and significance*

Section 1.8. *Thesis structure*

1.1. Introduction

For a long time now, all web browsers have had the desire of being able to host and execute feature-rich, compute-intensive, and complex applications or simply CIAs, within their EE, while striving to achieve native desktop-level performance. This need has been present across any device, be it a desktop computer or mobile device which has a web browser installed on it as stated by Vilk and Berger (2014), Powers et al. (2017), Wagner (2017) and Wen et al. (2020). Web browsers have pursued various long-term relationships to this end since their birth in 1990 (Berners-Lee, 1990), but every technology relationship has always fallen short of expectation, which eventually led to the technology being abandoned.

There was Adobe Shockwave formerly Macromedia Shockwave, which appeared in early 1995. It was anticipated that one could use Adobe Shockwave to create CIAs, but one could never create anything more than simplistic colourful games and media. The limiting factors created intentionally by its software architecture and design have led to its engineered inability to create anything more complex, as can be deduced from its reference guide by Yeaman and Dawson (1996). Adobe Shockwave still lingers around to this day, but its support was officially ended by Adobe in 2019.

Soon after, there were Java Applets, which appeared in the middle of 1995. Its software architecture and design were different and delivered much more feature-richness and thus one was able to create more feature-rich applications, but not necessarily compute-intensive ones. Java Applets always felt clunky though, because one had to download and install a plug-in to gain the functionality. This inherent shortcoming caused it to be viewed as an outsider by all web browsers as stated in Topic (2016). After numerous highlighted security shortcomings, many web browsers eventually disabled the functionality by default, so one had to specifically turn the functionality back on again if one wanted to use it. This was really the start of the end for Java Applets and it too is no longer supported, since 2019.

Then there was the JavaScript Programming Language (JS), which appeared in late 1995. This seemed like a relationship made in heaven, it is feature-rich, well-supported, and with a syntax akin to C Programming Language (C), C++ Programming Language (C++), and Java. It also did not suffer the fate of being of a plug-in nature but was built into all web browsers, which made for an exceptionally seamless user experience, by not having to download and install anything extra to get the functionality.

JS however has flaws in its software architecture and design too, such as not being able to handle 64-bit arithmetic, not having shared memory across its multitasking threads, having no unmanaged heap, and being unable to provide any synchronous features, such as raw Transmission Control Protocol/Internet Protocol (TCP/IP) networking sockets as outlined in Rossberg et al. (2018) and Fette and Melnikov (2011). JS is still actively used today and for all intents and purposes, even with its flaws, will remain part of all web browsers for many years to come.

There was also Macromedia Flash, appearing around 1996. It did seem much more promising than Adobe Shockwave, with its Three Dimensional (3D) capabilities and being adopted for use by movie industry heavyweights such as Disney and Fox Studios early on. At the end of the day though, Macromedia Flash was not quite suited for CIAs. In fact, once Macromedia was acquired by Adobe, Adobe Flash was shunned by Apple, because its software architecture and design made it too resource-intensive. Eventually, Apple succeeded in driving its web browser called Safari apart from Adobe Flash. Adobe Flash is still around these days, although official support for it ended at the end of 2020.

There were also other short-term relationships, such as those with Apple Quicktime, as well as Microsoft ActiveX and Silverlight. These technologies never led anywhere either as they were proprietary in nature and were not well-supported, if at all, beyond specific Operating Systems (OSs), like Microsoft Windows and Macintosh OS X. If one were using a Unix-based or other OS, one would have limited or no way at all of viewing the content.

A key challenge preventing web browsers from hosting and executing CIAs with native desktop performance is the lack of web browser technologies with the necessary software architecture and design capable of supporting them, as stated in Rossberg et al. (2018). For instance, if one tried to develop a web browser application that communicates with a server residing on the internet, across a TCP/IP networking port 7123, it would be impossible to implement.

Even when using the existing WebSocket technology as described in Fette and Melnikov (2011), it would be impossible because WebSocket only allows for using TCP/IP networking ports 80 and 443. It is this kind of limitation that is causing a barrier to all web browsers reaching their full potential. What all web browsers need to move beyond this barrier is an EE, unconstrained by historical restrictions, limitations and inadequacies, other than those which would make its use unsafe.

An EE that would easily work with a web browser on a desktop computer as well as a mobile device across multiple microprocessor chipsets like the Intel/AMD x86_64 ones as well as the ARM, or even the Sun UltraSPARC ones. An EE that can be easily compiled too, from various low-level and high-level programming languages, such as Assembler, C, C++, Java, JS or Python, to name a few. An EE that has similar or if possible, better performance in comparison to that of a native desktop as proposed by Rossberg et al. (2018) and Wen et al. (2020).

As background, all web browsers have over their history supported several technologies that have allowed them to host and execute applications. Some of these technologies were directly built into a web browser, thus not needing any extensions or plug-ins. Others came in the form of an extension or plug-in, such as Adobe Shockwave, Java Applets and Adobe Flash which unfortunately, due to the nature of extensions and plug-ins, always seemed to have to play catch-up once a new version of a web browser was released.

With the arrival of JS, which is both feature-rich and built into every web browser, it became possible to create applications that are feature-rich, but not necessarily compute-intensive. JS has since become the *de facto* technology with which to host and execute applications within any web browser, as stated by DiPierro (2018). One can attribute the wide acceptance of JS to the fact that there was no hindrance in deploying it due to its built-in nature, as stated in Rossberg et al. (2018), as well as its feature-richness, as compared to any other similar technologies available today.

Many attempts have been made to overcome JSs shortcomings as proposed by Vilk and Berger (2014), Eich (2015), Rossberg et al. (2018), Reiser and Bläser (2017) and Kataoka et al. (2018). Such as not providing an unmanaged heap, not being fully multi-threaded and only being asynchronous in nature, which in turn requires all JS applications to be event-driven by consequence. Google developed the Google Web Toolkit (GWT), which allows applications written in a limited form of Java to be hosted within a web browser.

Mozilla's research division developed asm.js, which allows many programming languages to be compiled into a limited form of JS as described by Herman et al. (2014). Mozilla's research division developed Emscripten, a Low-Level Virtual Machine (LLVM) based compiler, that compiles C, C++ and Rust programming languages into JS as described by Zakai (2011). Emscripten has shown the most promise of being able to create CIAs that can be hosted in any web browser. However, because Emscripten compiles to JS, it is still limited to the same shortcomings that JS has.

In early 2018, Rossberg et al. (2018) proposed WebAssembly Programming Language (WASM), which is also built into a web browser. From a performance point of view, it has been shown to be about twice as fast as JS. More advanced use cases have also been showcased by Ménétrey et al. (2021). Again, as with Emscripten, WASM uses the Virtual Machine (VM) as its underlying EE and therefore it too is not an optimal solution, because as with Emscripten, it is also limited to the same shortcomings that JS has.

1.2. Research problem

When the first web browsers were created, such as the National Center for Supercomputing Applications (NCSA) Mosaic and the Netscape Navigator, there was no or little need for them to be able to host and execute CIAs with native desktop performance. As the use of web browsers have increased over time and across devices such as mobile devices, so has the need for them to now be able to host and execute CIAs with native desktop performance.

The problem stemming from this need is rather simple, if a web browser cannot host and execute CIAs with native desktop performance, then users will always be forced into using their desktop native CIAs instead. This is not ideal, especially when users are becoming more mobile (Fernando et al., 2013; Kharb et al., 2021).

What users need is something with a radically different software architecture and design, something that is built from the ground up and that focuses first and foremost on performance. For instance, a VM based on a low-level instruction set like the Reduced Instruction Set Computer Version 5 (RISC-V) as described by Waterman and Asanović (2019). Implemented as a foundational layer, any programming language can then be implemented for it, such as Assembler, C or C++. Such an implementation could provide near native-like performance, given the similarities of it to certain instruction set architectures as found in today's Central Processing Units (CPUs).

Once this foundational layer has been created, one could add other layers on top of it, such as networking, graphics, storage and any other layers that a native desktop would have access to. Thereby fulfilling our mandate of developing an improved web browser EE by giving it the capabilities with which to host and execute CIAs with native desktop performance.

To summarise, web browsers cannot host and execute CIAs with native desktop performance, because the existing web browser technologies are based on software architecture and designs that make them restrictive, limiting and inadequate, which ultimately forces CIAs to only be created for, hosted and executed on native desktops.

1.3. Objectives and research question

1.3.1. Objectives

Given the research problem stated above, the goal of this study is to develop a new web browser EE, which can be used by any web browser to host and execute CIAs with native desktop performance.

The four key objectives correlating to the research question of this study are to:

Objective 1

Identify the features and limitations of existing web browser EEs that can host and execute applications.

Objective 2

Employ a suitable research methodology by determining which ones are best aligned to deliver on the requirements and goal of this study.

Objective 3

Develop a prototype web browser EE that is capable of hosting and executing CIAs with native desktop performance.

Objective 4

Evaluate and benchmark the performance of the prototype web browser EE against those discovered through the SLR, thereby testing the empirical hypothesis of this study.

1.3.2. Hypothesis and research question

The empirical hypothesis of this study is that:

Hypothesis

A web browser EE that can host and execute CIAs on any device with native desktop performance can be created.

Given the hypothesis, the singular research question that this study endeavours to answer and which can be outlined using the framework as proposed by Kofod-Petersen (2015), is:

Research Question

What software architecture and design does a new web browser EE need to comprise of in order to be able to host and execute CIAs with native desktop performance?

1.4. Theoretical background and related work

1.4.1. Theoretical background

The theoretical background for this study is based on the most recent advancements within web browser EEs. Wagner (2017) notes that “we’ve added new capabilities to the Web, like audio and video streaming, Two Dimensional (2D) and 3D graphics, typography, peer-to-peer communications, data storage, offline browsing, as well as multitouch, location, and camera inputs. But we continue to struggle with performance, specifically the ability to run applications hosted within a web browser as quickly as those on a native desktop”.

Based on that, Wagner (2017) details the performance improvements observed after using asm.js, which they created, which is a sort of optimised implementation of the JS language. The study then claims that with asm.js, one could create much faster JS applications than handwritten JS applications, almost as fast as the native desktop. However, no actual benchmarking, methods of data collection or analysis was provided with this publication in fact, no research design or methodology was provided, so it is impossible to verify the provided claims.

The study further states, that the improved performance when using asm.js is noticed only when providing specific patterns of code to the underlying EE, being the JS VM. Furthermore, asm.js still had the same limitations as JS, like the 64-bit arithmetic limitation. One can only conclude from this that even though some performance improvements have been observed by optimising the JS code that is executed within the JS VM, it hardly serves as a verifiable universal solution, especially when the underlying EE is the JS VM, which has known limitations.

Further to this, Wagner (2017) introduces WASM, which is proposed as the next evolutionary step to asm.js. Rossberg et al. (2018) provide details on WASM and elaborate further by stating that it will address various web application problems, such as those that make them unsafe and slow. Furthermore, previous attempts at solving this problem by using ActiveX, Native Client or asm.js have always fallen short. Web applications can then be developed using this new portable low-level programming language, which will give them the properties that a low-level compilation target should have.

The key point to focus on from this study is that WASM is a new programming language and not a new EE. In fact, by design, it strongly suggests that implementations of WASM use their existing JS VM as its underlying EE, together with specific WASM enhancements. One such enhancement is trying to provide shared memory across threads, this is because the JS VM was never designed to provide such a feature. So as with Wagner (2017), the approach taken was to make improvements to the language and not necessarily the EE.

Rossberg et al. (2018) provide some benchmarking where out of the 24 benchmarks that were run, four were shown to have equal or better performance than a native desktop. This benchmarking was done across a sampling range of 15 cycles and then averaged. One cannot help but ponder the idea that if WASM was implemented from the ground up and did not use the JS VM as its underlying EE, that a better result may have been attainable.

Reiser and Bläser (2017) went even further by proposing a way to cross-compile JS code into WASM stating that the desire for performance critical web applications will necessitate the need for faster and more predictable web browser EEs, even though the performance of today's JS VMs may be sufficient for current web applications. The study provided some benchmarking using two different web browsers as part of the analysis, namely Firefox and Chrome, which details how much improvement there is to the performance of JS code once converted to WASM. Again the approach taken here is also focused on the language improvements and not the EE.

Recently, Yan et al. (2021) performed a systematic study of WASM, with the aim of comprehending its performance benefits. Specific areas covered in that study were across its source code compilation, using Just-In-Time (JIT) compilation at runtime, analysing the performance of the EE and analysing its memory management, which is significantly higher in comparison to JS. Given the nature of the chosen benchmarks, the results of the study were mixed, which may have contributed to certain generalised conclusions.

The shortcomings or gaps in previous studies, which quickly comes to light, are that they focused solely on improvements and optimisations to the JS language and have done nothing or very little to improve the underlying EE, being the JS VM. Given that JS code is executed within the JS VM, it will continue to be restricted by the current limitations of the JS VM.

Similarly, whilst WASM is proposed as a new programming language, it does little to solve the existing JS VM restrictions, limitations and inadequacies either. For instance, where WASM have tried to improve threading, by introducing a form of shared memory between threads, it comes across more like a makeshift solution, instead of an optimal one, simply because the proposed solution had to take the existing JS VM architecture and design constraints into account.

1.4.2. Related work

Other closely related literature for this study is taken from various sources and subject areas such as, but not limited to, Rich Internet Applications (RIAs) and Single Page Applications (SPAs), as well as more performance-based research that covers additional areas of JS and WASM. The first literature that one can find that relates to the exploration of having desktop equivalent applications on the web is an article by Taft (2003), in which a term is coined by an employee of Macromedia, by the name of David Mendels, that term was RIA.

Since then, various researchers have explored the subject of RIAs, like Fraternali et al. (2010), where they capture the meaning of an RIA quite eloquently by stating that RIAs can have an interface that is based on a single page design, together with subpages. The user's interactions are then managed through these subpages, similarly to how a native desktop application would work. This architecture and design negates the need to refresh the full page with each user interaction and using JS, individual page elements of the application can then be loaded, displayed and updated, independently as required.

Similarly, Casteleyn et al. (2014) elaborate further by stating that the same features and functionality as that of a native desktop application can generally be found in an RIA. The study by Casteleyn et al. (2014) also provided a good mapping of how the development of the technologies that produce RIAs have progressed over the preceding decade.

Various other literature, all of which are focused on improving the performance of web applications were consulted, such as the work of Bourgoin and Chailloux (2015) that provided innovative ideas of using WebCL as a means to improve web application performance. In another study by Ho et al. (2017), the authors focus on making improvements to the WebGL technologies as a means to improve web application performance and then there is the study by Park et al. (2018) where the idea that was derived was to re-use the optimised code for JS once it was pre-compiled by its Ahead-of-Time (AOT) compiler.

Malle et al. (2018) and Jangda et al. (2019) provide insights into the performance of applications executing on a native desktop versus those executing within a web browser. In some cases, applications were more than twice as fast when executed on a native desktop compared to a web browser. The study by Jangda et al. (2019) is especially critical of the performance of WASM or the lack thereof contrary to the study by Fras and Nowak (2019) who derived different conclusions surrounding the WASM performance while reviewing it when used to create SPAs.

Recently, there was also Zakai (2011) and Zakai (2018) who introduced Emscripten, which is a compiler capable of converting C code into JS code. Again approaching the problem of web application performance from the JS language point of view. A study by Arteaga et al. (2020) proposes an innovative optimisation for the WASM compiled code. Taivalsaari and Mikkonen (2017) round everything off with a comparative study of the state of web applications for the preceding decade, which provide a good understanding of where the focus of the research on this topic has been.

Thus from previous research work conducted within this subject area, one can see that the focus has almost solely been on improving and optimising the JS language. The study proposed through this research project is to create an innovative software architecture and design, which will then be used to develop a new web browser EE that is completely new and not dependent on any existing web browser technologies, including the JS VM. By creating an optimal software architecture and design, and then developing a new web browser EE based on it, it is theorised that one can achieve the goal of having a web browser EE host and execute CIAs with native desktop performance.

1.5. Design, methodology and ethics

1.5.1. Design

This study uses automata theory (Hopcroft et al., 2006), or as it is also known, the theory of computation (Sipser, 2012) as its foundation, underpinned by programming language theory (Pierce, 2002; Harper, 2013) and virtualisation theory (Gaj et al., 2015; Randal, 2020). This study is based on quantitative research using Design Science Research (DSR) as defined by Peffers et al. (2007), Peffers et al. (2012), Wieringa (2014) and Myers and Venable (2014) within information systems (Hevner et al., 2004; Hevner & Chatterjee, 2010).

The reasons for choosing DSR are best explained by using Hevner et al.'s (2004) seven proposed guidelines for DSR. These guidelines provide a framework for conducting and evaluating DSR projects in the field of information systems and related disciplines that emphasises the creation of practical solutions while adhering to rigorous research principles. These guidelines also help researchers design and develop innovative artefacts that address real-world problems. These guidelines will show how DSR is firmly embeded into this study, as follows:

Guideline 1: **Design as an artefact** - Where this study seeks to explore the development of a new web browser EE, as defined by the second objective of this study.

Guideline 2: **Problem Relevance** - Where the problem domain of this study is web browser EEs, as outlined in the research problem of this study.

Guideline 3: **Design Evaluation** - Where this study evaluates the new web browser against existing ones, as defined by the third objective of this study.

Guideline 4: **Research Contributions** - Where this study seeks to provide software architecture and design contributions, as detailed in the methodological contributions of this study.

Guideline 5: **Research Rigour** - Where this study seeks to rigorously define the requirements of the new web browser EE, as outlined in the delineation of this study.

Guideline 6: **Design as a Search Process** - Where this study seeks to innovatively develop this new web browser EE based on current trends and previously unexplored use cases of existing technologies, as described in the theoretical background, delineation and methodological contributions of this study.

Guideline 7: **Communication of Research** - Where this study will seek to publish the findings hereof in the most relevant manner, through the publication of the resulting thesis as well as via journal publications and/or conferences.

Another research type evaluated was that of Experimental Research (Babbie, 2016), but it was found to not completely align with the intent of this study as that type of research focuses on the experiments and the design of the experiment plan only and does not lend itself to the creation of an artefact, such as a new web browser EE, which is the primary intent of this study. Experimental Research is also primarily concerned with exploring cause-and-effect relationships through controlled experiments, while DSR focuses on creation and evaluation of practical artefacts to address specific problems in fields such as information systems and engineering.

Action research (Iivari & Venable, 2009) was also evaluated but found to not be suitable for use by this kind of study. The reason for this is that while both Action Research and DSR involve practical problem-solving, Action Research is characterised by its collaborative and participatory nature usually with practitioners or stakeholders, with a primary focus on improving processes, practices and generating recommendations, whereas DSR emphasises the creation and evaluation of innovative artefacts to address specific problems, often in fields related to technology and design.

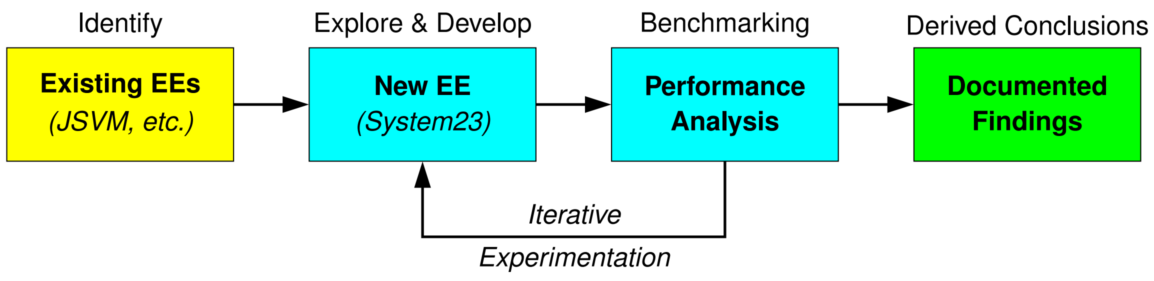

Figure 1.1: Conceptual Framework

The conceptual framework for this study (Kivunja, 2018) is depicted in figure 1.1, which aligns with this study's four objectives, the key parts of which are to identify, explore, develop, benchmark, and then derive conclusions. It is important to note that the iterative loop relating to the explore, develop and benchmarking parts aligns well with the expected "Design-Build-Test-Test Loop" cycle of DSR. Where in the context of DSR, the "Design-Build-Test-Test Loop" cycle is a structured iterative process that researchers follow to create, evaluate, and refine the artefacts that they develop.

This cycle is repeated as needed, with each iteration leading to enhancements and improvements in the artefact. Throughout the DSR process, researchers document their decisions, actions, and the rationale behind their design choices to ensure transparency and the potential for knowledge dissemination. This process is central to DSR and allows for the systematic development and improvement of innovative solutions. The goal of DSR is thus to ensure that the final artefact is not only innovative but also practical and effective in solving the identified problem.

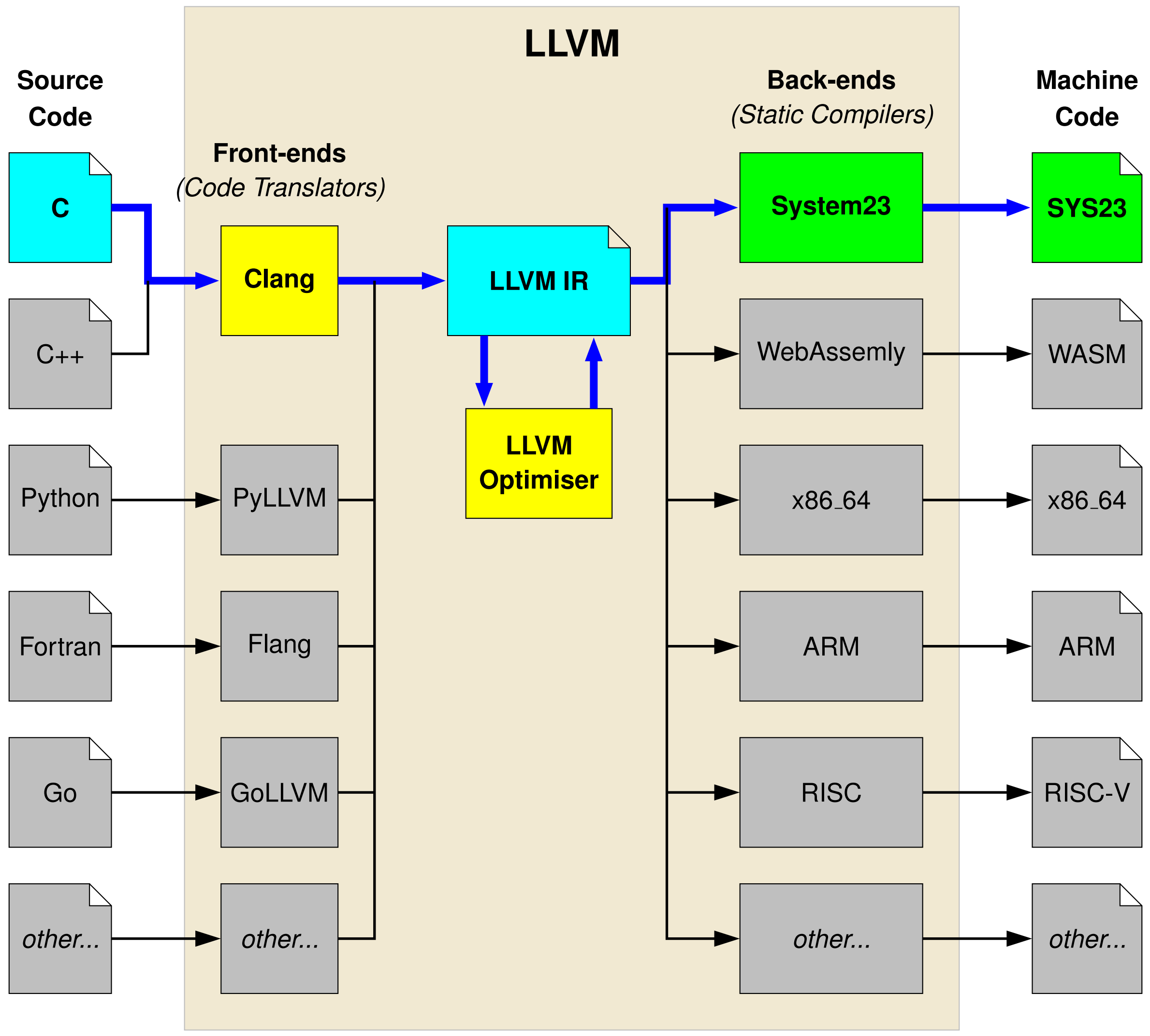

Figure 1.2: Conceptual System23 (SYS23) Compiler Toolchain

For the sake of convenience, the new web browser EE that is to be explored and developed shall provisionally be named System23 (SYS23). Figure 1.2. depicts the conceptual view of the SYS23 compiler and toolchain, which will primarily look to leverage and enhance the existing LLVM compiler and toolchain technologies for its needs. The SYS23 compiler is a two stage static compiler and can be defined as follows:

Stage 1: Translates Low-Level Virtual Machine Intermediate Representation (LLVMIR) code into SYS23 assembler code.

Stage 2: Compiles SYS23 assembler code into SYS23 machine code.

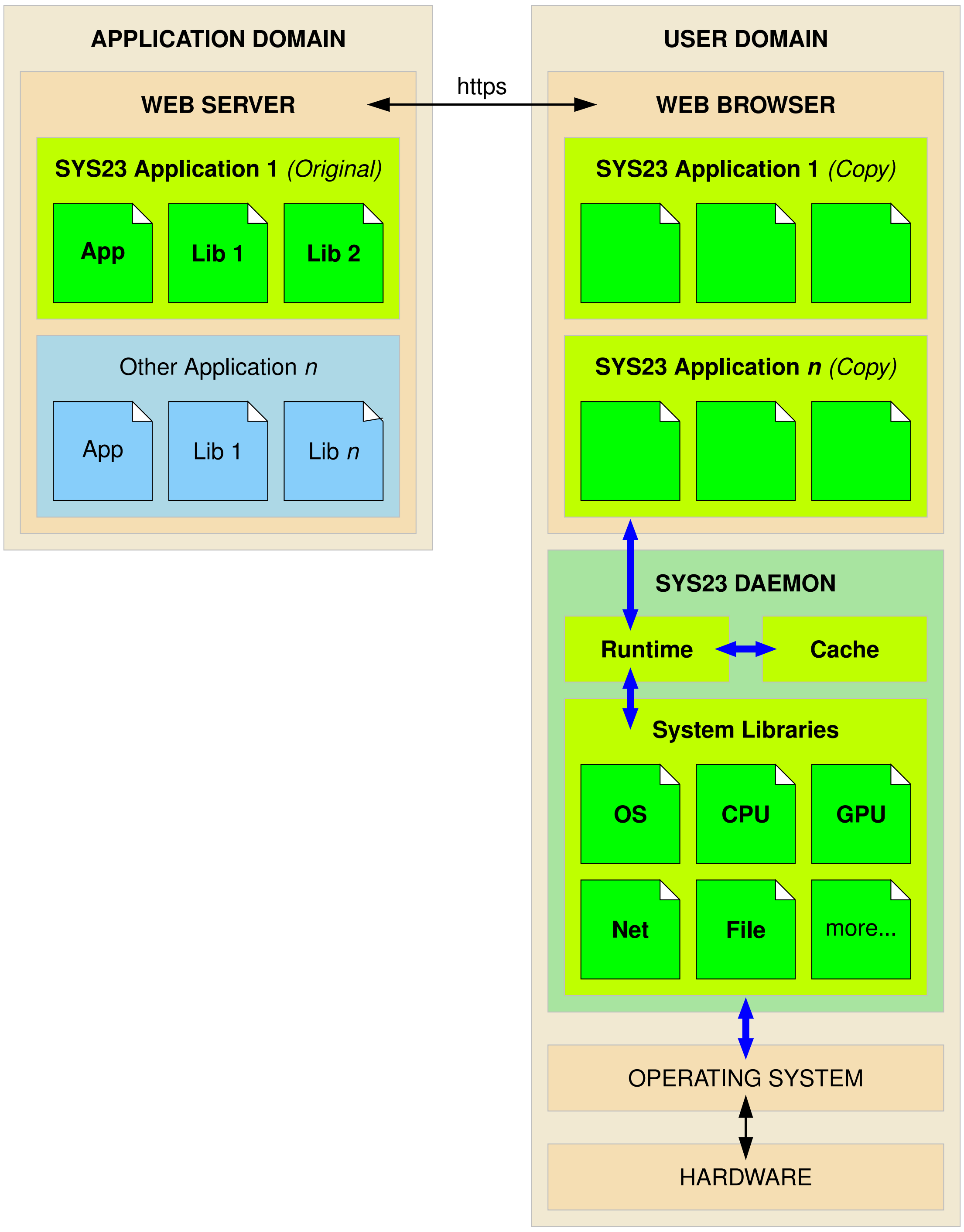

Figure 1.3: Conceptual SYS23 Web Application Retrieval

Using the C4 architectural model for visualising software architecture (Brown, 2022), figure 1.3. depicts the C4 software context (level 1) conceptual view of a potential SYS23 application being downloaded or retrieved from a cache, if already previously downloaded. Figure 1.4. depicts it's potential use cases:

1. As a client (application 1), when part of a client/server application configuration. Where

the backend application server can be either SYS23-based or one based on other existing technologies, therefore negating the need to rewrite them.

2. As a standalone application (application 2).

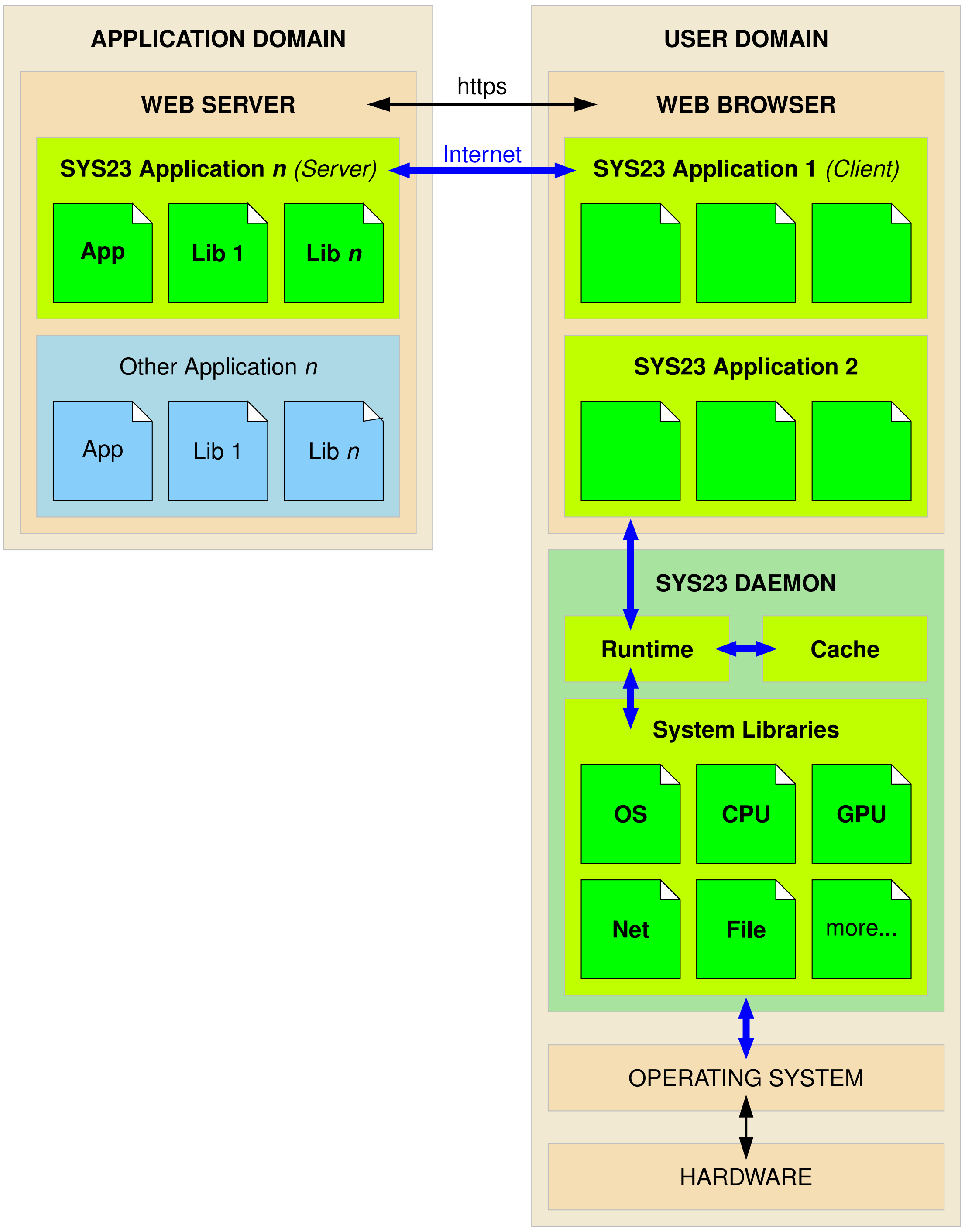

Figure 1.4: Conceptual SYS23 Web Application Use Cases

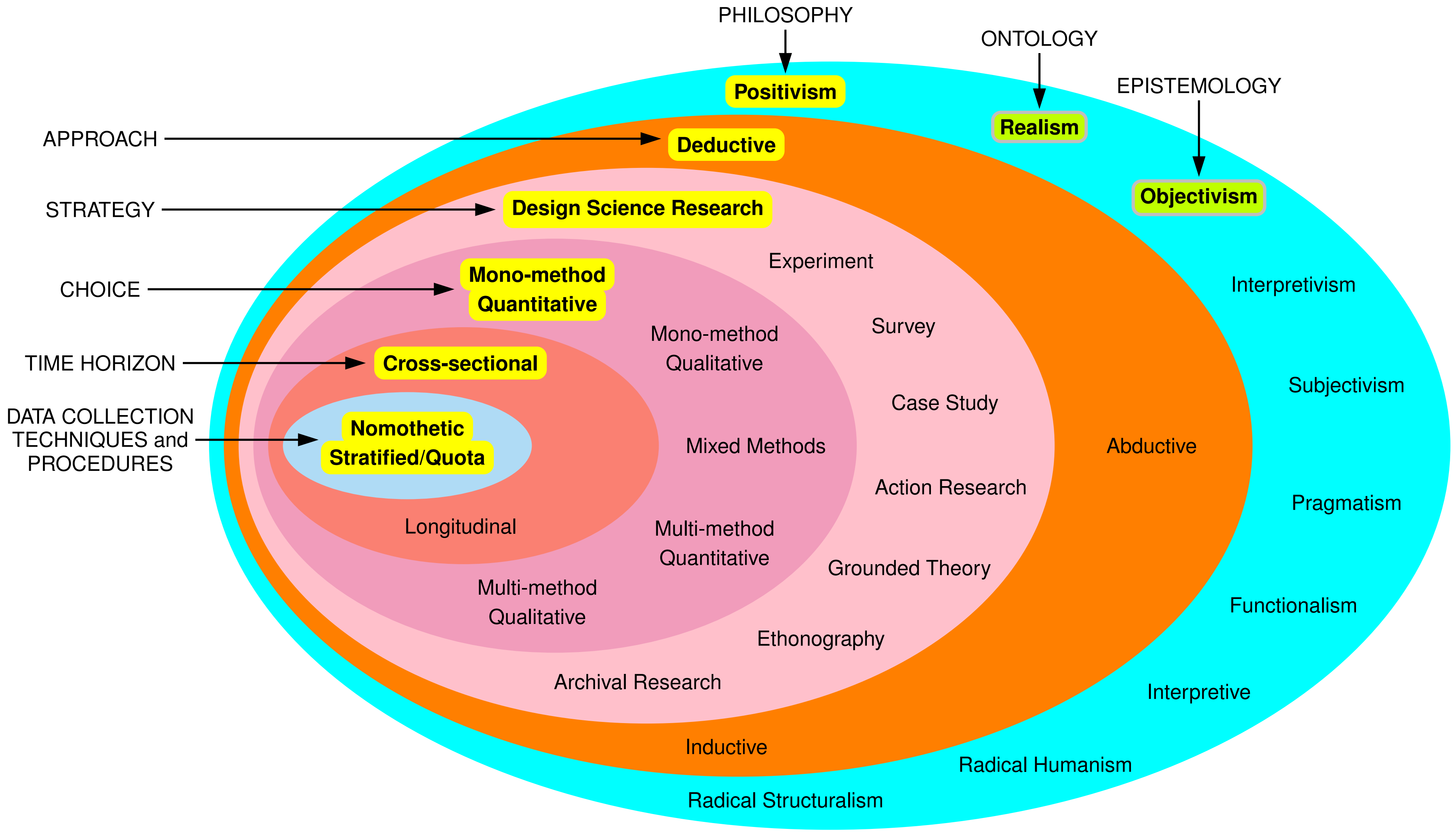

Figure 1.5: Expanded Research Onion taken from Saunders et al. (2019)

1.5.2. Methodology

The goal of this study is solution-orientated, to produce a web browser EE and the context will cover all web browsers. Figure 1.5, outlines the complete research methodology for this study, as based on Saunders et al. (2019) Research Onion, together with additional portions relating to the research and study types as defined within the research design.

1.5.2.1. Research philosophy

Given the nature of this study, the philosophical position is that of positivism. The reason for this is that the hypothesis of this study is firmly grounded in positivism, seeking to reveal a truth by using quantitative data in the form of performance-based statistical data derived through experimentation. The ontological position or how the researcher views this world is through realism as a positivist, i.e. viewing the world as being separate from humans and their interpretation thereof. The epistemological position or how the researcher should investigate the world is through objectivism as a positivist, i.e. viewing observable evidence as the only form of defensible scientific findings.

The reasoning for selecting positivism is that when working with computer systems, there is only one singular reality that exists because when a specific input is given to a computer system, it will always produce a specific output and this output will never change, not unless the input changes. The singular reality for this study is the observable evidence generated from the benchmarking of the new web browser EE. That observable evidence is the only form of defensible scientific findings that will indicate if the new web browser EE is equal to or better in performance to existing web browser EEs or not.

1.5.2.2. Research approach

The approach adopted by this study is deductive through iterative development based on multiple cycles of exploration, design, development and evaluation, which aligns well with the intent to iteratively develop and measure as defined by DSR. An etic perspective guides this study so that the researcher can objectively interpret the findings of our benchmarking and provide a sound value judgement.

1.5.2.3. Research strategy

The strategy employed is that of design science through iterative experimentation, where the evaluated results can be benchmarked against existing web browser EEs. For this study, the researcher will be using the JS VM as the baseline web browser EE. It serves as both our baseline web browser EE and also as the key constraint to overcome.

1.5.2.4. Methodological choice

Given the iterative experimentation strategy of this study, a mono-method quantitative choice (Saunders et al., 2019; Minichiello et al., 1990) based on experiments alone, was used to gather numerical data derived from benchmarking the performance of the web browser EE. Alternatively, employing a multi-method quantitative choice that incorporates methods such as surveys or observations did not align with the iterative experimentation strategy and the nature of this study and was therefore avoided.

Figure 1.6 shows the data collection choices (Saunders et al., 2019) that are available and were considered for this study.

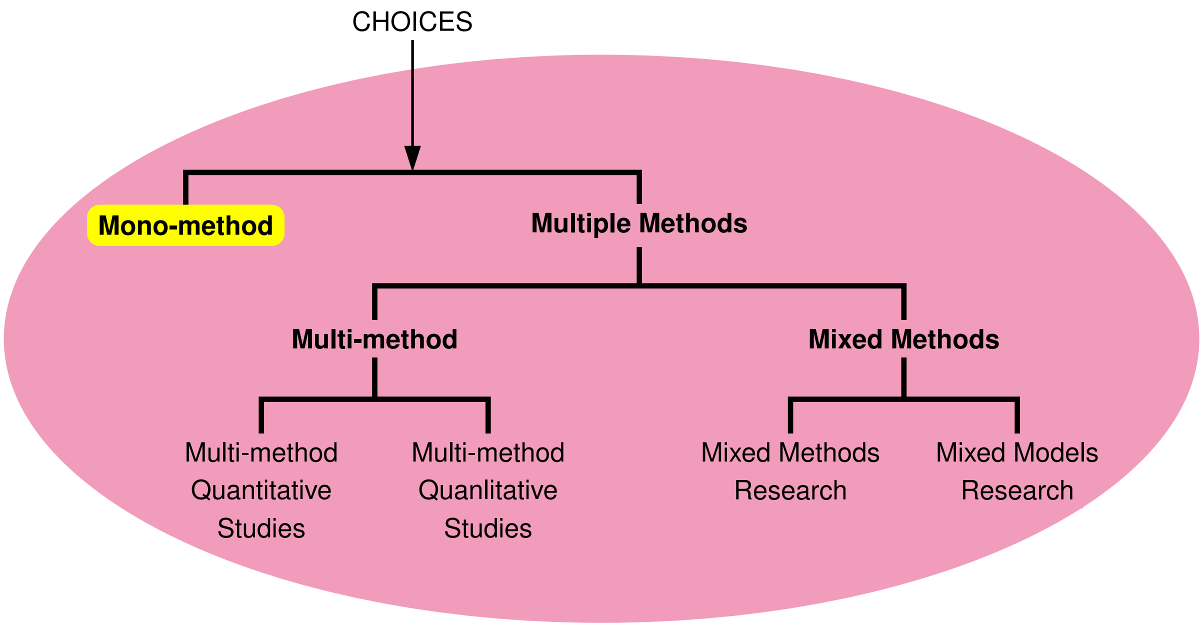

Figure 1.6: Data Collection Choices (Saunders et al., 2019)

1.5.2.5. Time horizons

The research was conducted as a cross-sectional study, as described by Bryman (2016). Given the intent to employ an iterative approach to the study, using a cross-sectional study would allow one to iteratively track the progress of the study through the benchmarking of the new web browser EE.

1.5.2.6. Data collection techniques and procedures

Together with the mono-method quantitative choice, this study used a nomothetic data collection technique and the sampling of the collected benchmarking data was done using the stratified and/or quota probability sampling methods (Baltes & Ralph, 2022). Performance analysis tools, such as JetStream[1] and/or PolyBench/C[2] were then used to analyse the sampled benchmarking data.

These performance analysis tools were also used by Salim et al. (2020) and Rossberg et al. (2018) to benchmark and analyse the performance data of WASM. Furthermore, the existing web browser EEs were used to establish a performance baseline against which the performance benchmarks of the new web browser EE were compared. Using statistical inference and numerical comparisons, the researcher was then able to build a view of the speed of the new web browser EE.

1.5.3. Ethics

Ethical considerations for this study were based around DSR and are best defined by the six ethical principles, as described by Myers and Venable (2014). The ethical considerations for this study based on these six principles are as follows.

1.5.3.1. Principle One - The public interest

The new web browser EE developed during this study will be available for use by anyone that uses a web browser. As the user will already have access to these desktop equivalent applications, there is no anticipated increase of any existing risk factors. In fact, there may be a decrease in risk factors due to web browsers being able to host applications within a sandbox and thereby preventing malware from propagating beyond the sandbox and infecting their native desktop.

1.5.3.2. Principle Two - Informed consent

Informed consent would be sought formally from all human subjects involved in this study. However, there was no involvement from any human subject, as all experimentation was based on computer system laboratory work only. Thus, data privacy or data protection as provided for by the Protection of Personal Information (POPI) Act (South Africa, 2013) did not pose any relevance to this study.

1.5.3.3. Principle Three - Privacy

Privacy and confidentiality risks and concerns of individuals and organisations are not anticipated to increase as the EE executed applications will still execute on the user's device

[1]https://browserbench.org/JetStream/
[2]https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1

within a web browser, with no increased risk of data leakage. Safeguards were sought to include protecting sensitive data and ensuring that data is stored and handled securely.

1.5.3.4. Principle Four - Honesty and accuracy

The researcher of the study acknowledged all referenced information in the prescribed Harvard style and committed to make available all research findings for academic and general public scrutiny. Bias across the research findings has been avoided by utilising known and well-documented methods of measurement. This allows anyone to scrutinise the research for plagiarism and the new web browser EE for accuracy.

1.5.3.5. Principle Five - Property

This talks about the ownership of the intellectual property rights of the research findings. There was no intention to apply for any bursary, scholarship, or any other funding from any organisation whatsoever and the fact that the research was undertaken by the researcher hereto, unaccompanied by any other researchers or research assistants, which implies that the researcher hereto owns the intellectual property rights of the research and findings. The publishing of the research and findings is thus at the sole discretion of the researcher hereto under the supervision and guidance of the Cape Peninsula University Of Technology (CPUT).

1.5.3.6. Principle Six - Quality of the artefact

Given the intended aim of the study is to produce a new web browser EE that is comparable to similar existing web browser EEs, it was deemed sufficient to evaluate the new web browser EE against existing testing apparatus to measure its accuracy to the produced specifications, thus ensuring it is of a high quality and has a low risk of failure. It was however not the intent of this study to build a web browser EE that is fully functional and production ready, but merely a prototype that demonstrates the results of this study.

As a final note, it is not the intention of this study to advance any military capability or aid in the loss of life through the use of the new web browser EE. Unfortunately, it will be impossible to know how the new web browser EE is being used, once this study has been published, which may very well lead to it being used for military purposes or to aid in the loss of life.

1.6. Delineation

Focusing on the exploration and development objective, figures 1.2-1.4 outline the conceptual view of the delineation of this study, which includes two parts:

1. A compiler and toolchain.

2. A web browser EE.

Using the Firefox web browser as a reference implementation, figure 1.7 details the exact component within the web browser that this study focussed on, being the EE. The exploration of the other components within the web browser, such as the user interface, browser and rendering engines, networking, persistent storage, display backend and Extensible Markup Language (XML) parser, was not conducted.

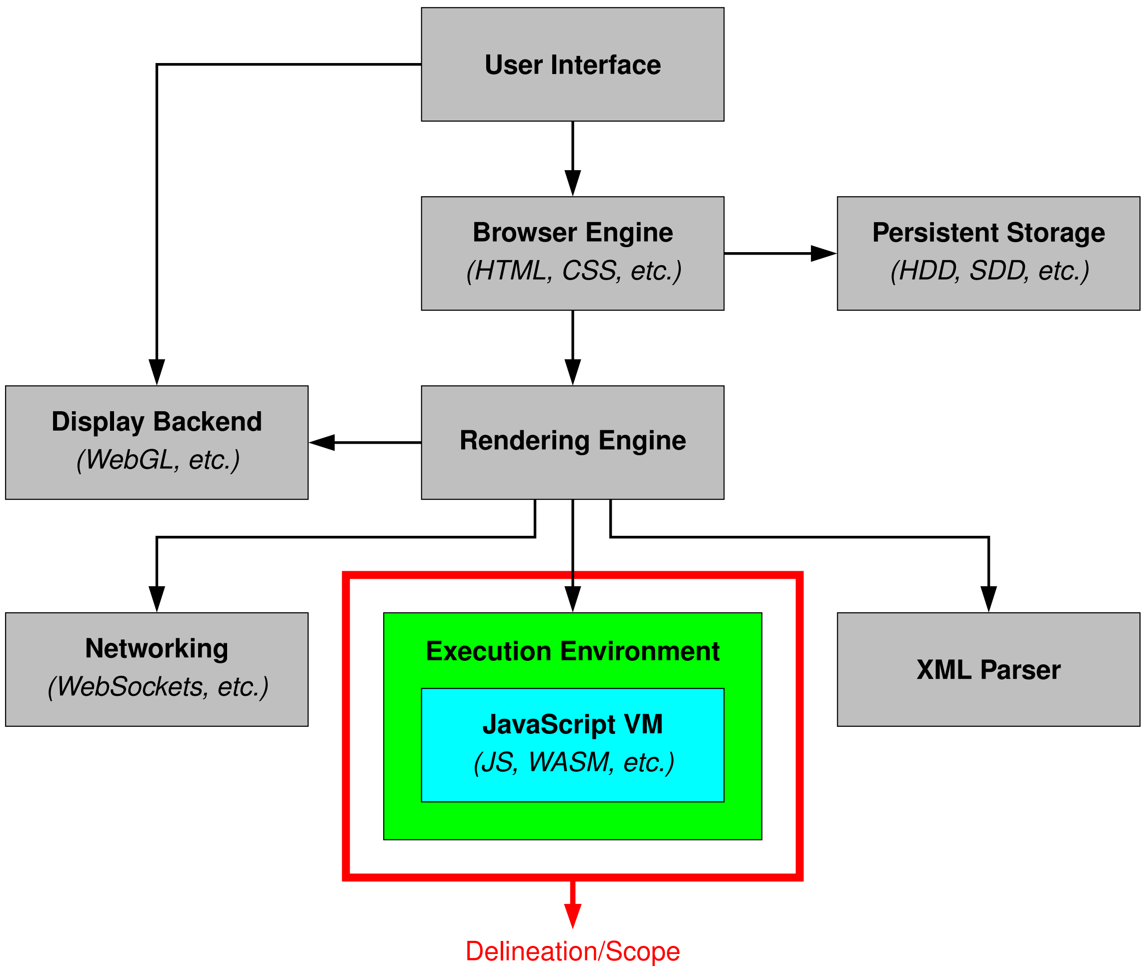

Figure 1.7: Firefox Web Browser Architecture

This new web browser EE has a software architecture and design not based on current web browser technologies, such as JS or WASM, to name a few. Its software architecture and design is from the ground up and not limited by any existing web browser technologies, except for those without which its use would be unsafe. It is not the intent that a whole new web browser be developed, but merely an EE, which can form part of any existing web browser.

For this study, the intention was to use the Firefox web browser to test and benchmark the new web browser EE. The Firefox web browser is open-source and thus allows one unrestricted access to its source code, which makes it easy to incorporate the new web browser EE into it. This is similar to the approach taken by the Tor group by using Firefox as the core to its web browser and then adding the Tor components as described by Dingledine et al. (2004) to it, to create the Tor web browser.

1.7. Outcomes, contribution and significance

1.7.1. Outcomes

As scoped in section 1.6., the outcome of this study is the development of a new web browser EE. Together with this, this study adds to the existing knowledge, such as Millhouse (2018), in the context of web browsers and EEs through the publication of this study and its findings.

1.7.2. Contribution

The contributions of this study can be examined from three perspectives, namely theoretical, methodological and practical.

1.7.2.1. Theoretical contribution

This study contributes theoretically, by offering insights into improving the performance of web applications by way of exploring different software architectures and designs for web browser EEs. Researchers could then in future further enhance the performance of web applications as it relates to web browser EEs, based on the software architecture and designs developed through this study.

1.7.2.2. Methodological contribution

This study contributes methodologically, by offering methods and approaches that have been developed, adapted, or improved, outlining software architectures and designs for web browser EEs with superior performance as compared to current options. These approaches may explore areas such as utilising alternative virtualisation methods for hosting the EE, such as OS level virtualisation, in contrast to the current JS VM that relies on process-level virtualisation.

Additionally, it involves possibly adapting an Instruction Set Architecture (ISA), typically used for hardware implementations, to be applied in the new virtualisation method. New methods and approaches for performance benchmarking of web browser EEs may also be developed, adapted, or improved if existing ones are found to be inadequate or unavailable for the required context.

1.7.2.3. Practical contribution

This study provides practical contributions by establishing a foundation on which a production-ready web browser EE can be built, by utilising the prototype EE developed through this study.

1.7.3. Significance

Numerous advantages present themselves if one were to provide web browsers with the ability to host and execute CIAs, with native desktop performance.

These may include:

1. Increased portability, whereby applications would not be required to be recompiled or rebuilt, so as to be able to be executed by different and disparate OSs.
2. Increased mobility, whereby users could continue using the same application when moving between a mobile device and a desktop computer.
3. Allowing enterprises to increase their staff's ability to embrace *Bring-Your-Own-Devices*, which avoids the need for applications to be installed onto a native desktop, thereby reducing the complexity that comes with managing applications accessed from those devices.
4. Reducing the support required when installations fail or become unstable due to an OS update, by avoiding the need to install applications onto a native desktop.
5. Reducing the direct and unfettered access of applications to the OS, which in turn limits the attack surface of malicious applications, since the malware that they may contain will be executed within a sandboxed environment.
6. A large number of organisations and individuals will benefit from this study, as web browsers are widely used on most mobile devices and desktop computers. Essentially, anyone who uses a web browser could potentially benefit from its findings.

1.8. Thesis Structure

This thesis comprises of six chapters and two appendices, set out as follows:

Chapter 1: ***Preface***

Provides an introduction and overview of the proposed study.

Chapter 2: ***Background***

Examines the background literature through a rigorous SLR to this research and discusses the research objective 1, using the review.

Chapter 3: ***Methodology***

Provides the philosophical stance and research methodology and addresses research objective 2.

Chapter 4: ***System Architecture and Design***

Proposes the new web browser EE and addresses research objective 3.

Chapter 5: ***Findings and Discussion***

Reviews the results obtained from performance testing the new web browser EEs and addresses research objective 4, while also covering objectives 1 through 3.

Chapter 6: ***Conclusion***

Articulates the researchers conclusions for this study, together with recommendations for the future directions of inquiry.

Appendix A: ***Benchmarking Suites***

Details all of the benchmarking suites discovered through the SLR.

Appendix B: ***Source Code***

The source code of both the SYS23 prototype, as well as the source code added to the benchmarking suite.

Appendix C: ***Benchmarking Runbook***

Details the commands that were used to complete the benchmarking.

CHAPTER 2

BACKGROUND

This chapter analyses the existing literature based on the topic of web browser EEs within the context of performance and also other areas that may relate to or influence aspects of performance, such as usability and User Experience (UX). This chapter also covers the first objective of this study, where the study endeavours to:

Objective 1

Identify the features and limitations of existing web browser EEs that can host and execute applications.

This chapter comprises of the following sections, structured as a chronological record:

Section 2.1. *History of the Web*

Section 2.2. *The birth of JavaScript*

Section 2.3. *The emergence of WebAssembly*

Section 2.4. *A structured literature review*

Section 2.5. *The theoretical grounding*

2.1. History of the Web

British scientist, Tim Berners-Lee, invented the World Wide Web (WWW) in 1989 while working at CERN[1]. He developed the two proposals for the WWW in March 1989 and May 1990 (Berners-Lee, 1990), then together with Robert Cailliau, a Belgian systems engineer, it was formalised for internal use at CERN in November 1990.

The main concepts are outlined in chapter 1, which describes important terms, such as a *web* of *hypertext* documents, viewable through a *browser*, globally distributed as the

[1]CERN is derived from the acronym for the French ”**Conseil Européen pour la Recherche Nucléaire**”, or European Council for Nuclear Research, a provisional body founded in 1952 with the mandate of establishing a world-class fundamental physics research organisation in Europe.

WorldWideWeb. Initially, the purpose of the Web was to simply share static documents globally, across as many different devices as possible. The world's first website had an address of info.cern.ch and the world's first Web page[2] was hosted there. CERN, preserving its history, has continued running their Web server and one can still access both the website and the Web page today.

After Tim Berners-Lee publicly announced the WWW in 1991, interest in it quickly spread across the globe, with Web servers coming online in the US by Christmas of 1991. The first web browser called Mosaic, was developed by the NCSA at the University of Illinois and released in 1993, followed closely by the Netscape Navigator, released in 1994. For the first time, users of personal and Apple computers had access to a web browser, which further increased the adoption of the WWW.

By 1994, the Web started peaking the interest of the commercial sector at which time the Web comprised of over 10,000 servers, 5% or 2,000 of which were owned by non-academic and non-research entities. At that stage, it became clear that the Web had many more use cases and was no longer seen as a tool for exclusive use by the academic and research world. As with the release of the Mosaic web browser, the interest generated by the commercial sector into the Web further increased the adoption of the WWW.

Even though the original idea behind the WWW was to create a global information system using computers, data networks and hypertext files to simply display pages of static information, by 1995 websites were pushing the limits and technical boundaries of what could be achieved on the Web. So the WWW and the global community using and supporting it had a need to evolve it into something more. In 1995 a company called Netscape started looking into moving beyond having websites with only static information.

What they came up with was a technology capable of executing applications within a web browser, with the aid of an embedded VM. This technology was the JS EE, which today is the standardised VM used by all web browsers for this purpose.

2.2. The birth of JavaScript

JS was invented by Brendan Eich in 1995, while he worked at Netscape. There is often some confusion between JS and Java, where the assumption is that they are the same thing, but the two have almost nothing in common, other than them having a similar source code syntax. Also, JS source code is never compiled into a binary format, but rather interpreted at run time by its VM. Whereas Java source code is compiled into a binary format called byte code, this byte code is then interpreted by its VM at run time.

[2]http://info.cern.ch/hypertext/WWW/TheProject.html

JS was originally called Mocha, but soon after became known as LiveScript, later on it was again renamed to JS. JS was initially developed for the Netscape 2 web browser, where early adopters of it were Microsoft's Internet Explorer 2 web browser, which started supporting it in late 1995. Later on in 1997, due to its rapid growth and global use, Netscape handed it over to the standards organisation called ECMA[3]. At that stage, JS was adopted as the ECMA-262 standard and also became known as ECMAScript.

A typical architecture for a JS EE is shown in figure 2.1. This includes all of the necessary components that allow it to take JS source code and translate it directly or indirectly via an interpreter, into machine code that a CPU can understand. This JS EE architecture is typical of how it is embedded into the Google Chrome and Microsoft Edge web browsers through what is known as the V8 JS EE, or how it is embedded into the Mozilla Firefox web browsers through the SpiderMonkey JS EE.

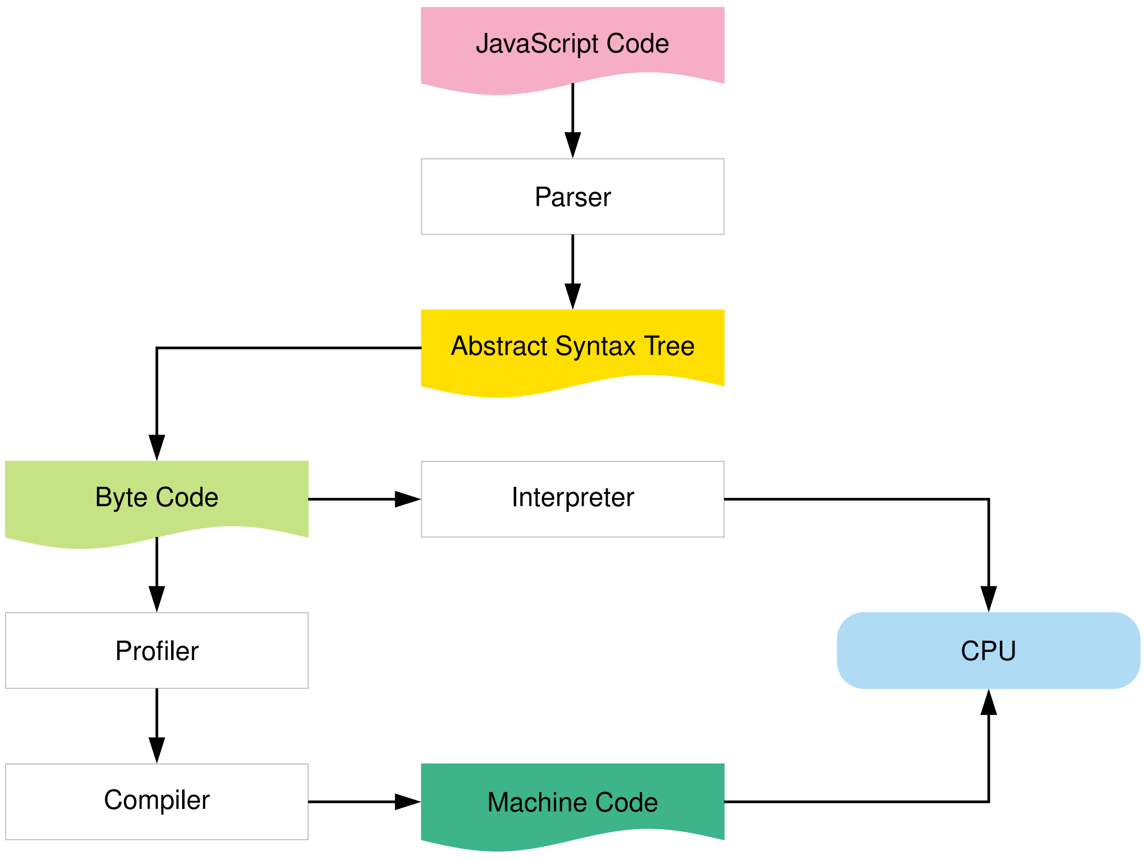

Figure 2.1: A Typical JS EE Architecture

Fast forward one decade and 2005 proved to be a revolutionary year for JS, due to the release of Ajax (Garrett, 2007), a suite of technologies that included JS. Web pages felt more like native desktop applications when they used Ajax, because it allowed web pages to be more interactive

[3]Ecma International is a non-profit standards organisation for information and communication systems. The current name, acquired in 1994, was due to the European Computer Manufacturers Association changing its name to reflect the organisation's global reach and activities

and responsive, thereby improving the user experience (Kluge et al., 2007). Figure 2.2 depicts the improvements introduced by Ajax, showing that they are specific to the communication between the Web server and web browser, in no way does Ajax alter the underlying JS EE.

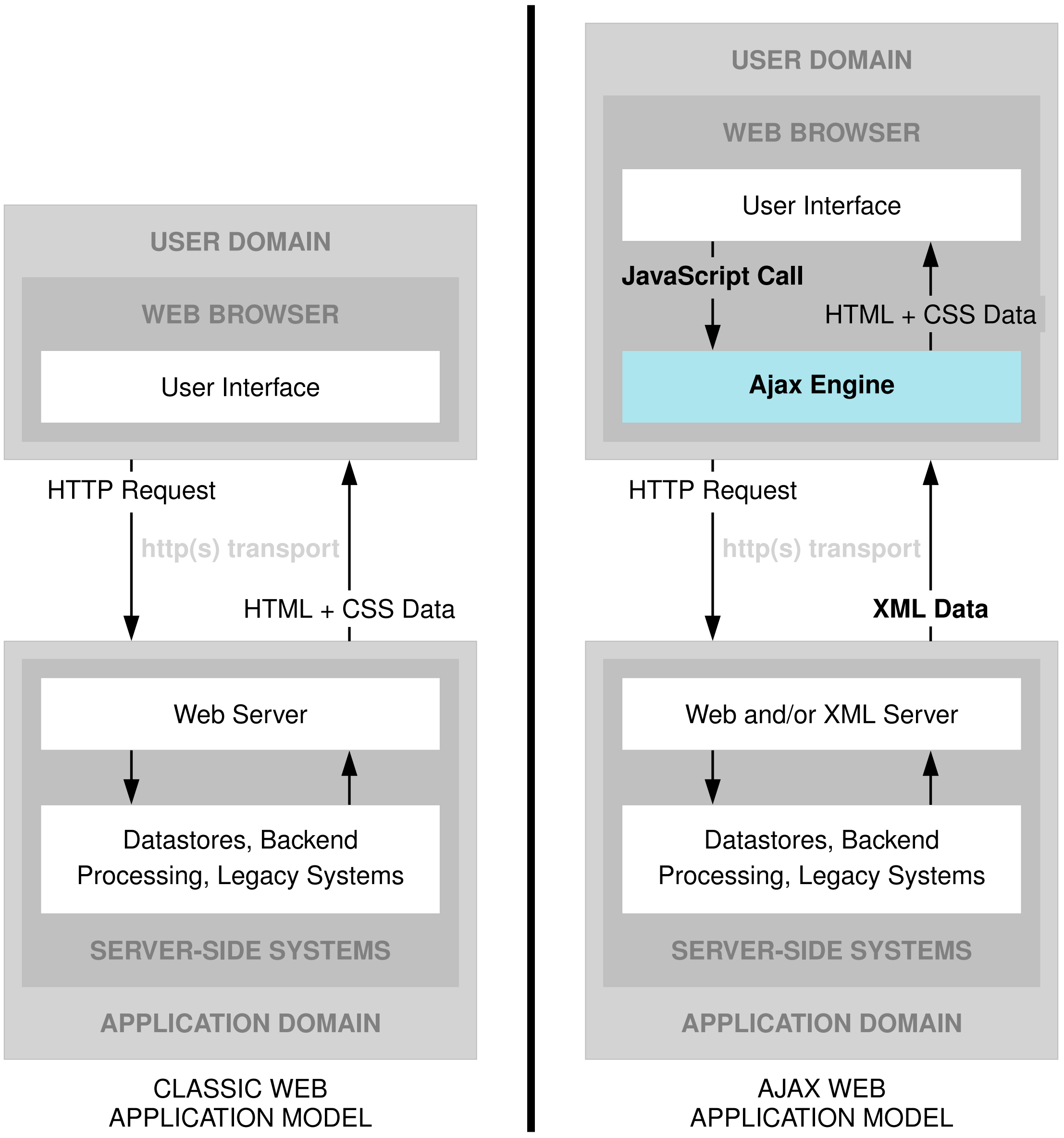

Figure 2.2: Ajax Web Application Model taken from Garrett (2007)

Following that, work undertaken by the CommonJS project in 2009, created the foundational components required for JS to be used outside of a web browser. This gave rise to technologies such as Node.js, an environment capable of running JS without needing a web browser. Even Java adopted the idea of allowing JS code to be embedded within Java code. This fundamental advancement allowed JS to now also be utilised for non-Web related needs and continued JS adoption even further.

Over the following years, JS went through many refinements and updates, which culminated in the 2015 release of ECMAScript 6 or JS 6, as it is more commonly known. By this point in time though, many JS flaws and shortcomings were well known and the industry again started looking for ways to move beyond them. One such innovative technology that attempts to fix some of the JS flaws and shortcomings was proposed in 2018, named WASM.

2.3. The emergence of WebAssembly

WASM was created by the World Wide Web Consortium (W3C) and made its first public appearance in early 2015 (Eich, 2015). Its main design goals at the time were to be efficient, fast, safe and portable. The intent behind these design goals was to try and fix some or all of the shortcomings of JS. To some extent, WASM has indeed lived up to this expectation.

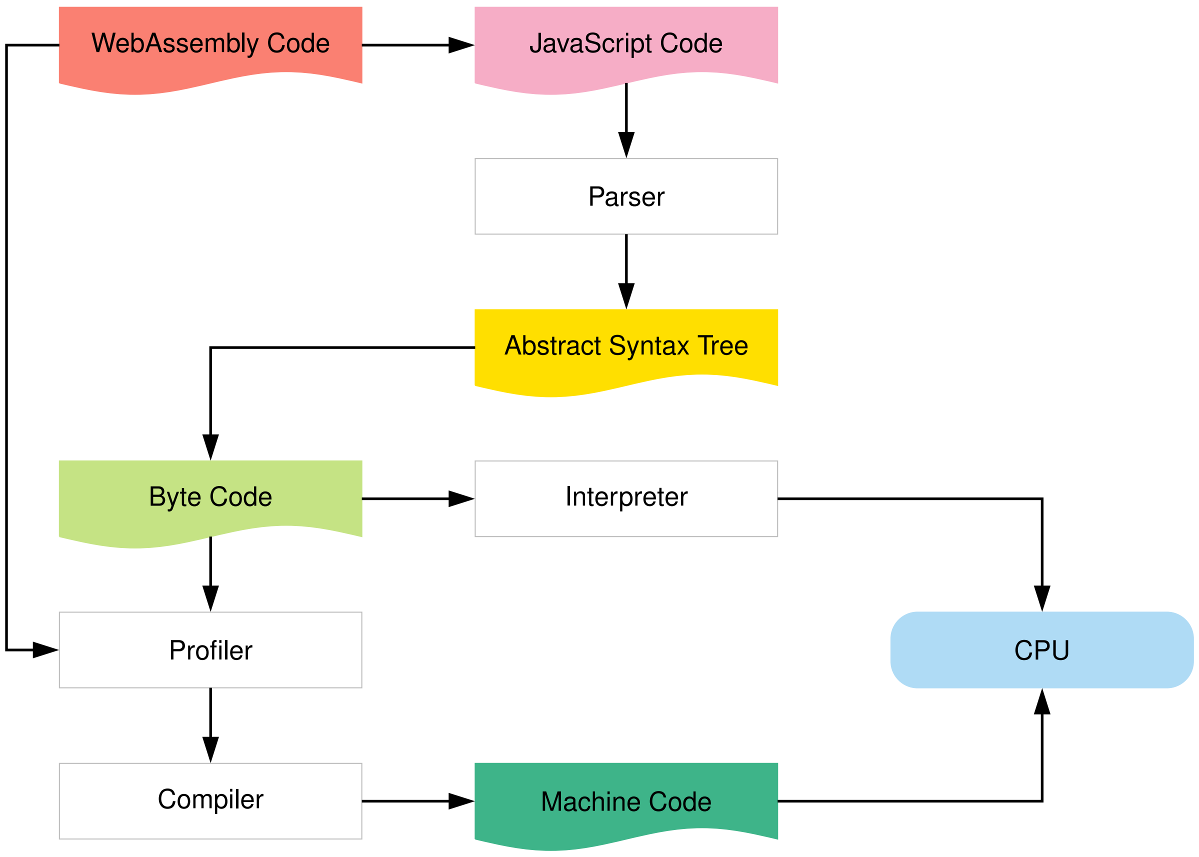

Figure 2.3: A Typical WASM EE Architecture

WASM at its core is based on a virtual ISA, which means that it is not based on any known physical CPU (Rossberg, 2022). Its virtual nature is specifically designed to allow it to be portable, so that it can be embedded into many different environments, such as web browsers. Further to its virtual nature and as with JS, WASM is also sandboxed within its VM, which means that it does not have any capability to access any functions that are located outside of its VM, not unless specific APIs within its VM provide access to those functions. Furthermore, all security concerns must be mitigated by the implementation, so as to prevent potential vulnerabilities, such as side channel attacks.

A typical architecture for a WASM EE is shown in figure 2.3. Immediately one can tell that it re-uses most of the existing components that form part of the JS EE architecture, this re-use is intentional and by design. The design further restricts WASM from accessing certain web browser functions directly, such as those relating to the DOM or Ajax, in order to access those functions WASM has to use JS APIs.

The WASM source code syntax uses what is called Symbolic Expressions. The rationale behind using a rather verbose syntax, is that the creators presumed that one will never write WASM source code directly, instead one would write source code in another language, such as C, and then compile the C source code into WASM source code using a tool such as LLVM.

```
#include <stdio.h>

char *greeting() {
    return "Hello World";
}
```

Listing 2.1: Hello World Function in C

Listing 2.1 shows the source code for a `Hello World` function written in C and listing 2.2 shows the equivalent function in JS.

```
function greeting() {
    return "Hello World";
}
```

Listing 2.2: Hello World Function in JavaScript

```
(module
    (table 0 anyfunc)
    (memory $0 1)
    (data (i32.const 16) "Hello World\00")
    (export "memory" (memory $0))
    (export "_Z8greetingv" (func $_Z8greetingv))
    (func $_Z8greetingv (; 0 ;) (result i32)
        (i32.const 16)
    )
)
```

Listing 2.3: Hello World Function in WebAssembly

The equivalent function written in WASM is shown in listing 2.3, which quite clearly articulates the verbosity of the WASM programming language. Even though WASM is viewed as the next evolutionary step for being able to execute compute-intensive applications within web browsers, there are still various shortcomings with it, such as not having adequate multithreading capabilities, which is a key requirement for creating multithreaded applications.

Even though these shortcomings are being worked on, it is unclear as to what impact the eventual solutions will have and if they will adequately and completely resolve the existing shortcomings. Currently, most web browsers incorporate one or both of the industry standard JS and WASM EEs. In order for this study to propose any further improvements to EEs, that will allow them to execute compute-intensive applications within web browsers, it is important to review the research studies that have been undertaken in this area.

Thus an SLR has been undertaken, in order to highlight the gap in the current research relating to not only the performance of web browser EEs, but also other potential gaps which closely relates to the area of concern, being web browsers.

2.4. A structured literature review

This SLR uses the structure described by Kofod-Petersen (2015), which has specifically been created for SLRs in the field of computer science research, but can also be used in the field of information and communication technology research, given its close relation to computer science. This structure contains three main phases; *planning*, *conducting* and *reporting*.

Kofod-Petersen's (2015) structure also states and describes how the three phases can be further broken down into several steps, where:

Phase 1 involves planning the review and can be broken down into:

1. Identification of the need for a review. (The first objective of this study identified this need)
2. Commissioning a review.
3. Developing a review protocol.
4. Evaluating the review protocol.

Phase 2 involves actually conducting the review and consists of five steps:

1. Identification of research.
2. Selection of primary studies.
3. Study quality assessment.
4. Data extraction and monitoring.
5. Data synthesis.

An intuitive way of depicting this phase is through the PRISMA 2020 flow diagram in figure 2.4 created by Page et al. (2021), for use in the field of health care and medical research, but easily adapted for use in the field of information and communication technology, given its abstract nature. The flow diagram articulates the various steps of the second (conducting) phase, which also allows one to clearly depict the results using the same flow diagram, during the third (reporting) phase.

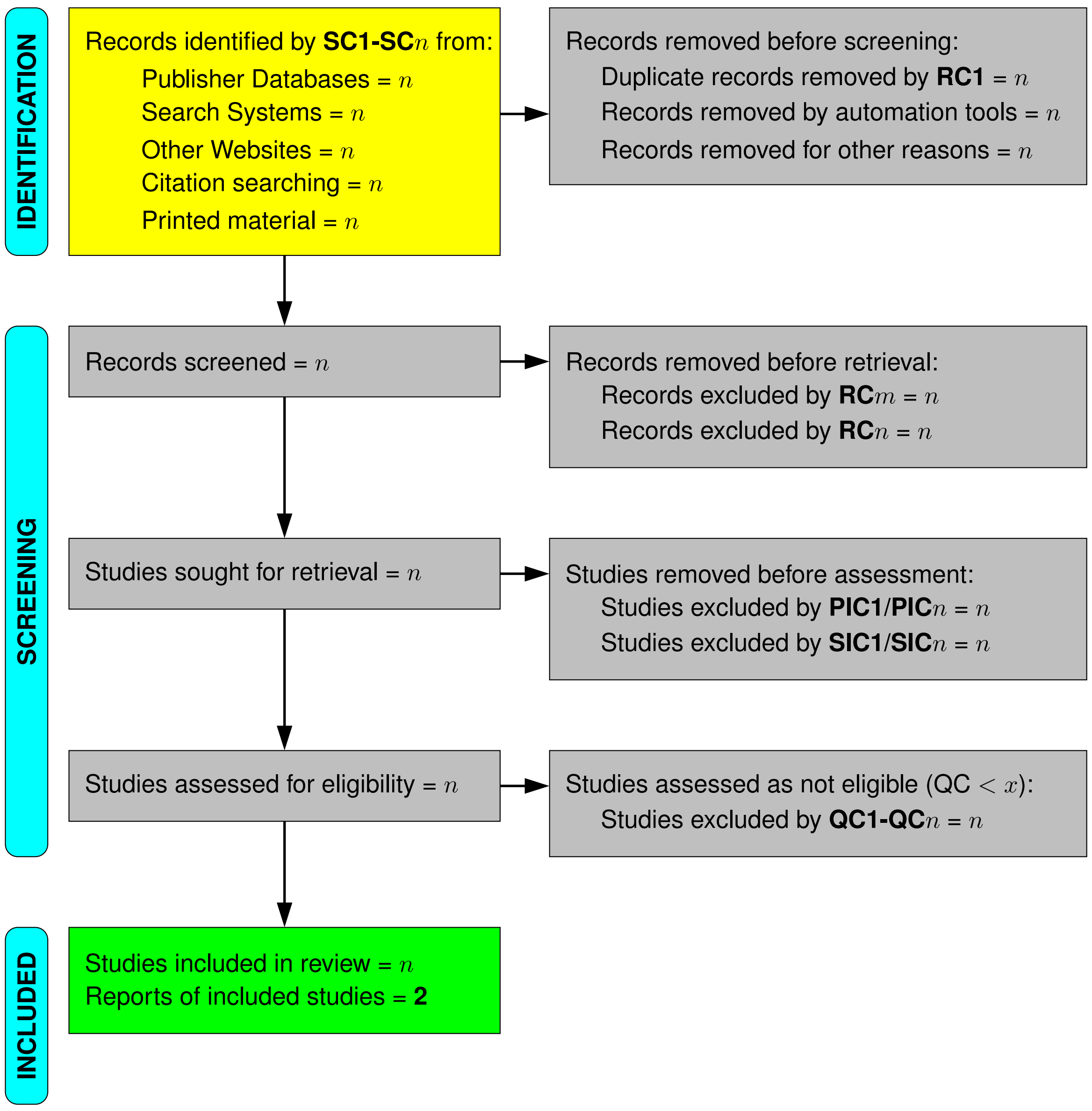
IDENTIFICATION
Records identified by **SC1-SC**n from:
Publisher Databases = n
Search Systems = n
Other Websites = n
Citation searching = n
Printed material = n
Records removed before screening:
Duplicate records removed by **RC1** = n
Records removed by automation tools = n
Records removed for other reasons = n
SCREENING
Records screened = n
Records removed before retrieval:
Records excluded by **RC**m = n
Records excluded by **RC**n = n
Studies sought for retrieval = n
Studies removed before assessment:
Studies excluded by **PIC1**/**PIC**n = n
Studies excluded by **SIC1**/**SIC**n = n
Studies assessed for eligibility = n
Studies assessed as not eligible (QC $< x$):
Studies excluded by **QC1-QC**n = n
INCLUDED
Studies included in review = n
Reports of included studies = **2**

Figure 2.4: Adapted Phase 2 Flow Diagram (Page et al., 2021)

There are three distinct stages within this flow diagram; *identification*, *screening* and *included*. These three stages align well with the Kofod-Petersen (2015) second phase steps, where *identification* covers step 1, *screening* covers steps 2-4, and *included* covers step 5. *Phase 3* deals with reporting on the findings of the review and consists of:

1. Specifying dissemination strategy.

2. Formatting the main report.

3. Evaluating the report.

2.4.1. Introduction

An SLR is a formal process of synthesising the information available from existing research studies. These research studies can be either primary or secondary research studies, where primary ones are given the most focus and referenced the most throughout the research being undertaken. It is important to note that an SLR is in no way a guarantee that one will find all relevant literature pertaining to the research area in question, even so, it is the intention of the researcher to uncover close to all available relevant literature that can be readily accessed today, through various readily available tools.

The reason why we undertake an SLR, besides uncovering all existing relevant literature, is the potential for the SLR to map out any existing solutions before a researcher even attempts to propose their own solutions, while also helping researchers avoid introducing any biases into their work. It also allows researchers to identify gaps within the current knowledge relating to the research area, as well as highlighting the areas where additional research is required. These gaps will be discussed in detail, in the section covering the theoretical grounding of this research.

Publishing the results of the SLR also benefits the research community by allowing others to avoid duplicating the effort required when undertaking an SLR. Given all of these benefits lets look at how this SLR will aid this research by having a brief discussion of it within the desired context.

2.4.2. Background

Dissecting the singular research question of this study:

Research Question

What software architecture and design does a new web browser EE need to comprise of in order to be able to host and execute CIAs with native desktop performance?

One comes to understand that the primary focus of this study is concerned with the performance of web browser EEs. Given also that we have earlier in the chapter provided an introduction to the two industry standard EEs, namely JS and WASM EEs, it stands to reason then that the SLR should primarily focus on those two EEs within the context of performance within web browsers.

Together with this primary focus, a secondary focus based on literature relating to performance from a UX point of view will also briefly be touched upon since the performance aspects of UX directly relates to and is dependent on the performance of EEs. This is because one cannot have an adequate UX if the interactive performance and response is too slow, which is especially true in certain domains, such as gaming, share trading and the remote control of unmanned vehicles and equipment, to name a few.

2.4.3. Aim and objective

As stated, the goal or aim of this study is to develop a new web browser EE, which can be used by any web browser to host and execute CIAs, with native desktop performance. Looking at this together with the first objective of this study, one can derive that this SLR is to provide a comprehensive and organised overview of existing research and literature on web browser EE performance.

Unlike a traditional narrative literature review, which may be more narrative and less structured, an SLR follows a systematic and organised approach. This SLR serves as a foundation for understanding the state of knowledge on web browser EE performance, assessing its quality, summarising existing knowledge, identifying research gaps and trends, creating a framework for understanding, informing future research, supporting evidence-based decision-making, and contributing to the academic discourse.

Further to this, Kofod-Petersen (2015) states: ”It is assumed that a specific problem ($\mathcal{P}$) is tackled using some specific constraints, methods and/or approaches ($\mathcal{C}$) to develop a system, application or algorithm ($\mathcal{S}$)”. So this SLR should seek to answer the four SLR related research questions as proposed by Kofod-Petersen (2015):

RQ1 ”What are the existing solutions to $\mathcal{P}$?”

RQ2 ”How does the different solutions found by addressing RQ1 compare to each other with respect to $\mathcal{C}$?”

RQ3 ”What is the strength of the evidence in support of the different solutions?”

RQ4 ”What implications will these findings have when creating $\mathcal{S}$?”

Where the variables $\mathcal{P}$, $\mathcal{C}$ and $\mathcal{S}$ correlate to:

$\mathcal{P}$ - EEs that are capable of hosting and executing CIAs, with native desktop performance.

$\mathcal{C}$ - Web browser EEs.

$\mathcal{S}$ - A new web browser EE.

2.4.4. Methodology

A study in the context of this SLR refers to any paper that has been published, either electronically or in printed form. A very limited amount of these studies may not have been put through a peer review process, such as personal or corporate blog posts. Where that is known to be the case, it will be clearly identified.

As a first step, a protocol needs to be formulated, the aim of which is to minimise bias by having a pre-defined set of eligibility criteria of what *will*, and what *will not*, be included in the SLR.

2.4.4.1. Protocol

The rationale for the protocol of the SLR for this research is formulated around seeking information that can be used to answer the singular research question within the context of and relating to performance. Using a robust set of objectives will allow the protocol to bolster and ensure the transparency, accountability and integrity of the completed SLR.

The objectives of this SLR are to:

1. Use the search terms from Table 2.1 to find all literature related to web browser EEs, where the studies focus on RIAs, JS and WASM, within the context of performance.
2. Use the selection and removal criteria from section 2.4.4.3. to filter all studies.
3. Use the assessment criteria from Table 2.4 to assign a score to the remaining studies.
4. Remove all studies below the required score.

The remaining subsections following hereto detail the methods to be used in locating, selecting and analysing identified studies, as part of this protocol.

2.4.4.2. Search strategy

The strategy by which various information retrieval systems and databases were searched, was to have as few as possible search terms. Each search term would also be composed of only the technology name and where the result set is initially too large, a context will be added to refine and reduce the result set.

The search term syntax utilises standard Boolean operator syntax rules, whereby parenthesis dictate precedence and where the ***AND***, ***OR*** and ***NOT*** operators assist in filtering out unneeded studies. String grouping, by using quotation marks, was also applied where the technology name is made up of multiple words. This forces the multiple words of the technology name to be seen as one word and prevents the information retrieval systems and databases from producing results for the individual words on their own.

It is important to note that the standard Boolean operators are not case-sensitive, so ***AND***, ***OR*** and ***NOT*** can be written in uppercase or lowercase letters. Additionally, different information retrieval systems and databases may have variations in their specific syntax and support for Boolean operators, so it is essential to refer to the documentation of the system that is to be accessed for precise details on how to construct supported Boolean queries. The table below details the search terms used for this SLR.

Table 2.1: SLR Search Terms

Identifier	Search Term	Context Used
ST1A	JavaScript	No
ST1B	JavaScript AND Performance	Yes
ST2	WebAssembly	No
ST3	”Rich Internet Application”	No

According to Kofod-Petersen (2015) the most prominent computer science databases that should be searched are; ***ACM Digital Library***, ***IEEE Xplore***, ***ISI Web of Knowledge***, ***ScienceDirect***, ***CiteSeerX***, ***SpringerLink*** and ***Wiley Inter Science***. One however has to note that the word database is used rather loosely by Kofod-Petersen (2015), because some of these databases like ***CiteSeerX***, are actually database aggregators or as they are commonly known, search systems.

Search systems do not publish any literature themselves but act as intelligent search engines that are able to search through one or more publisher databases. Often they would also create extra metadata to accompany literature found in publisher databases, such as citation cross-referencing.

Table 2.2: SLR Primary Publisher Databases and Search Systems

Publisher Databases
ACM Digital Library
IEEE Xplore
IngentaConnect
JSTOR
SciELO
ScienceDirect
SpringerLink

Search Systems
EbscoHost
CiteSeerX
Google Scholar
ResearchGate
Scopus
Semantic Scholar
Web of Science

The proposed search systems and publisher databases as suggested by Kofod-Petersen (2015) were used, since this study is broadly situated in the field of computer science. They were also in turn combined with those reviewed in Gusenbauer (2019) and Gusenbauer and Haddaway (2020), to derive two sets of search systems and publisher databases, as per Table 2.2, which will be targeted by this SLR.

Together with this, this SLR will also search through lesser known publisher databases that form part of the ***ACM Digital Library*** through what is called ***The ACM Guide to Computing Literature***. Table 2.3 details these lesser known publisher databases.

Table 2.3: SLR Secondary Publisher Databases and Search Systems

Publisher Databases
Academic Press
Butterworth-Heinemann
Hindawi
IBM
IGI Global
Inderscience Publishers
International WWW Conferences Steering Committee
IOS Press
John Wiley & Sons
Kluwer Academic Publishers
Pergamon Press
Usenix Association
VLDB Endowment

Search Systems
Connected Papers

One secondary search system, ***Connected Papers*** was also reviewed. The benefit of this search system is that it presents one with an elegantly structured graph of one's primary literature and how they are connected to other related literature. As with all other literature result sets, not all connected literature is of relevance to this study and they were reviewed following the same protocol as with all other discovered literature.

2.4.4.3. Literature selection

As a secondary phase, we can use the derived selection and removal criteria to filter the overall set of literature once they have been identified, whilst deferring to the ranking methods of the various search systems and publisher databases to provide the most representative and relevant set of results during the initial phase of discovery. Given that the initial set of identified literature could comprise of several million items, these criteria, once used, should provide a more manageable set of results which can then be quality assessed.

Selection criteria:

SC1 Empirical studies on JS performance in the context of EEs.

SC2 All empirical studies on WASM performance in the context of EEs, as WASM has only been publicly available since 2018.

SC3 All empirical studies on RIA performance in the context of EEs, as the expectation was to not find many studies on this subject while searching publisher databases.

Removal criteria:

RC1 Duplicate literature of the same study. Even though this is frowned upon in the scientific publication world, it does occasionally occur.

RC2 JS related literature before 2014, which would provide this study with a 10-year review period from 2014 to 2023. Note that JS was created in 1995, but did not become mainstream until at least 2002.

RC3 Literature that is not in English, unless an English translation is available.

2.4.4.4. Quality assessment

The quality assessment stage aims to eliminate studies that are not thematically relevant, by using the criteria listed in Table 2.4, namely the relevant primary inclusion criteria (PIC), secondary inclusion criteria (SIC) and initial quality criteria (QC).

Table 2.4: Inclusion and Quality Criteria

Identifier	Criteria
PIC1	The primary focus of the study is $\mathcal{P}$
PIC2	The study presents empirical results in relation to $\mathcal{P}$ and/or $\mathcal{S}$
SIC1	The context of the study is $\mathcal{C}$
SIC2	The study proposes $\mathcal{S}$
QC1	The aim of the study is clearly stated
QC2	There is a clear comparison between the assessed study and other studies

All studies are subjected to an initial quality assessment and then evaluated against eight additional quality criteria as outlined in Table 2.5. By combining those eight criteria with the initial two, we can assign each study a score ranging from 1 to 10. Depending on how well a study meets the quality criteria, it can receive a full point (1), a half point (½), or zero points (0).

For this SLR, we will primarily focus on studies that only have zero points up to a maximum of two quality criteria, while also scoring at least seven points or higher in total.

Table 2.5: Additional Quality Criteria, including QC1 and QC2

Identifier	Criteria
QC1	The aim of the study is clearly stated
QC2	There is a clear comparison between the assessed study and other studies
QC3	Are design decisions justified when proposing a solution
QC4	Can one reproduce the test data set
QC5	How appropriate is the test data set size
QC6	Can one thoroughly explain and reproduce the experiments
QC7	Can one distinguish as to how the algorithms from the assessed study compares to other algorithms
QC8	Can one explain and justify the performance metrics used
QC9	Are the results and findings clearly explained
QC10	For the findings presented, does the test evidence support it

2.4.4.5. Data extraction and monitoring

Once the final set of studies have been identified, a data extraction stage will be undertaken to extract certain data elements from the studies, which will later be discussed. The primary data elements to be extracted are as follows:

1. **Title**
2. **Authors**
3. **Publication Type**
4. **Year**
5. **QA Score**

Secondary data elements that will also be extracted are:

1. **Testbeds**
2. **Benchmarking EEs**
3. **Benchmarking Algorithms**

The secondary data elements will assist the researcher in formulating this study's research instruments, allowing this study to closely align its benchmarking with what has been previously used, while at the same time being critical of where it may be found to be inadequate and/or inappropriate.

Further to this, there is an intent by the researcher to continually monitor the publisher databases and search systems to ascertain if any new studies have been published since the initial SLR was completed. This is to ensure that this study has reviewed at least the last decades worth of available studies.

2.4.4.6. Data synthesis

Data synthesis in the context of this study, refers to the process of bringing together and integrating data from various published studies in order to draw meaningful conclusions or develop a comprehensive understanding of a particular research question. It involves analysing, summarising, and combining data from different studies or datasets to generate new insights or to answer the research question that may not be addressed by individual studies alone.

Data synthesis is commonly used in systematic reviews and meta-analyses, where researchers aim to systematically review and analyse existing studies on a specific topic. In these cases, data synthesis involves extracting relevant information from multiple studies, evaluating the quality of them, and then combining the results to provide a more comprehensive and statistically robust overview of the available evidence as was done through the SLR of this study, together with the QA assessments performed.

Furthermore, to the data synthesis performed during the SLR of this study, no statistical analysis was performed on the set of QA assessed studies with a score equal to or above the specified threshold, nor on their provided data. In addition, this SLR contains some fundamental facts extracted and summarised from the identified set of studies, which are presented and discussed in the subsequent sections. Some insights into areas of research similar to this study, that require further investigation are also identified and briefly discussed.

2.4.4.7. References found and discussion

The set of studies unearthed through the SLR is believed to be as comprehensive and extensive as possible given the initial result set, before any refinements or filtering, was at the time about 76 thousand studies from a total of about 60 million studies across all of the primary databases alone. Studies were also not only gathered from the various publisher databases and search systems, but also by cross-checking and analysing the references section of the studies assessed for eligibility.

Given the age and mainstream use of JS, it was expected that it would result in the biggest portion of the initial result set, which comprised of 73 thousand studies, followed by RIA with about two thousand studies and finally WASM with about a thousand studies. Subsequently, after adding the search context to the JS search, the initial result set was reduced to about 50 thousand studies, which is depicted as a breakdown across the publisher databases, search systems and citations in the *Identification* section of 2.5.

The subsequent metrics and process flow depicted in figure 2.5 show the various stages of refinement and filtering that the discovered set of studies were put through in order to derive at the final set of 133 studies to be assessed as being directly or indirectly related to this study, as well as studies relating to areas of unrelated research. After assessing the 133 studies for quality and alignment to this study it was found that 16 studies are closely aligned with this study and that they form the foundational base hereof. The two reports that were published refer to this study as well as to a conference paper that was presented and based on these findings.

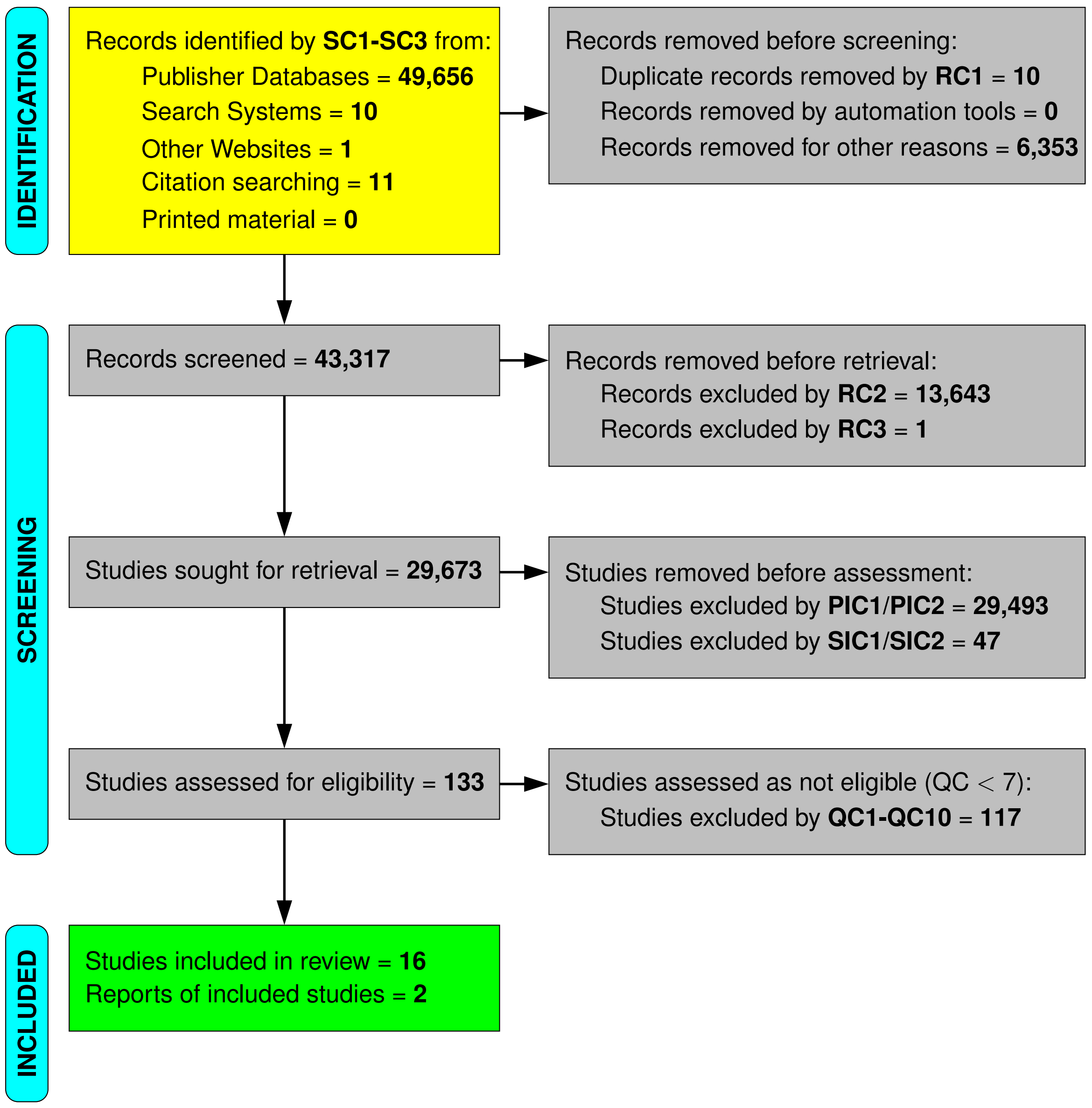
IDENTIFICATION
Records identified by **SC1-SC3** from:
Publisher Databases = **49,656**
Search Systems = **10**
Other Websites = **1**
Citation searching = **11**
Printed material = **0**
Records removed before screening:
Duplicate records removed by **RC1** = **10**
Records removed by automation tools = **0**
Records removed for other reasons = **6,353**
SCREENING
Records screened = **43,317**
Records removed before retrieval:
Records excluded by **RC2** = **13,643**
Records excluded by **RC3** = **1**
Studies sought for retrieval = **29,673**
Studies removed before assessment:
Studies excluded by **PIC1/PIC2** = **29,493**
Studies excluded by **SIC1/SIC2** = **47**
Studies assessed for eligibility = **133**
Studies assessed as not eligible (QC < 7):
Studies excluded by **QC1-QC10** = **117**
INCLUDED
Studies included in review = **16**
Reports of included studies = **2**

Figure 2.5: Phase 2 Flow Diagram With Results

The studies shown in Table 2.6 together with the databases from which they were extracted, have been chosen to reflect and describe the present state of web browser EEs with relation to and in the context of performance, while also providing insight into possible answers to the singular research question of:

Research Question

What software architecture and design does a new web browser EE need to comprise of in order to be able to host and execute CIAs with native desktop performance?

Table 2.6: SLR Studies Assessed for Eligibility

Publisher Databases	Total	Studies
ACM Digital Library	55	(Ahn et al., 2014; Arteaga et al., 2020; Bhansali et al., 2022; Bian et al., 2019; Bourgoin & Chailloux, 2015; Calegari et al., 2019; Casteleyn et al., 2014; Chandra et al., 2016; Choi et al., 2019; Fukuda & Yamamoto, 2008; Gong et al., 2015; Haßler & Maier, 2021; Herrera et al., 2018; Hilbig et al., 2021; Ho et al., 2017; Hockley & Williamson, 2022; Jansen & van Groningen, 2016; Kataoka et al., 2018; Kharraz et al., 2019; Konoth et al., 2018; Lehmann & Pradel, 2022; Liu et al., 2022; Ma et al., 2019; Mäkitalo et al., 2021; Marion & Jomier, 2012; Matsakis et al., 2014; Miller, 1968; Møller, 2018; Musch et al., 2019b; Na et al., 2016; Nießen et al., 2020; Park et al., 2016; Park et al., 2018; Pinckney et al., 2020; Puder et al., 2013; Reiser & Bläser, 2017; Rempel, 2015; Romano et al., 2020; Romano & Wang, 2020; Rossberg et al., 2018; Salim et al., 2020; Selakovic & Pradel, 2016; Serrano, 2018; Serrano, 2021; Stiévenart et al., 2022; Sun & Ryu, 2018; Titzer, 2022; van Hasselt et al., 2022; Watt, 2018; Watt et al., 2019; Watt et al., 2020; Wirfs-Brock & Eich, 2020; Yan et al., 2021; Zakai, 2011; Zhao et al., 2019)
IEEE Xplore	29	(Cao et al., 2017; De Macedo et al., 2021; De Macedo et al., 2022; DiPierro, 2018; Dot et al., 2015; Frankston, 2020; Heo et al., 2016; Liu, 2019; Ménétrey et al., 2021; Park et al., 2017; Radhakrishnan, 2015; Rahimi, 2021; Šipek et al., 2019; Šipek et al., 2021; Southern & Renau, 2016; Spies & Mock, 2021; Sun et al., 2019; Szewczyk et al., 2022; Tushar & Mohan, 2022; Ueda & Ohara, 2017; Verdú & Pajuelo, 2016; Vilk & Berger, 2014; Wagner, 2017; Wang, 2021; Wang, 2022; Wang et al., 2019; Wen et al., 2020; Zakai, 2018; Zhuykov et al., 2015)
SpringerLink	23	(Aponte, 2020; Belkin et al., 2019; Borisov & Kosolapov, 2020; Bruyat et al., 2021; Fink & Flatow, 2014; Fras & Nowak, 2019; Kienle & Distante, 2014; Koper & Woda, 2022; Kozlovičs, 2020; Liu & You, 2022; Lyu, 2021; McAnlis et al., 2014a; McAnlis et al., 2014b; McAnlis et al., 2014c; Mikkonen et al., 2019; Musch et al., 2019a; Nicula & Zota, 2022; Odell, 2014; Ortiz, 2022; Taivalsaari et al., 2018; Taivalsaari & Mikkonen, 2018; Yu et al., 2020; Zhuykov & Sharygin, 2017)
ScienceDirect	3	(Brito et al., 2022; Huber et al., 2022; Van Es et al., 2017)

(continued on next page)

Table 2.6: SLR Studies Assessed for Eligibility (continued)

Publisher Databases	Total	Studies
IngentaConnect	1	(Auler et al., 2014)
Kluwer Academic Publishers	1	(Cho et al., 2015)
Other	21	(Bormann, 2018; Choi & Moon, 2019; Doherty & Thadani, 1982; Eich, 2015; Fette & Melnikov, 2011; Fraternali et al., 2010; Hardie, 2016; Herman et al., 2014; Jangda et al., 2019; Jiang & Jin, 2017; Lehmann et al., 2020; Letz et al., 2018; Lubbers & Greco, 2010; Malle et al., 2018; Mazaheri et al., 2022; McManus, 2018; Rossberg, 2022; Szabó & Nehéz, 2019; Taivalsaari & Mikkonen, 2017; Yin et al., 2015; Zakai, 2017)
Total Studies	**133**	

Once the quality assessment was completed, some studies were discovered that alluded to additional areas of research unrelated to this study. These studies can be grouped into the following areas of research which other researchers may wish to investigate in future studies.

The areas of research, in no particular order are:

1. Those aspects concerning the energy usage required to execute JS and WASM applications, particularly on limited-power or low-power devices like Internet of Things (IoT) devices and smartphones, as highlighted by De Macedo et al. (2021).

2. Those aspects that relate to attacks targeting a user's web browser, that aims to gain control of the user's web browser and underlying computer system, which was investigated by Nicula and Zota (2022) and Rahimi (2021).

3. Those aspects regarding the use of JS and/or WASM as a way to break hardware security through side-channel attacks, as observed by Mazaheri et al. (2022).

4. Those aspects concerning obfuscating JS and/or WASM code, which is a technique employed to disguise malicious code as explored by Bhansali et al. (2022), Borisov and Kosolapov (2020) and Sun et al. (2019).

5. Those aspects relating to security vulnerabilities and code efficiencies of the generated WASM code when compared to the natively equivalent C or C++ code as examined by Brito et al. (2022), Hilbig et al. (2021) and Liu (2019).

6. Those aspects regarding security, where malicious code is injected into WebGL and/or WASM-based applications, with the aim of sneaking them into a web browser. Distributed password cracking tools, denial of service slaves, and distributed cryptocurrency mining or cryptojacking, are examples of such malicious code as researched by Belkin et al. (2019), Bhansali et al. (2022), Bian et al. (2019), Hilbig et al. (2021), Kharraz et al. (2019), Konoth et al. (2018), Musch et al. (2019b), Romano et al. (2020) and Yu et al. (2020).

2.5. The theoretical grounding

Following on from the brief theoretical background provided in chapter 1, we can now expand on that by reviewing the studies identified by this study's SLR, which further seeks to reinforce our theoretical grounding for this study. We start by looking at the earliest studies related to this research, in other words, RIA, where the SLR identified six relevant RIA studies that warranted further in-depth analysis.

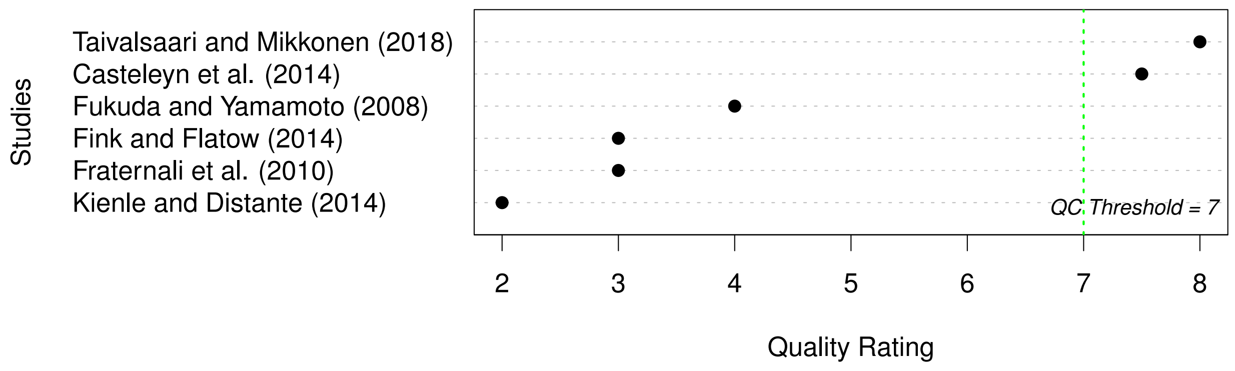

Figure 2.6: Quality Assessment of Six Relevant RIA Studies

While reviewing the SLR data, one can quickly form a viewpoint that most of the six studies identified and assessed that pertain to RIA, as per figure 2.6, can be discounted. This is because none of them cover any aspects pertaining to performance analysis or evaluation of RIA-based web browser EEs, thus they have no bearing on this study. However, some interesting ideas were discovered which gives thought to some innovative ways in which one could possibly try to solve the problem that this study is focused on.

One was that of Mozilla's project *XUL*[4], which is an XML-based language that aids in the building of feature-rich and cross-platform applications (Fraternali et al., 2010). Another one was that of Taivalsaari and Mikkonen (2018), where they reviewed a key technical manifestation as they put it, which is the *Lively Kernel*[5] system, originally created by Sun Microsystems. Other than those few interesting aspects, the remainder of the studies have nothing more to offer.

On the whole, the RIA technologies can be viewed as a historical attempt at creating desktop equivalent applications within a web browser (Casteleyn et al., 2014; Kienle & Distante, 2014), with varying success. It can be deduced from the data presented in figure 2.7 that studies focused on RIA technologies have now mostly ceased. One simply has to look at the small

[4]https://wiki.mozilla.org/XUL
[5]http://lively-kernel.org/

number of studies discovered to form this viewpoint, in comparison with the large number of studies that were and continue to be discovered for both JS and WASM. Although studies in JS do seem to be dwindling since 2019, which correlates to an increase in studies into WASM at that time.

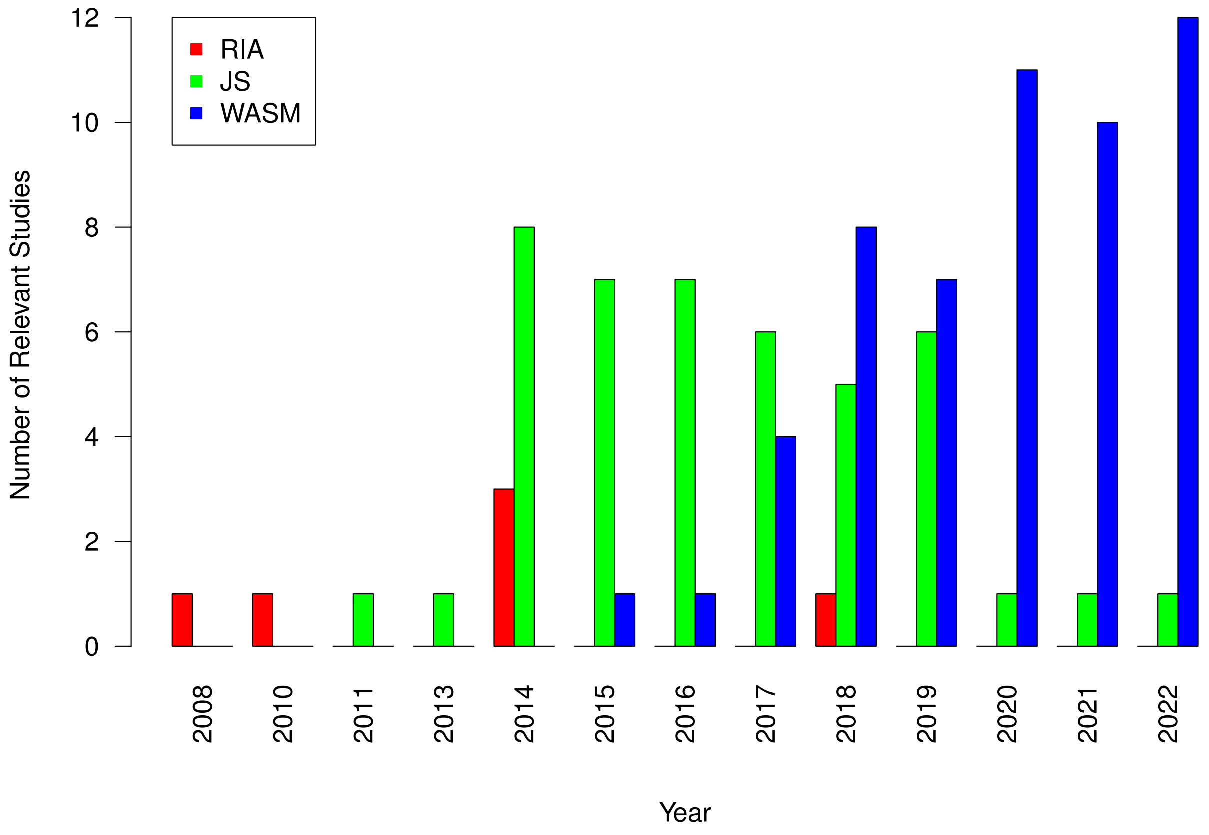

Figure 2.7: Distribution of Relevant Studies by Year

Next, we turn to JS, where the SLR assisted in identifying 44 relevant studies, with 6 studies being directly applicable to this study, see figure 2.8. The foremost web browser EE is considered to be JS or simply the JavaScript Virtual Machine (JSVM), a technology that gave rise to what is ordinarily referred to as *Web 2.0*, the next progression of the WWW that revolutionised the web browser. *Web 2.0* can be characterised by a shift in how the internet is used, by being more interactive and interconnected, and can be described by an increased presence of user-generated content and enhanced user-friendliness.

The reviewed JS studies can be grouped into several common research themes or areas:

1. The Emscripten and asm.js derivatives.

2. Enhancing the JS type system.

3. Improving JS compilation.

4. Analysing JS performance and its underlying EE.
5. Improving the JS programming language.
6. Performance tuning the JSVM to improve usability.
7. JSVM caching and memory management techniques.
8. The remaining studies focused on JS reviews, spanning a decade or more.

Studies relating to the research area of *Improving JS compilation* can also be further categorized into subfields such as:

1. AOT compilation.
2. JIT compilation.
3. JS compilation tooling.

Given the dynamically typed nature of JS, some studies have proposed possible solutions by pairing a statically typed system with JS instead of the default dynamically typed one. That would allow for the use of AOT compilation or to improve the JIT compilation performance, where the former is preferred. Other studies explored ways of using tools such as Emscripten, as a way of cross-compiling programming languages such as C and C++ that are statically typed, into JS. However, this approach is irrelevant to this study and can be seen as counter productive, as it introduces additional execution overhead to applications that originally did not require it.

Additional intriguing technologies that were revealed through the JS segment of the SLR were those relating to Microsoft's Xax browser plugin model, Google's Native Client (NaCl) and Portable Native Client (PNaCl) (Douceur et al., 2008). Xax offers an innovative method for executing legacy applications within a web browser by providing features such as native code execution, consistent OS and library interfaces, and memory isolation. However, only a single research paper has ever been published in regards to Xax, there has also been no public review of any prototype nor any publication of any scientific data, given that one has no alternative but to regard the concept as theoretical.

More exploration of Xax, NaCl, and PNaCl is justified, as those technologies focus on using web browsers as a delivery mechanism, while also using them to distribute and execute native applications. The logical assumption behind these studies is that to achieve the same performance of native applications executing on a desktop within web applications, one would require the delivery of native applications through the web browser. As such, those completed studies could provide a possible foundation for the proposed solution in this study.

The studies on JS shed some light and understanding of what causes native desktop environments to outperform JS EEs. This performance gap is primarily due to the dynamically typed system that JS is based on, which requires interpretation at runtime. Since native applications are executed directly by the underlying ISA, it is nearly impossible for an interpreted language like JS to achieve the same level of performance due to the additional processing and overhead that is incurred by the interpreter.

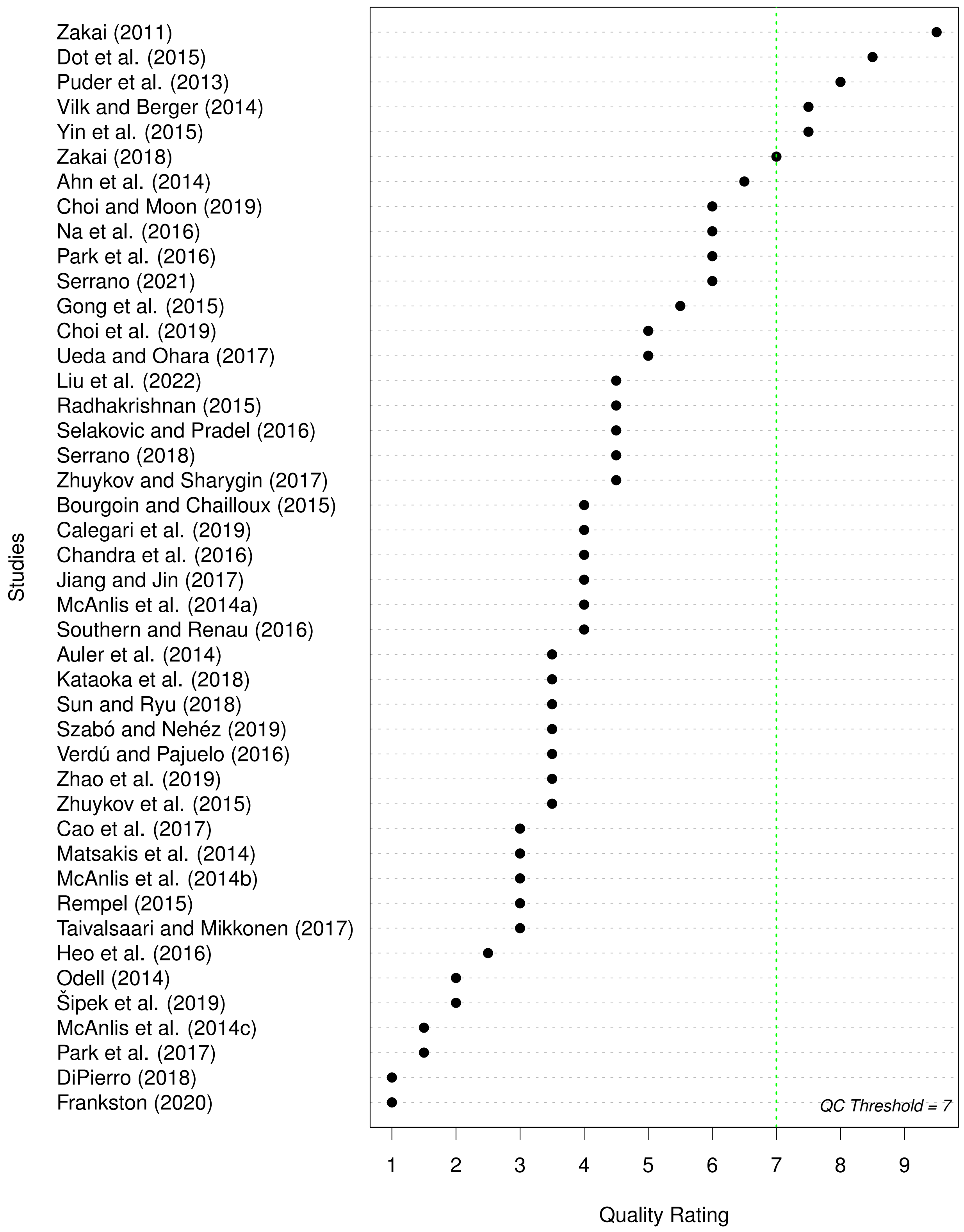

Figure 2.8: Quality Assessment of 44 Relevant JS Studies

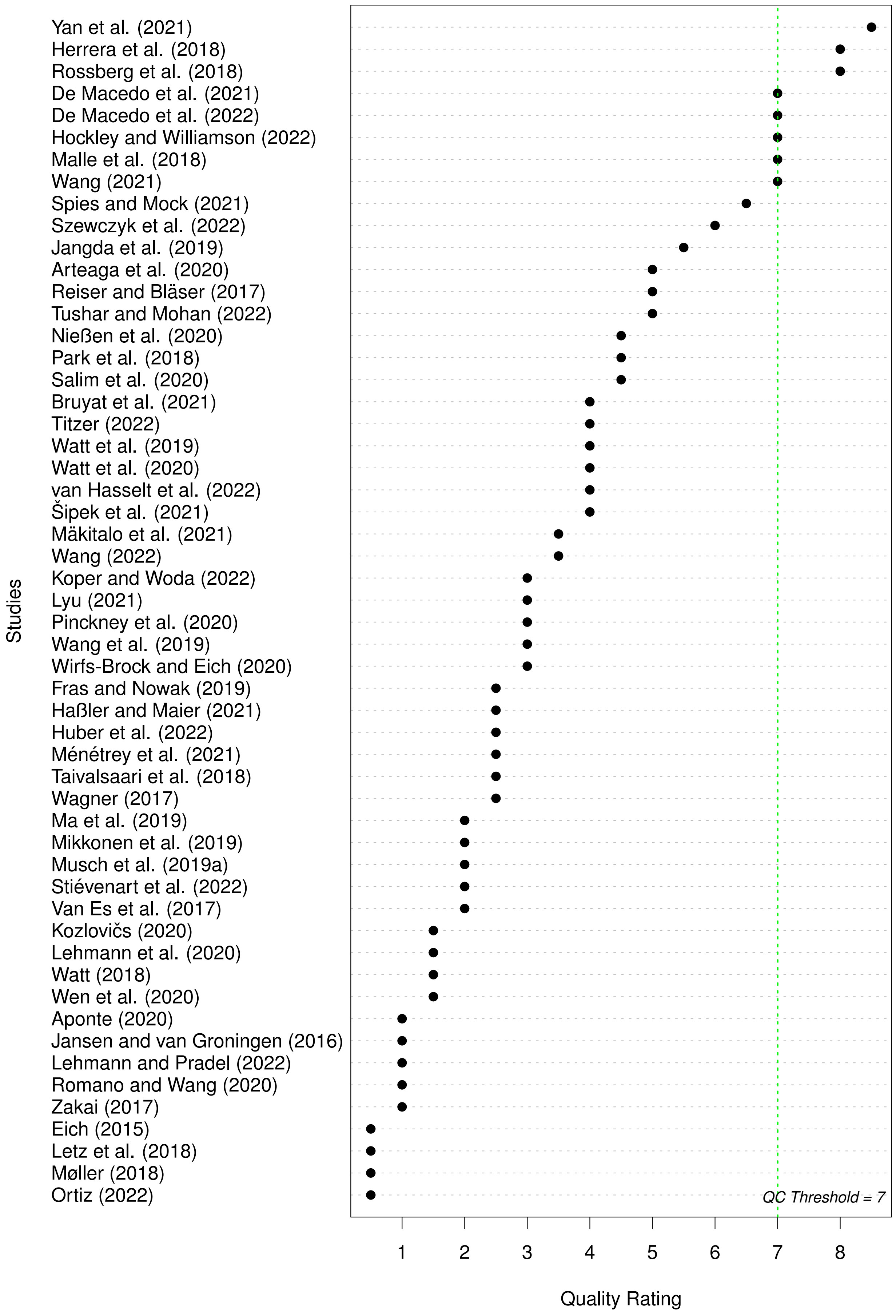

Figure 2.9: Quality Assessment of 54 Relevant WASM Studies

Revisiting the latest advancements in EE technologies, WASM emerged as a key focus, with 54 relevant studies identified. Of these, eight were deemed directly relevant to this research, as shown in figure 2.9. WASM was first announced in 2015 (Eich, 2015) with an initial release in 2017 (Wagner, 2017; Zakai, 2017) and is considered to be the next evolutionary step in the advancement of Web applications. It is an intermediary software-based ISA, not linked to any hardware-based ISA and its compiled code is distributed in a binary format, unlike JS which is distributed in a plain text format.

Similar to JS, the studies evaluated for WASM can be grouped into several common research areas or themes:

1. Introduction to WASM and its preceding technologies.
2. Reusing the JSVM for other programming languages such as WASM.
3. Cross-compilation other programming languages to WASM.
4. Analysing WASM performance and energy consumption.
5. Caching and/or reusing AOT compiled WASM code.
6. Exploring the WASM architecture and specification.
7. Analysing WASM malware, vulnerabilities and obfuscation.
8. The development of SPAs
9. The remaining studies that delved into UI/UX and the overall architectural aspects of the web browser.

Given that WASM has only been around for about six years at the time of this study, it is a fairly new technology and as expected quite a few studies have been covering various aspects of it to date. At least 18 or one third of the relevant WASM studies deal with aspects pertaining to performance and/or energy consumption in comparison to both JS and native applications. This is not surprising given its predecessor's known limitations when it comes to performance and serves to again highlight the overall need to improve on the performance thereof.

The directly relevant studies into WASM, that provide foundational viewpoints and data for this study are those by De Macedo et al. (2021), De Macedo et al. (2022), Herrera et al. (2018), Hockley and Williamson (2022), Malle et al. (2018), Rossberg et al. (2018), Wang (2021) and Yan et al. (2021). The study by Rossberg et al. (2018) is the *primary* foundational study on which this study is based. Rossberg et al.'s (2018) study offers valuable insights into the performance of current web browser EEs as well as providing useful data that indicates that further research into the field is warranted.

The number one takeaway from these directly relevant studies, as well as from the non-directly relevant studies are that the performance of WASM applications as compared to JS applications, as well as their equivalent native desktop applications, is that the results are quite varied. In some instances, the WASM applications will perform better

(Malle et al., 2018) and in other instances worse (De Macedo et al., 2021; De Macedo et al., 2022; Herrera et al., 2018; Hockley & Williamson, 2022; Jangda et al., 2019; Rossberg et al., 2018; Wagner, 2017; Yan et al., 2021).

The paper by Malle et al. (2018) was especially interesting from the viewpoint that it found certain WASM applications to be faster than their equivalent native desktop applications, but it could not explain why that is and that further investigation was required. The second takeaway looking at the energy consumption of WASM applications as compared to JS applications, as well as their equivalent native desktop applications, is that the results are also quite varied. Native desktop applications use less energy than WASM applications, which in turn use less energy than JS applications (De Macedo et al., 2022; Haßler & Maier, 2021; van Hasselt et al., 2022)

Then looking at the various benchmarking suites and techniques used across the relevant WASM studies, we find that several studies used PolyBench/C as part of their benchmarking methodology (Nießen et al., 2020), while the study by Jangda et al. (2019) argues that any benchmarking using PolyBench/C is flawed because the benchmarks use applications that are too small to fully benchmark performance. This will need to be considered and mitigated when formulating the benchmarking methodology of this study.

Other benchmarking techniques used consist of subsets from the Rosetta Code[6], which is a programming chrestomathy Web site (Arteaga et al., 2020; De Macedo et al., 2021; Malle et al., 2018), as well as compute-intensive algorithms such as those relating to Prime numbers, Fibonacci numbers, Floyd-Warshall paths, Huffman coding, permutations, Fast Fourier transforms and the comparison as well as sorting of millions of random numbers. All of which are good options which can be looked at when formulating the benchmarking methodology of this study.

Table A.1 summarises all of the benchmarking suites discovered through the SLR, while Table 2.7 maps those discovered benchmarking suites to the various relevant studies of this SLR together with how the performance results of those studies were measured. Less than half of the relevant studies articulated some form of performance results, where most of them were measured across multiple runs from which the arithmetic mean ($\bar{x}$) or geometric mean (b) were taken.

The difference between using the arithmetic and geometric means for performance benchmarking (Szewczyk et al., 2022; Van Es et al., 2017) is well documented by Fleming and Wallace (1986), where they argue that the geometric mean is preferable for performance benchmarking over the arithmetic mean in certain situations. They emphasise two primary reasons highlighting the advantages of the geometric mean in performance benchmarking, particularly when dealing with data involving growth rates, ratios or situations where extreme values might significantly affect the results. These two primary reasons are:

[6]https://rosettacode.org/

1. **Sensitivity to Changes**: The geometric mean is less sensitive to extreme values or outliers than the arithmetic mean. In performance benchmarking, outliers or extreme values can skew the data and distort the overall picture of performance. The geometric mean's property of mitigating the impact of extreme values makes it more robust for comparisons across different entities or time periods.

2. **Multiplicative Nature**: Performance measures often involve ratios or percentages that represent growth rates or other multiplicative relationships. The geometric mean is suitable for dealing with multiplicative processes, as it preserves the proportional relationships between different observations. It accurately represents the rate of change over time, making it a more appropriate measure for performance comparisons in certain contexts.

Table 2.7: Relevant Studies Mapped to Benchmarking Suites

Study	**Benchmarking Suites**	**Result Measurements**
Ahn et al. (2014)	JSB, KRA, OCT and SUN	*Unspecified*
Arteaga et al. (2020)	RS1	*Unspecified*
Auler et al. (2014)	CLG	*Unspecified*
Bourgoin and Chailloux (2015)	AL1	*Indirectly specified*
Chandra et al. (2016)	OC1, SUN and JE1	*Indirectly specified*
Choi et al. (2019)	OCT	*Unspecified*
Choi and Moon (2019)	AL2	*Indirectly specified*
De Macedo et al. (2021)	CL1 and RS2	*Unspecified*
De Macedo et al. (2022)	CL1 and RS2	*Unspecified*
Dot et al. (2015)	KRA, OCT and SUN	9 warm up runs $\prec \bar{x} = x_1$
Gong et al. (2015)	OCT and SUN	*Unspecified*
Heo et al. (2016)	JSB	$\bar{x} = \frac{1}{5}\sum_{r=1}^{5} x_r$
Herrera et al. (2018)	OST	$b = \left(\prod_{r=1}^{30} x_r\right)^{\frac{1}{30}}$
Jangda et al. (2019)	PBC, SP06 and SP17	*Unspecified*
Jansen and van Groningen (2016)	AL3	*Unspecified*
Kataoka et al. (2018)	SUN	*Unspecified*
Koper and Woda (2022)	AL4	$b = \left(\prod_{r=1}^{50} x_r\right)^{\frac{1}{50}}$
Lehmann et al. (2020)	SP17	*Unspecified*
Matsakis et al. (2014)	SUN	*Unspecified*

(continued on next page)

Table 2.7: Relevant Studies Mapped to Benchmarking Suites

Ménétrey et al. (2021)	PBC	$\bar{x} = \frac{1}{3}\sum_{r=1}^{3} x_r$
Na et al. (2016)	OCT, SUN and PBC	$b = \left(\prod_{r=1}^{10} x_r\right)^{\frac{1}{10}}$
Nießen et al. (2020)	PBC	$\bar{x} = \frac{1}{100}\sum_{r=1}^{100} x_r$
Park et al. (2016)	OCT	$\bar{x} = \frac{1}{10}\sum_{r=1}^{10} x_r$
Park et al. (2017)	SUN	*Unspecified*
Park et al. (2018)	OCT	*Unspecified*
Pinckney et al. (2020)	AL5	*Unspecified*
Puder et al. (2013)	AL6 and CLG	*Unspecified*
Radhakrishnan (2015)	KRA, OCT and SUN	$b = \left(\prod_{r=1}^{10} x_r\right)^{\frac{1}{10}}$
Reiser and Bläser (2017)	AL7	*Unspecified*
Rossberg et al. (2018)	PBC	*Unspecified*
Salim et al. (2020)	CLG, JET and PBC	*Unspecified*
Serrano (2018)	OCT, JET and SUN	$\bar{x} = \frac{1}{30}\sum_{r=1}^{30} x_r$
Serrano (2021)	AL8 and JET	*Unspecified*
Šipek et al. (2021)	AL9	50 warm up runs $\prec \bar{x} = \frac{1}{300}\sum_{r=1}^{300} x_r$
Southern and Renau (2016)	OCT	$b = \left(\prod_{r=1}^{500} x_r\right)^{\frac{1}{500}}$
Spies and Mock (2021)	PBC	$b = \left(\prod_{r=1}^{15} x_r\right)^{\frac{1}{15}}$
Szewczyk et al. (2022)	PBC and SP17	*Unspecified*

(continued on next page)

Table 2.7: Relevant Studies Mapped to Benchmarking Suites

Titzer (2022)	PBC	$\bar{x} = \frac{1}{100}\sum_{r=1}^{100} x_r$
Tushar and Mohan (2022)	AL10	*Unspecified*
Ueda and Ohara (2017)	ACM	*Unspecified*
Van Es et al. (2017)	LAR	*Unspecified*
Wang (2021)	PBC	$b = \left(\prod_{r=1}^{10} x_r\right)^{\frac{1}{10}}$
Wang (2022)	WAB	*Unspecified*
Wen et al. (2020)	SP06	$\bar{x} = \frac{1}{5}\sum_{r=1}^{5} x_r$
Yan et al. (2021)	PBC and CHS	$b = \left(\prod_{r=1}^{5} x_r\right)^{\frac{1}{5}}$
Zakai (2011)	AL11	*Unspecified*
Zakai (2018)	AL12	*Unspecified*

One innovative technology that was brought to light is that of the WebAssembly System Interface (WASI) (Haßler & Maier, 2021; Lehmann et al., 2020; Ménétrey et al., 2021; Salim et al., 2020; Spies & Mock, 2021; Stiévenart et al., 2022; Szewczyk et al., 2022; Wang, 2022), which is both an Application Binary Interface (ABI) as well as an Application Programming Interface (API) with a POSIX-like set of syscall functions, specifically adapted to suit the needs of WASM.

WASI is intended to be modular and as portable as WASM is, by not being bound to any one specific ISA, OS or software library. WASI gives rise to some interesting possibilities and ideas, for instance, what if one embedded a WASI-like layer within a sandbox that could host and execute NaCl-like applications or even plain native applications. These applications could then easily be downloaded through the web and safely executed on any compatible device, without the need to be interpreted, converted or compiled.

Other studies also discovered, act as supplementary evidence in support of this study, where they provide further insights into other areas that may relate to or influence aspects of performance, such as usability and UX. These studies include those by Miller (1968) and Doherty and Thadani (1982).

With the advent of computer systems, it was originally argued by psychologists such as Miller (1968), that a response time of two seconds from a computer system, was the longest that a user should wait, before their attention began to stray. Later though, Doherty and Thadani's (1982) observed that a sub-400 millisecond response time dramatically increased a user's interactions or the number of transactions that they are able to complete, at different skill levels.

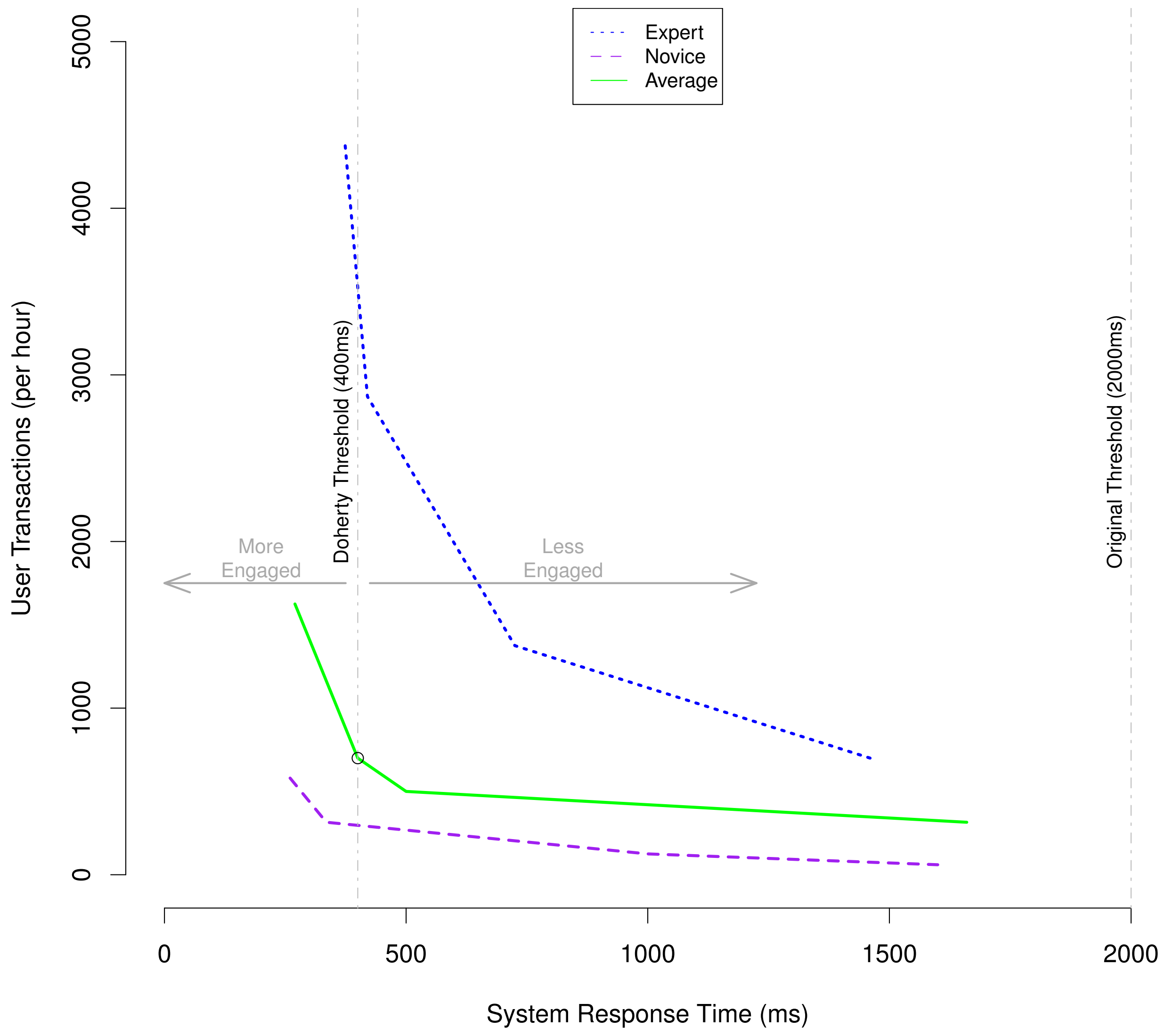

Figure 2.10: Adapted Doherty Threshold (Doherty & Thadani, 1982)

This threshold is commonly known as the *Doherty Threshold* today, where if a computer system responds in sub-400 milliseconds, it is deemed to have exceeded the *Doherty Threshold* and may even become addictive to users in certain use cases, such as online gaming. It is thus believed that having web browser EEs respond in under sub-400 milliseconds would not only provide some psychological benefits to users, but also assist in keeping them more engaged and making them more productive.

2.6. Summary

In this chapter, we detailed the history of web browser EEs, after which we systematically gathered and discussed the literature closely related to web browser EEs within the context of performance. We then identified the most prominent literature that forms the theoretical grounding for this study based on three areas of interest, being RIA, JS and WASM.

In the next chapter we will delve into the philosophical stance and research methodology for this study and expand on the details already introduced in chapter 1.

CHAPTER 3

METHODOLOGY

This chapter considers the philosophical stance together with the research methodology used for this study. This chapter also covers the second objective of this study, where the study aims to:

Objective 2

Employ a suitable research methodology by determining which ones are best aligned to deliver on the requirements and goal of this study.

This chapter covers aspects pertinent to the research foundation of this study, across the following areas:

Section 3.1. *Type of research*

Section 3.2. *Philosophical stance*

Section 3.3. *Methodological alignment*

Section 3.4. *Design theory*

Section 3.5. *Research instruments*

Section 3.6. *Data acquisition and evaluation*

Section 3.7. *Ethical considerations*

Section 3.8. *Research limitations*

3.1. Type of research

Expanding on the underpinning theory that was selected for this study in section 1.5., together with insights extracted from the SLR, we can now definitively state that we will be investigating run-time systems, specifically web browser EEs, which is a sub-category of programming language theory (Pierce, 2002; Harper, 2013) and a sibling to virtualisation theory (Gaj et al., 2015; Randal, 2020), which is also a sub-category of programming language theory.

Together with this, we will also propose as to how another sub-category of programming language theory called Foreign Function Interfaces (FFIs) (Grimmer et al., 2018; Rossberg et al., 2018; Pinckney et al., 2020), could be implemented in the proposed solution that would allow for sandboxing of the EE, which is an important security requirement for any web browser EE. These theories constitute the *What* of this study, the *How* is more straight forward and aligned with the *How* of the primary studies uncovered in the SLR.

For the *How* of this study, we previously stated that we intend experimenting with a prototype EE artefact and extracting performance data from it, then comparing that data to the performance data of existing web browser EEs. Given the field within which this study falls, namely computer science, together with the discussed *What* and *How*, we can assert that this study is an applied computer science study, which can be mapped through Kumar's (2018) three viewpoints as depicted in figure 3.1. This mapping shows the primary viewpoints together with the secondary supportive viewpoints for this study.

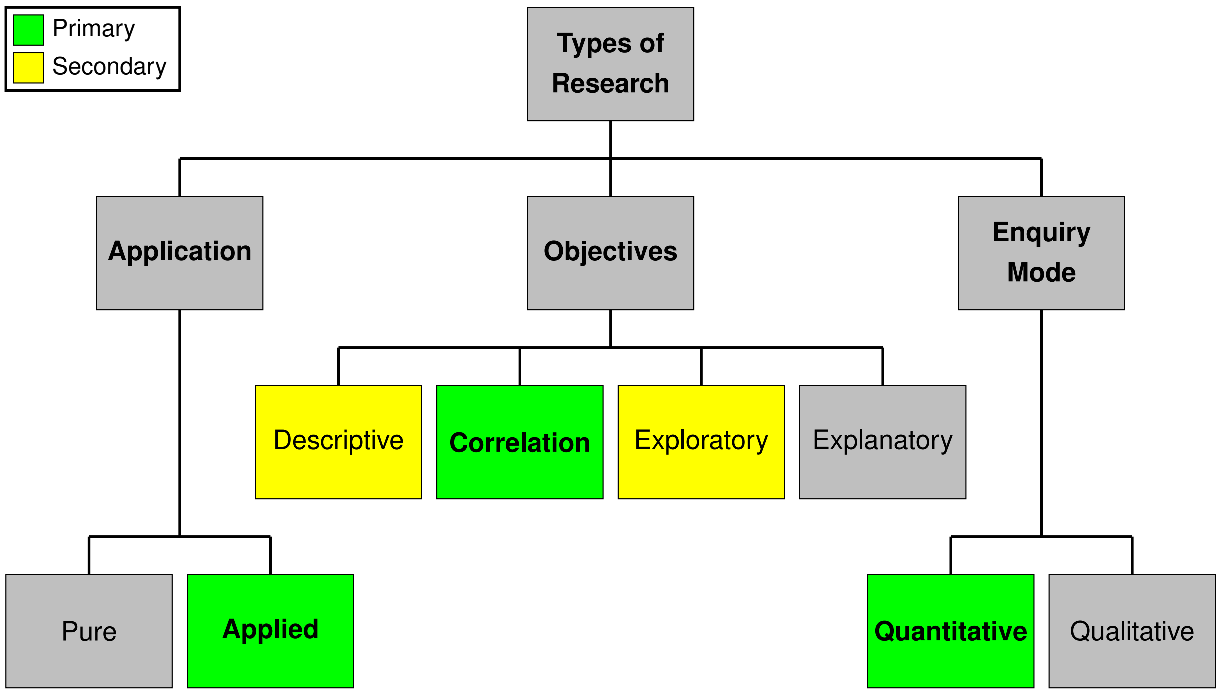

Figure 3.1: Types of Research Viewpoints (Kumar, 2018)

3.1.1. Application

This study pertains to the category of applied research, which aims to address tangible real-world challenges. Applied research refers to a systematic and purposeful investigation that is conducted to address specific, practical problems or issues in the real world. The primary objective of applied research is to generate knowledge, information, or solutions that can be directly used or applied to improve existing processes, products, policies, or practices (Saunders et al., 2019; Sanders et al., 2022).

Key characteristics of applied research include:

1. **Practical Focus**: Applied research is focused on addressing practical issues and finding solutions to real-world problems. It aims to provide tangible benefits and improvements in various domains.
2. **Problem-Solving Orientation**: The research is guided by the need to solve specific problems or challenges. Researchers work to identify practical solutions that can have a positive impact on society, industries, or individuals.
3. **Utilisation of Existing Knowledge**: Applied research often builds upon existing knowledge and theories, using them as a foundation to address specific problems and develop innovative solutions.
4. **Interdisciplinary Approach**: It often involves an interdisciplinary approach, drawing from multiple fields of study to create comprehensive solutions. Collaborations across disciplines are common in applied research.
5. **Application of Technology and Methods**: Technology and various research methods are frequently used to develop and implement new processes, systems, products, or services that address the identified problems.
6. **Action-Oriented and Timely**: Applied research aims for timely and actionable results. It is designed to provide solutions that can be implemented and make a difference in a relatively short period.
7. **Feedback and Adaptation**: Applied research often involves a feedback loop, where findings are evaluated and refined based on the outcomes of practical application. This iterative process helps improve and adapt solutions over time.

The goal is to ensure that the research outcomes have a direct and positive impact on the real world.

3.1.2. Objectives

This study has a correlation research objective, which means that the research focuses on exploring and analysing the relationship between variables to determine the extent to which changes in one variable are associated with changes in another. The objective is to measure the degree and direction of correlation between the variables, which helps in understanding patterns and potential predictive relationships.

For this study, the two variables under observation will be:

1. **EE Overhead** - The overhead imposed by EEs in order for them to execute an application, for example parsing the source application code and compiling it into machine code as in the case of JS code.

2. **EE Performance** - The performance of EEs when executing an application.

Methods used by the objective will include:

1. **Correlation Analysis**: Statistical techniques such as the Pearson Correlation Coefficient, Spearman Rank Correlation Coefficient, or the Kendall Rank Correlation Coefficient may be used to quantify the strength and direction of the relationship between the two variables.
2. **Data Collection**: Data is collected on multiple variables of interest from a sample or population to perform correlation analysis.
3. **Data Interpretation**: The results obtained from correlation analysis are interpreted to understand the nature and strength of the relationship between the two variables.

3.1.3. Enquiry mode

This study's inquiry mode is quantitative, which refers to the approach or method used to collect and analyse numerical or quantifiable data in a study. This is based on the belief that in quantitative research, the focus is on collecting data that can be measured and analysed statistically to identify patterns, relationships, and trends (Bryman & Bell, 2011). The inquiry mode involves systematic and structured data collection methods to gather information from a sample or population.

Some common quantitative inquiry modes or methods are: surveys and questionnaires; experiments; observational studies; structured interviews; secondary data analysis; content analysis; *ex post facto* (after-the-fact or causal-comparative) research; and cross-sectional studies (Creswell & Creswell, 2018; Saunders et al., 2019). Consequently, this study will conduct controlled experiments to manipulate variables and measure the effects on outcomes, allowing for cause-and-effect conclusions.

3.2. Philosophical stance

By definition and according to Bryman & Bell (2011) as well as Bryman (2016), a positivist philosophical position or stance, often associated with the positivist school of thought, is characterised by an empirical and scientific approach to understanding the world. Furthermore, Creswell & Creswell (2018) and Saunders et al. (2019) asserted that the philosophical and epistemological stance of positivism emphasises the importance of empirical evidence, observation, and the scientific method as the primary means to gain knowledge and understanding of the natural and social phenomena.

Bryman & Bell (2011), Bryman (2016), Creswell & Creswell (2018) and Saunders et al. (2019) suggest that the key characteristics of a positivist philosophical stance include:

1. **Empirical Observation and Measurement**: Where positivists believe that knowledge should be based on empirical evidence obtained through sensory experience and direct observation. Measurement and quantification are emphasised to achieve objectivity and rigour in understanding phenomena.
2. **Scientific Method**: Where positivists advocate for the application of the scientific method, which involves systematic observation, experimentation, data collection, analysis, and the formulation of hypotheses and theories. Hypotheses are tested and verified through empirical research to establish scientific laws or principles.
3. **Objectivity and Neutrality**: Where positivism aims for objectivity in research, minimising the influence of personal biases, values, and interpretations. The researcher is seen as an objective observer, separate from the subject of study.
4. **Generalisation and Prediction**: Where positivists seek to generalise research findings to broader populations or situations, aiming for universal laws or principles. The goal is to make predictions based on observed patterns and relationships.
5. **Reductionism**: Where positivism often involves breaking down complex phenomena into simpler, analysable parts to study their individual characteristics and relationships.
6. **Critical Realism**: Where positivism embraces a realist ontology, asserting that there is a reality external to the human mind that can be known and understood through empirical investigation.
7. **Value-Free Science**: Where positivism advocates for separating value judgements and ethical considerations from the scientific inquiry, focusing solely on empirical facts and phenomena.

In light of this, a positivist philosophical stance aligns well with this study in view of the fact that it emphasises the importance of empirical or statistical evidence as well as the scientific method in acquiring knowledge and understanding the world, thereby striving to build a systematic, structured and objective understanding of reality.

3.3. Methodological alignment

Considering that this study will be using DSRs methodology as part of its strategy, in addition to its alignment with Hevner et al.'s (2004) seven proposed guidelines for DSR, DSR and positivism also shares certain foundational principles that allow them to work well together in several ways (Iivari & Venable, 2009):

1. **Empirical Foundation**:

Positivism	DSR
Emphasises the importance of empirical evidence in the pursuit of knowledge.	Grounded in empirical observations and practical experiences, seeking to solve real-world problems through systematic design and evaluation.

2. **Scientific Rigour and Methodology**: Both positivism and DSR uphold the principles of scientific rigour and methodological approach.

Positivism	DSR
Advocates for a rigorous scientific method.	Follows a structured design process, often characterised by iterative cycles of design, implementation, evaluation, and refinement.

3. **Objective and Systematic Approach**:

Positivism	DSR
Seeks objectivity and neutrality in the research process, aiming to minimise biases and subjectivity.	Shares this objective by adopting a systematic approach to design, focusing on the creation of artefacts or solutions guided by well-defined criteria and objectives.

4. **Hypothesis Testing and Validation**:

Positivism	DSR
Emphasises formulating hypotheses and testing them through empirical observations.	Follows a similar approach, where design hypotheses are formulated based on a theoretical foundation, and these hypotheses are then tested and validated through the design and evaluation of artefacts.

5. **Generalisation and Applicability**:

Positivism	DSR
Aims for generalisability of findings to broader populations or contexts.	The aim is to produce artefacts (designs, models, methods) that are not only effective for a specific problem but can also be generalised and applied to similar problems or domains.

6. **Evidence-Based Decision Making**: Both philosophies advocate for evidence-based decision-making.

Positivism	DSR
Relies on empirical evidence to draw conclusions and make informed decisions.	Uses evidence from the design, implementation, and evaluation of artefacts to inform design decisions and improvements.

7. **Quantitative and Qualitative Data Analysis**:

Positivism	DSR
Often employs quantitative data analysis techniques, that align with certain aspects of DSR.	Quantitative evaluation methods are used to assess the effectiveness and efficiency of design artefacts. However, DSR also acknowledges the value of qualitative analysis to understand user experiences and gather feedback.

While DSR aligns with a positivist philosophy in these aspects, it is important to note that DSR also recognises the value of interpretivism and pragmatism, especially in understanding user needs, involving stakeholders, and considering contextual factors during the design process. DSR often incorporates a mix of approaches, including positivist, interpretivist, and pragmatic elements, to create effective and usable design solutions.

3.4. Design theory

The primary objective of design theory is to advance design science by capturing the model of thought which is specific to design. In the field of engineering and more specifically software engineering, the creation of formal design theories is akin to a pursuit of increased generality, abstraction, and rigour. This pursuit follows a variety of paths, one of which gave rise to Concept-Knowledge (C-K) theory which, together with DSR, will be used by this study. DSR and C-K theory are both frameworks used in the field of design and innovation, and they can be complementary in guiding and structuring the design process (Hatchuel et al., 2013).

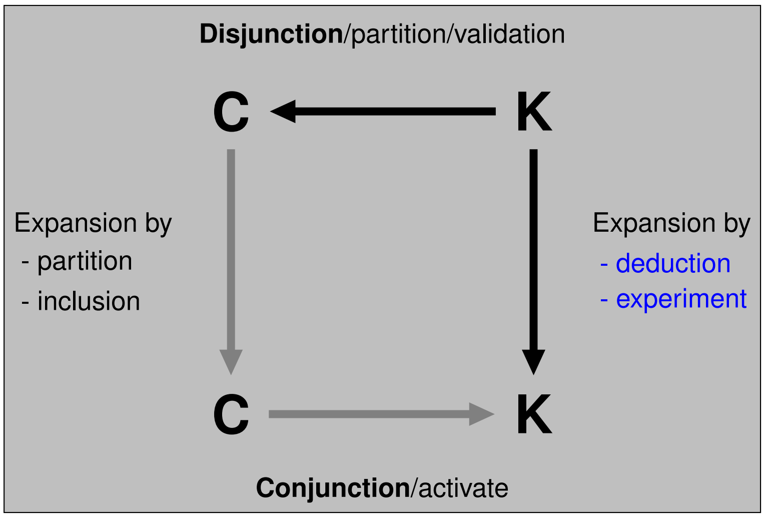

Figure 3.2: C-K Theory Design Square (Hatchuel & Weil, 2003)

As previously elaborated in section 1.5.1., DSR is a methodology used to address complex problems and create innovative solutions. It involves a structured process of creating and evaluating artefacts to solve specific problems. These artefacts could be designs, models, methods, or even software systems. DSR emphasises the importance of rigour, relevance, and design knowledge in the research and development process.

C-K Theory is a theoretical framework used to explain and support the innovation and design process. It focuses on the creation and integration of concepts $\mathcal{C}$ and knowledge $\mathcal{K}$ in the design and innovation activities. The theory distinguishes between existing knowledge $\mathcal{K}$ and the creation of new concepts $\mathcal{C}$, highlighting the importance of exploring the space of possible concepts and their relationships (Hatchuel & Weil, 2003).

As one can see from figure 3.2, C-K theory fits well with this study's research approach of deduction and research strategy of experimentation, as deduction and experimentation are inherently part of its design cycle. Looking deeper, DSR and C-K theory integrate with each other and are complementarity in the following ways:

1. **Idea Generation and Exploration**: C-K theory emphasises the exploration of the concept space $\mathcal{C}$ to generate novel ideas and concepts. DSR can use this theoretical foundation to guide the ideation and concept development phase, ensuring a systematic and structured approach.
2. **Artefact Development and Iteration**: DSR focuses on developing artefacts to solve specific problems. C-K theory can guide the iterative development of artefacts by emphasising the evolution and refinement of concepts $\mathcal{C}$ based on acquired knowledge $\mathcal{K}$ and feedback from evaluations.
3. **Relevance and Rigour**: DSR emphasises the importance of both rigour and relevance in design research. C-K theory, with its emphasis on the creation of meaningful and novel concepts $\mathcal{C}$, aligns with the need for relevance. It also helps structure the research process in a rigorous manner.
4. **Understanding the Conceptual Space**: C-K theory helps in understanding and navigating the conceptual space by distinguishing between existing knowledge $\mathcal{K}$ and the creation of new concepts $\mathcal{C}$. This understanding can guide the design process by allowing researchers to strategically explore and select concepts for artefact creation.

In summary, C-K theory provides a theoretical foundation for understanding the conceptual space and generating novel ideas, while DSR offers a structured methodology to develop and evaluate artefacts based on these ideas. Integrating C-K theory into the DSR process can enhance the ideation and concept development phases, resulting in more innovative and relevant solutions to complex problems.

3.5. Research instruments

Research instruments are tools that are used to collect, measure and analyse data that were collected in relation to the hypothesis, research question and objectives (Bryman &

Bell, 2011; Bryman, 2016). The research instruments that will be used in this study can be divided into three distinct parts, namely the testbed, which is a controlled and real-world like environment, used for benchmarking; the EE artefacts being benchmarked; and the benchmarking algorithms.

3.5.1. Testbed

The testbed is composed of an Intel-based computer hardware running the latest versions of the Fedora[1] variant of the Linux OS. Note that Fedora is the free community version of the commercial RedHat[2] Linux OS. Therefore when a reference is made to RedHat or RedHat documentation, it implies Fedora as well.

The technical details of the testbed are as per Tables 3.1 and 3.2:

Table 3.1: Computer Hardware

Component	**CPU**
Aspect	**Details**
Model	Intel Core i7-7567U @ 3.50 GHz
Frequency	Base 3.50 GHz, Max 4.00 GHz
Architecture	x86_64 (supports 32-bit and 64-bit applications)
Hyperthreading per CPU Core	2
CPU Cores per Socket	2
Sockets	1
Total CPUs	4
BogoMIPS	6,999.82
Byte Order	Little Endian

Component	**GPU**
Aspect	**Details**
Model	Intel Iris Plus Graphics 650
Capacity	up to 32 GiB VRAM

Component	**Memory**
Aspect	**Details**
Capacity	16 GiB RAM
Speed	2,133 MT/s (DDR4 2,400 MT/s)

For benchmarking purposes a sibling pair of CPUs will be isolated, tuned and used as the

[1] https://fedoraproject.org/
[2] https://www.redhat.com/

discrete CPU hardware for the benchmarked EEs. A sibling pair of CPUs refers to the Hyperthreading or Simultaneous Multithreading threads that reside within a single CPU Core. Therefore, a sibling pair of CPUs will share certain resources like the L1 and L2 CPU caches, data lanes and so forth. Similar CPU hardware were also used by the SLR relevant studies of Jangda et al. (2019), Na et al. (2016), Stiévenart et al. (2022), Szewczyk et al. (2022) and Verdú and Pajuelo (2016).

Table 3.2: Computer Software

Component	**OS**
Aspect	**Details**
Type	Fedora Linux
Architecture	x86_64 (64-bit)
Version	39 (released on 7 Nov 2023)
Kernel	6.9.x
Bash (shell)	5.2.x

3.5.2. Benchmarking execution environments

The EEs will consist of two distinct computer software environments that will allow for measuring the existing and proposed solutions performance. For the existing solution, the most recent version 20.12.x of the Node.js[3] JSVM which can also execute WASM-based programming code, will be used. For the proposed solution, the newly engineered SYS23 EE, which executes compiled C-based programming code, will be used.

In order to have the best OS environment within which to benchmark the EEs, it is recommended to use computer hardware that contains a CPU with multiple CPU cores. At minimum it should contain at least two CPU cores per socket, each having at least two hyperthreads per CPU core. Having such a configuration will allow one to isolate one of the CPU cores and dedicate it to the benchmarking activity, without being influenced excessively if at all, by OS noise (de Oliveira et al., 2023), while the remaining CPU core will be used by the OS and the other applications.

The following Linux OS or simply kernel configurations will be set, in order to provide the best low latency OS environment by reducing the OS noise within which the two EEs will be measured:

1. **CPU Isolation**: By default, the kernel's scheduler distributes threads evenly across all available CPUs. To prevent non-application or system threads from affecting one's application threads, one can utilise the `isolcpus` kernel command line option. Load

[3]https://nodejs.org/

balancing for the isolated CPUs are disabled by this option, causing threads to be directed to the non-isolated CPUs by default.

Note that in order for an application thread to use an isolated CPU one must specifically pin the application to the isolated CPU using the `taskset` command. For example, to pin the System23 runtime with a process ID of 999 to CPU 7, one would issue this command:

Shell Command

```
$ taskset -pc 6 999

pid 999's current affinity list:  0-5
pid 999's new affinity list:  6
```

Using `isolcpus` can enhance the performance of applications that are sensitive to jitter or latency. Periodically, the kernel scheduler must interrupt all CPUs to assess if rebalancing or process migration is necessary. CPUs that have been flagged as being isolated avoid those interrupts, thereby reducing the overhead they might otherwise introduce to the applications running on the CPUs. Below is an example of how one would isolate CPUs 7 and 8 by adding it to the kernel command line, using the `grubby` command:

Shell Command

```
$ grubby --update-kernel DEFAULT --args="isolcpus=6,7"
```

To confirm that the kernel command line was indeed updated, use:

Shell Command

```
$ grubby --info DEFAULT | grep args

args="...  ro rhgb quiet isolcpus=6,7"
```

To verify that the requested CPUs were indeed isolated, one can use this command after rebooting:

Shell Command

```
$ cat /sys/devices/system/cpu/isolated

6-7
```

One can also see which CPUs were not isolated using this command:

Shell Command

```
$ cat /sys/devices/system/cpu/possible

0-5
```

Even when `isolcpus` is used, several system threads will still be allocated to the isolated CPUs, by the kernel. One can however still move some of these system threads to the non-isolated CPUs. The following command will move all system threads, namely threads with a parent process ID of 2 to CPUs 1 through 6:

Shell Command

```
$ pgrep -P 2 | xargs -i taskset -pc 0-5 {}

pid 3's current affinity list:  0-7
pid 3's new affinity list:  0-5

pid 4's current affinity list:  0-7
pid 4's new affinity list:  0-5

...
```

Alternatively one can also use the `tuna` command to move all system threads away from CPUs 7 and 8:

Shell Command

```
$ tuna isolate --cpus=6,7
```

To verify system thread migrations, one can display the thread or CPU affinities of all threads, by utilising the `tuna` command, where 0xff means that the thread can execute on any CPU and an integer value means that it is pinned to that specific CPU. In the below example, process with ID 13 is pinned to CPU 4 and process with ID 17 is pinned to CPU 1:

Shell Command

```
$ tuna show_threads

                      thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary              cmd
1       OTHER      0     0xff     18027         3219         systemd
2       OTHER      0     0xff      6750           29        kthreadd
3       OTHER      0     0xff        11            0 pool_workqueue_release
4       OTHER      0     0xff         2            0 kworker/R-rcu_g
5       OTHER      0     0xff         2            0 kworker/R-rcu_p
6       OTHER      0     0xff         2            0 kworker/R-slub_
7       OTHER      0     0xff         2            0 kworker/R-netns
13      OTHER      0        3         6            1 kworker/R-mm_pe
14      OTHER      0     0xff        51            3 rcu_tasks_kthread
15      OTHER      0     0xff        14            0 rcu_tasks_rude_kthread
16      OTHER      0     0xff         7            0 rcu_tasks_trace_kthread
17      OTHER      0        0   1158153         3390     ksoftirqd/0
...
```

Additionally, one should move all kernel workqueues to the CPUs that have not been isolated. For instance, to allocate all workqueues to CPUs 1 through 6, use a bit mask of 0x3f, where the default is 0xff:

Shell Command

```
$ find /sys/devices/virtual/workqueue -name cpumask -exec echo 3f > {} \;
```

One can list the current kernel workqueue affinities, to confirm that they have moved:

Shell Command

```
$ find /sys/devices/virtual/workqueue -name cpumask -print -exec cat {}
\;

/sys/devices/virtual/workqueue/cpumask
3f
/sys/devices/virtual/workqueue/writeback/cpumask
3f
```

To verify that the CPUs were successfully isolated, one can check the number of thread context switches occurring on each CPU:

Shell Command

```
$ perf stat -e 'sched:sched_switch' -a -A --timeout 30000

 Performance counter stats for 'system wide':

CPU0                149,090      sched:sched_switch
CPU1                118,949      sched:sched_switch
CPU2                155,864      sched:sched_switch
CPU3                124,657      sched:sched_switch
CPU4                288,309      sched:sched_switch
CPU5                517,772      sched:sched_switch
CPU6                  1,077      sched:sched_switch
CPU7                  2,772      sched:sched_switch

       30.036827225 seconds time elapsed
```

A very low context switch count should be shown for the isolated CPUs.

2. **Timer Ticks**: The kernel's scheduler operates at regular intervals on individual cores to manage the transition between active threads, these are called timer ticks. These timer ticks can lead to unpredictable delays in applications sensitive to latency. If you have isolated specific cores to your application, each running a single thread, you can mitigate against these interruptions by employing the kernel command line option called `nohz_full`, to reduce the number of timer tick interrupts.

   For example, if one has already isolated CPUs 7 and 8 using `isolcpus`, then one can use this command to reduce the interrupt timer ticks on the same CPUs:

   Shell Command

   ```
   $ grubby --update-kernel DEFAULT --args="nohz_full=6,7"
   ```

   To confirm that the kernel command line was indeed updated, use:

   Shell Command

   ```
   $ grubby --info DEFAULT | grep args

   args="...  ro rhgb quiet nohz_full=6,7"
   ```

   To verify what boot time kernel parameters were set do the following after rebooting:

Shell Command

```
$ cat /proc/cmdline

BOOT_IMAGE=(hd0,msdos1)/...  ro rhgb quiet nohz_full=6,7
```

One can then use this command to verify that the timer tick frequency has indeed been reduced by using the `perf` command:

Shell Command

```
$ perf stat -e 'irq_vectors:local_timer_entry' -a -A --timeout 30000

 Performance counter stats for 'system wide':

CPU0              27,074      irq_vectors:local_timer_entry
CPU1              25,419      irq_vectors:local_timer_entry
CPU2              27,279      irq_vectors:local_timer_entry
CPU3              25,923      irq_vectors:local_timer_entry
CPU4              26,061      irq_vectors:local_timer_entry
CPU5              25,390      irq_vectors:local_timer_entry
CPU6                   2      irq_vectors:local_timer_entry
CPU7                   3      irq_vectors:local_timer_entry

       30.077830680 seconds time elapsed
```

Note that the timer tick cannot be completely eliminated.

3. **CPU Frequency Scaling**: Maximising the CPU frequency will assist in negating the need for the CPU to increase or decrease its frequency as the workload increases or decreases. As the frequency scaling is uncontrollable by a user, it is advisable to have a stable and uniform frequency so as to avoid any unpredictable speedups or slowdowns by the CPU, which could lead to erratic results, even when running the exact same task over and over.

   This command will show one how long it takes a CPU to switch frequency, which can be a considerable amount of time:

Shell Command

```
$ cpupower frequency-info | grep "transition latency"

maximum transition latency:  20.0 µs
```

Use this command to ensure that one is able to manipulate the CPU core frequency, this should list all of the available scaling governors:

Shell Command

```
$ cpupower frequency-info | grep governors

available cpufreq governors:  conservative ondemand userspace powersave
performance schedutil
```

Thereafter, use this command to list all of the available CPU frequency scaling governors, whereby there should be a policy for each CPU:

Shell Command

```
$ find /sys/devices/system/cpu -name scaling_governor

/sys/devices/system/cpu/cpufreq/policy6/scaling_governor
/sys/devices/system/cpu/cpufreq/policy4/scaling_governor
/sys/devices/system/cpu/cpufreq/policy2/scaling_governor
/sys/devices/system/cpu/cpufreq/policy0/scaling_governor
/sys/devices/system/cpu/cpufreq/policy7/scaling_governor
/sys/devices/system/cpu/cpufreq/policy5/scaling_governor
/sys/devices/system/cpu/cpufreq/policy3/scaling_governor
/sys/devices/system/cpu/cpufreq/policy1/scaling_governor
```

One can then set the scaling governor for the desired CPU to the maximum frequency, which is also known as the performance level frequency. For example to set the maximum frequency for CPU 8, one would use this command:

Shell Command

```
$ echo performance > /sys/devices/system/cpu/cpufreq/policy7/scaling_governor
```

4. **Interrupt Affinity**: Latency spikes from interrupt request (IRQ) processing can be minimised by adjusting the CPU affinity to IRQs. This can be done using the `irqbalance` command, which will also automatically isolate any CPU cores that were specified as part of the kernel command line parameter `isolcpus`. Execute this command to isolate the CPU cores previously specified by `isolcpus`:

Shell Command

```
$ irqbalance --foreground --oneshot
```

One can check the CPU affinity for all IRQs by using this command:

Shell Command

```
$ find /proc/irq/ -name smp_affinity_list -print -exec cat {} \;

/proc/irq/0/smp_affinity_list
0-5
/proc/irq/1/smp_affinity_list
0-5
...
```

One can also monitor any isolated CPU, to ensure that they are not receiving any IRQs, here CPU 8 has been isolated and should be processing zero IRQs:

Shell Command

```
$ watch cat /proc/interrupts

Every 2.0s: cat /proc/interrupts      example.com: Wed Jul 31 20:57:02 2024

            CPU0         ...        CPU7
  0:          42         ...           0    IO-APIC    2-edge      timer
  1:           0         ...           0    IO-APIC    1-edge      i8042
  8:           1         ...           0    IO-APIC    8-edge      rtc0
  9:           0         ...           0    IO-APIC    9-fasteoi   acpi
  ...
```

5. **Disable Swap**: A significant page fault and latency spike will be incurred when attempting to retrieve data that has been swapped from microchip-based memory to disk-based memory, as disk-base memory access is far slower than microchip-based memory access. One can prevent this by disabling disk-based memory swapping:

Shell Command

```
$ swapoff -a
```

Note that sometimes one may find that the disk-based memory swapping automatically re-enables itself because of some builtin OS safeguards. To prevent that from happening, one can use the `systemctl` command to shutdown the disk-based memory swapping, including any safeguards. Firstly one has to find all active swap units.

Shell Command

```
$ systemctl --type swap

  UNIT                 LOAD   ACTIVE SUB    DESCRIPTION
-----------------------------------------------------------------------
dev-mapper-luks.swap loaded active active /dev/mapper/luks
dev-zram0.swap       loaded active active Compressed Swap on /dev/zram0
...
```

Thereafter one has to stop each swap unit.

Shell Command

```
$ systemctl stop 'dev-mapper-luks.swap'
```

Finally masking each unit, where this masking is what prevents it from automatically restarting.

Shell Command

```
$ systemctl mask 'dev-mapper-luks.swap'

Created symlink /etc/systemd/system/dev-mapper-luks.swap → /dev/null.
```

One can then verify that we are no longer swapping to disk by displaying the current swap information:

Shell Command

```
$ swapon --show
```

6. **Disable Transparent Huge Pages**: Transparent Huge Pages is a feature of the kernel, whereby it automatically promotes regular sized memory pages into huge pages. This process can lead to latency spikes both during the promotion of memory pages and when memory compaction occurs. The support of Transparent Huge Page can be disabled by executing the following command:

Shell Command

```
$ echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

7. **Disable NUMA Memory Balancing**: Automatic migration of memory pages between Non-Uniform Memory Access (NUMA) nodes will cause memory page faults and latency

spikes for applications using the affected memory. Automatic NUMA memory balancing can be disabled by disabling the `numad` service as well as executing the following command:

Shell Command

```
$ echo 0 > /proc/sys/kernel/numa_balancing
```

8. **Disable Mitigations for CPU Vulnerabilities**: Mitigations against CPU vulnerabilities should be turned off, as those mitigations can have an unknown negative affect on certain types of application processing. It is entirely possible that a certain mitigation can effect an application executing in one EE while having no effect on a similar application executing in another EE.

   CPU mitigations against vulnerabilities typically reduce the CPUs performance in the following ways:

   (a) **Speculative Execution Mitigations**: Speculative execution allows CPUs to run faster by guessing ahead, limiting or disabling it reduces these performance gains, leading to slower execution of instructions.

   (b) **Increased Context Switching**: The overhead of additional context switches slows down CPU performance, especially in workloads that involve frequent transitions between user and kernel modes.

   (c) **Memory Fencing and Cache Management**: Additional memory fencing and cache management overhead increases latency, as it interrupts normal operations and forces the CPU to reload data from slower memory sources.

   (d) **Disabling Hyperthreading/Simultaneous Multithreading**: Disabling Hyperthreading/Simultaneous Multithreading effectively reduces the number of simultaneous execution threads, meaning the CPU can handle fewer tasks in parallel, leading to lower overall performance.

   (e) **Increased Synchronisation and Locking**: Increase the need for context switches, synchronisation, and cache invalidation. More synchronisation means less concurrency, which can bottleneck performance, especially in multi-core or multi-threaded applications.

   Therefore, in order to level the playing field for all applications and EEs it is best to entirely disable these mitigations (Szewczyk et al., 2022). To begin, one can list all of CPU mitigations that have been implemented and in turn these mitigations can be disabled:

Shell Command

```
$ lscpu

...

NUMA:
  NUMA node(s):           1
  NUMA node0 CPU(s):      0-7
Vulnerabilities:
  Gather data sampling:  Not affected
  Itlb multihit:         KVM: Mitigation: VMX disabled
  L1tf:                  Mitigation; PTE Inversion; VMX conditional cache
                         flushes, SMT vulnerable
  Mds:                   Mitigation; Clear CPU buffers; SMT vulnerable
  Meltdown:              Mitigation; PTI
  Mmio stale data:       Unknown: No mitigations
  Retbleed:              Not affected
  Spec rstack overflow:  Not affected
  Spec store bypass:     Mitigation; Speculative Store Bypass disabled via
                         prctl
  Spectre v1:            Mitigation; usercopy/swapgs barriers and __user
                         pointer sanitization
  Spectre v2:            Mitigation; Retpolines, IBPB conditional, IBRS_FW,
                         STIBP conditional, RSB filling, PBRSB-eIBRS Not
                         affected
  Srbds:                 Not affected
  Tsx async abort:       Not affected
```

At boot time one can disable all mitigation as follows:

Shell Command

```
$ grubby --update-kernel DEFAULT --args="mitigations=off"
```

As before, to confirm that the kernel command line was indeed updated, use:

Shell Command

```
$ grubby --info DEFAULT | grep args

args="...  ro rhgb quiet mitigations=off"
```

As before, one can verify what boot time kernel parameters were set using the following, after rebooting:

Shell Command

```
$ cat /proc/cmdline

BOOT_IMAGE=(hd0,msdos1)/...  ro rhgb quiet mitigations=off
```

Listing the CPU mitigations again will now show that no mitigations are enabled and that the CPU is vulnerable to exploits:

Shell Command

```
$ lscpu

...

NUMA:
  NUMA node(s):            1
  NUMA node0 CPU(s):       0-7
Vulnerabilities:
  Gather data sampling:    Not affected
  Itlb multihit:           KVM: Mitigation: VMX disabled
  L1tf:                    Mitigation; PTE Inversion; VMX vulnerable
  Mds:                     Vulnerable; SMT vulnerable
  Meltdown:                Vulnerable
  Mmio stale data:         Unknown: No mitigations
  Retbleed:                Not affected
  Spec rstack overflow:    Not affected
  Spec store bypass:       Vulnerable
  Spectre v1:              Vulnerable: __user pointer sanitization and
                           usercopy barriers only; no swapgs barriers
  Spectre v2:              Vulnerable, IBPB: disabled, STIBP: disabled,
                           PBRSB-eIBRS: Not affected
  Srbds:                   Not affected
  Tsx async abort:         Not affected
```

In order to measure how effective each OS configuration change is in mitigating against OS noise, one should use an OS profiling tool such as the *Linux Kernel OS Noise Tracer*[4]. This will provide one with an indication of the source of any OS noise as well as performance metrics that can be used to best tune the OS for benchmarking purposes.

The relevance and importance of these OS level configurations in mitigating against OS noise is one aspect that has been found to be missing from the vast majority of SLR relevant studies, except for the study by Szewczyk et al. (2022). This aspect is so crucial that if not mitigated against, it can dramatically change the results of the measured performance (de Oliveira et al., 2023).

[4]https://docs.kernel.org/trace/osnoise-tracer.html

3.5.3. Benchmarking algorithms

Researchers depend on empirical evidence, but quantifying the precise advantages of a solution analytically is often challenging. In collaborative research areas such as computer science, it is advantageous for the community to adopt a shared set of benchmarks to evaluate their work. These benchmark suites improve the reproducibility of experiments, enabling more consistent comparisons of results across various studies (Yuki, 2014).

Table 3.3: PolyBench/C v4.2.1 Benchmarks

Benchmark	Description
2mm	2 Matrix Multiplications $(\alpha * A * B * C + \beta * D)$
3mm	3 Matrix Multiplications $((A * B) * (C * D))$
adi	Alternating Direction Implicit solver
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
cholesky	Cholesky Decomposition
correlation	Correlation Computation
covariance	Covariance Computation
deriche	Edge detection filter
doitgen	Multi-resolution analysis kernel (MADNESS)
durbin	Toeplitz system solver
fdtd-2d	2-D Finite Different Time Domain Kernel
floyd-warshall	Vertice pairs shortest path in a weighted graph
gemm	Matrix-multiply C=alpha.A.B+beta.C
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
gramschmidt	Gram-Schmidt decomposition
heat-3d	Heat equation over 3D data domain
jacobi-1d	1-D Jacobi stencil computation
jacobi-2d	2-D Jacobi stencil computation
lu	LU decomposition
ludcmp	LU decomposition followed by Forward Substitution
mvt	Matrix Vector Product and Transpose
nussinov	Dynamic programming algorithm for sequence alignment
seidel-2d	2-D Seidel stencil computation
symm	Symmetric matrix-multiply
syr2k	Symmetric rank-2k update
syrk	Symmetric rank-k update
trisolv	Triangular solver
trmm	Triangular matrix-multiply

PolyBench/C which uses the polyhedral model, is an example of such a benchmark suite that many studies now utilise (Jangda et al., 2019; Ménétrey et al., 2021; Na et al., 2016; Nießen et al., 2020; Rossberg et al., 2018; Salim et al., 2020; Spies & Mock, 2021; Szewczyk et al., 2022; Titzer, 2022; Wang, 2021; Wang, 2022; Yan et al., 2021), significantly aiding in establishing a common foundation for empirical validations. The benchmarking algorithms that will be used by this study are the PolyBench/C algorithms as per Table 3.3, that were used by other related studies as per the SLR.

The polyhedral model is a versatile representation for arbitrarily nested loops that can be algebraically represented in the Static Control Parts (SCoP) format. The SCoP format contains four elements for each statement, namely, iteration domains, access relations, dependence polyhedra/relations and the program schedule. Originally, all array subscripts, loop bounds, and branch conditions needed to be affine functions of loop index variables and global parameters for analysis. Ongoing research has however significantly broadened the range of analysable programs by polyhedral frameworks.

In addition, the PolyBench/C algorithms implement the Performance Application Programming Interface (PAPI)[5], which allows for in-depth performance benchmark reporting (Barry et al., 2023). PAPI supplies a standardised interface and methodology for performance counter data collection from diverse hardware and software components. These include hardware components such as CPUs, Graphics Processing Units (GPUs) and more, as well as virtual and cloud environments from most major technology manufacturers.

This industry-wide collaboration ensures seamless integration of PAPI with new architectures as they are released. As the popularity of the PAPI component architecture increases, performance tools that interface with PAPI will automatically gain the capability to measure these new data sources. From a software engineering and research point of view, PAPI allows one to extract the relationship between the performance of hardware processing and software performance.

3.6. Data acquisition and evaluation

Selecting the best sampling technique for measuring computer software performance depends on various factors, including the specific goals of one's performance analysis, the nature of the software, and the available resources (Pace, 2021). Some commonly used sampling techniques for computer software performance analysis are:

1. **Time-Based Sampling**: Collect performance metrics at regular time intervals, such as every millisecond, second, minute, and so on. This technique provides a consistent view of software performance over time and is useful for identifying trends and patterns.

[5]https://icl.utk.edu/papi/

2. **Event-Based Sampling**: Capture performance data based on specific events or triggers within the software, such as system calls, exceptions, or user interactions. This technique is valuable for understanding performance during critical events or scenarios.

3. **Transaction-Based Sampling**: Capture performance metrics for each transaction or user interaction with the software. This technique is particularly useful for systems with high transaction volumes, such as web applications.

4. **Percentage-Based Sampling**: Sample a certain percentage of transactions or requests for performance measurement. For example, you could sample 10% of requests, providing an overview of the system's performance while reducing the data collection overhead.

5. **Adaptive Sampling**: Dynamically adjust the sampling rate based on system conditions, workload, or performance characteristics. For instance, increase the sampling rate during peak usage periods to capture more data during high-load scenarios.

6. **Stratified Sampling**: Divide the software into different strata, such as modules, components or user types and collect performance metrics independently for each stratum. This allows for a focused analysis of specific areas within the software.

7. **Critical Path Sampling**: Focus on sampling the critical paths or most performance-sensitive sections of the software. This approach helps identify bottlenecks and areas that significantly impact overall performance.

8. **Random Sampling**: Select samples randomly from the software's execution. This technique can help avoid bias and ensure a representative sample of the software's performance.

9. **User-Based Sampling**: Sample performance based on specific user profiles or personas to understand how different types of users experience the software. This is especially useful for applications with diverse user interactions and role-based access restrictions.

10. **Load-Based Sampling**: Adjust the sampling rate based on the current system load or traffic. Higher sampling rates during heavy load periods can provide insights into performance under stress.

The choice of the sampling technique will depend on factors such as the criticality of the application, the nature of performance requirements, the workload characteristics, and the available tools and resources. It is often beneficial to combine multiple sampling techniques to gain a comprehensive understanding of software performance. Experimentation and adaptation based on the specific context are key to finding the most effective sampling approach for a particular software system (Baltes & Ralph, 2022).

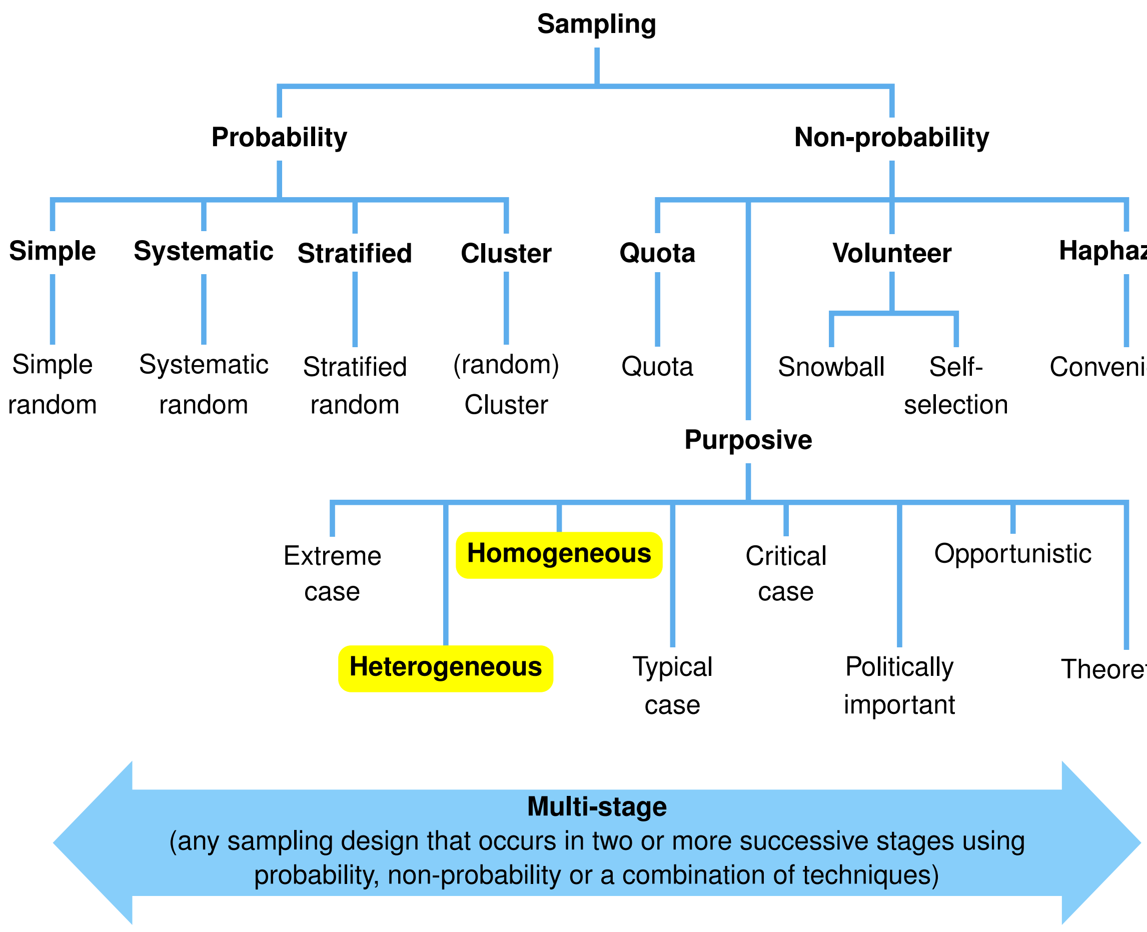

Figure 3.3: Sampling Techniques (Saunders et al., 2019)

The data acquisition and evaluation that will be employed by this study have been formulated and partially based on what was discovered through the SLR. The primary as well as the other studies identified through the SLR have mostly employed non-probability purposive sampling, by utilising homogeneous sampling involving a small set of specific compute-intensive workloads. Having a small set of specific compute-intensive workloads from which to sample benchmarking data from is considered both rational and logical.

In non-probability sampling methods, excluding quota sampling, determining an appropriate sample size is ambiguous. There are no set rules, unlike in probability sampling. Instead, it is crucial and foundationally critical to establish a logical link between your chosen sampling approach and the objective and scope of your research. The selected sample can serve specific purposes, such as illustrating a specific aspect or generalising findings to theories, rather than representing a whole population (Saunders et al., 2019).

These samples are available in C and are software applications that are based on compute-intensive algorithms as listed in the previous section. These samples are also easily converted into JS and WASM to allow for uniform benchmarking across the three EEs. Having uniformly implemented algorithms to benchmark across the three EEs is considered fundamental to the validity of the benchmarking data collected and the overall validity of this study.

In addition to the homogeneous sampling, heterogeneous sampling aspects will also be added in to allow for a multi-stage type sampling regime as suggested by Saunders et al.'s (2019) in figure 3.3. Given that, it would be impossible to collect benchmarking data from every possible compute-intensive algorithm ever invented, having a multi-stage sampling regime will at the very least allow for a wider benchmarking view than with some previous studies.

3.7. Ethical considerations

In addition to Myers and Venable's (2014) six ethical principles which this study intends to adhere to, there are also other ethical considerations for non-participant based applied experimental computer science studies, where researchers do not directly involve human subjects but conduct experiments on computer systems or software. These ethical considerations are critical in ensuring responsible research and technology development. They include:

1. **Data Privacy and Security**: Ensure that sensitive data used in experiments, if any, is handled and stored securely, following best practices for data privacy and security to prevent unauthorised access or breaches.
2. **Responsible Data Usage**: Use data ethically and responsibly, obtaining appropriate permissions and ensuring that data is used for the intended purpose without causing harm or violating privacy.
3. **Avoidance of Bias and Fairness**: Check for biases in data, algorithms, or models that may lead to unfair outcomes, discrimination, or perpetuation of inequalities. Strive for fairness and inclusivity in system behaviour.
4. **Transparency and Reproducibility**: Promote transparency in reporting methodologies, assumptions, and results to facilitate the replication and validation of experiments by the wider research community.
5. **Appropriate Use of Artificial Intelligence (AI)**: Apply AI and machine learning responsibly, ensuring that systems are explainable, accountable, and not used to propagate misinformation, harm, or malicious intent.
6. **Environmental Impact**: Consider and mitigate the environmental impact of experiments, algorithms, or technologies, striving for energy-efficient and sustainable solutions.
7. **Compliance with Laws and Regulations**: Adhere to applicable laws, regulations, and guidelines governing the use of technology, ensuring compliance with ethical standards set by relevant authorities.
8. **Avoidance of Harm**: Design and conduct experiments in a way that minimises the risk of harm to individuals, communities, or the environment. Prioritise safety and well-being.

9. **Conflict of Interest Disclosure**: Disclose any potential conflicts of interest that could influence the design, conduct, or reporting of the experiment, ensuring transparency in relationships with stakeholders.
10. **Ethical Use of Algorithms**: Ensure that algorithms developed or used in experiments are designed with ethical considerations, respecting human rights, and avoiding potential harm or discrimination.
11. **Accountability and Responsibility**: Take responsibility for the outcomes and implications of the experiments, communicating results accurately and avoiding exaggeration or misrepresentation.
12. **Informed Decision Making**: Provide clear information to stakeholders and decision-makers regarding the capabilities, limitations, and potential risks associated with the experimental technology.
13. **Oversight and Review**: Seek ethical oversight and review of research proposals and experiments by internal or external ethics committees to ensure compliance with ethical guidelines and standards.

Adhering to these ethical considerations will help ensure that this study is conducted in a responsible and beneficial manner, contributing positively to society and the advancement of technology.

3.8. Research limitations

This study may encounter several common limitations that can affect the validity, generalisability, and robustness of its findings. Some common research limitations that we are aware of and may strive to mitigate are:

1. **Sample Size and Representativeness**: Limited sample size can affect the generalisability of results to a broader population. If the sample is not representative of the target user base or system users, the findings may not be applicable in real-world scenarios.
2. **Sampling Bias**: Bias in the selection of participants or systems can skew the results and compromise the validity and reliability of the study. Biased sampling may not accurately reflect the diversity of the population or system usage patterns.
3. **Generalisability of Findings**: The controlled settings of experiments may limit the real-world applicability and generalisability of the findings. Extrapolating results to different contexts or scenarios should be done cautiously.
4. **Laboratory vs. Real-World Conditions**: Experiments conducted in a controlled laboratory settings may not accurately mimic real-world conditions, potentially affecting the relevance and applicability of the results in practical settings.

5. **Operationalisation and Measurement**: Defining and measuring variables in a precise and consistent manner can be challenging. Ambiguities or inconsistencies in the operationalisation of variables may impact the reliability and validity of the measurements.
6. **Limited Time Frame**: Constraints on time may limit the depth and breadth of the study. Long-term effects, user learning curves, or system performance over extended periods may not be adequately captured.
7. **Resource Constraints**: Limited resources, including budget, technology, or personnel, can constrain the scale and scope of the study, impacting the comprehensiveness and robustness of the research.
8. **Uncontrolled Variables**: Unforeseen variables or confounding factors that are not controlled can introduce noise into the study, making it challenging to isolate the effects of specific variables of interest.
9. **Ethical Limitations**: Ethical constraints, such as limitations on data collection, privacy concerns, or restrictions on human experimentation, can affect the design and execution of experiments.
10. **Technological Limitations**: The technological limitations of existing tools, frameworks, or platforms may restrict the complexity and realism of the experimental setup, thereby affecting the accuracy of results.
11. **Simplicity of Models**: Simplified models or assumptions may be necessary to make the experiment tractable, but they can oversimplify real-world complexities, potentially limiting the accuracy of predictions or insights.
12. **Publication Bias**: Studies with statistically significant or positive results are more likely to be published, leading to a bias in the available literature and potentially skewing the overall understanding of a particular phenomenon.

Acknowledging and addressing these limitations in research designs, methodologies, and interpretations are crucial for improving the rigour, relevance, and applicability of applied experimental computer science studies. Researchers should transparently communicate these limitations to provide context for the findings and guide future research directions.

3.9. Summary

In this chapter, we provided the philosophical stance together with the research methodology that will be used for this study. We then looked at the design theory and discussed how this study aligns with DSR. We also then suggest a set of existing and previously used research instruments, that were used with similar studies as outlined in the SLR. As outlined in the SLR it was found that the PolyBench/C benchmarking suite formed an integral part of those study's research instruments, and as such its use in our study was both prudent and expected.

Further to that, we also briefly noted the ethical implications of our study and what was required in order for this study to uphold a high standard of ethics. We also then reviewed the limitations of this study in detail, so that any conclusions derived at, can be fully understood within the context within which they were attained. Finally, we outlined how data will be acquired through experiments and then evaluated so that we can derive sound conclusions which will be presented in chapter 5.

In the next chapter, we will present a system architecture and design for a proposed solution, which re-evaluates and builds on what was presented in chapter 1.

CHAPTER 4

SYSTEM ARCHITECTURE AND DESIGN

This chapter examines ways to improve the performance of web browser EEs through the creation of a new web browser EE, better suited to current user performance demands. This chapter also covers this research's objective three, where the researcher endeavours to:

Objective 3

Develop a prototype web browser EE that is capable of hosting and executing CIAs with native desktop performance.

This chapter explores aspects of the new web browser EE based on the following:

Section 4.1. *Design considerations*

Section 4.2. *Conceptual design*

Section 4.3. *Prototype architecture*

Section 4.4. *Data collection*

Section 4.5. *Data precision*

Section 4.6. *Evaluation*

4.1. Design considerations

Before we delve into the actual web browser EE prototype architecture and design, let us first ponder what key characteristics such an ideal prototype would be comprised of in a perfect world. Three distinct characteristics come to mind, first there is the ability to receive and execute high-performance applications, second is the ability to employ a high-performance runtime environment, and the third is to employ a Zero Trust Architecture (ZTA) so that in the event of executing a malicious application, it is unable to harm the device within which it is executing (Rose et al., 2020).

To further understand these three characteristics and to form a high-level view of each, we will now dissect each characteristic.

4.1.1. Reinventing the wheel

The ability to receive and execute high-performance applications is synonymous with natively installing software on a computer system. Harnessing that ability through a web browser by simply browsing to a specific web page, which would trigger a natively compatible application from downloading on one's computer system, would thus be ideal. Instead of trying to create a universally compatible computer language that is capable of being executed on every existing computer system.

One would thus avoid *reinventing the wheel* by creating a solution to a problem that has already been solved, typically with existing, well-established methods or technologies. By using deductive reasoning, one can formulate several reasons as to why avoiding the reinvention of the wheel is beneficial:

1. **Efficiency**
   (a) *Time Savings*: Given that existing solutions have already been designed, developed and field tested, it stands to reason that reusing existing solutions can save a considerable amount of time in comparison to creating a completely new solution. This allows one to instead focus on adding unique aspects in support of a web browser EE.
   (b) *Resource Optimisation*: Given also then that valuable resources are not used for developing a new solution, it reduces the need for duplicating efforts. Thereby allowing resources to be used more effectively elsewhere, for example to enhance existing solutions so as to improve the security robustness of the proposed web browser EE.
2. **Quality and Reliability**
   (a) *Proven Solutions*: Existing solutions have typically over time been repeatedly tested, analysed extensively in various forms, and actively used as intended. That mature nature of existing solutions gives rise to them being viewed as well-established solutions with proven reliability and track record as compared to new solutions that are still immature and unproven.
   (b) *Reduced Bugs*: Furthermore, given the extensive repeated testing, analysis and everyday use of well-established solutions, it can result in fewer design defects and bugs as compared to developing a new solution from scratch, which still needs to advance through several cycles of testing and analysis, together with extensive everyday use.
3. **Cost Savings**
   (a) *Development Costs*: Developing a new solution can be expensive in terms of both time and money, as it stands to reason that any new development would require some margin of time to complete, as well as some amount of funding to cover the development effort and required resources. Whereas using an existing solution, one

would be able to leapfrog those initial time and cost constraints, since they have already been consumed.

   (b) *Maintenance Costs*: Maintaining custom solutions can also be costly compared to using widely adopted tools and libraries with community support, since custom solutions tend to be niche, proprietary and not extensively supported or used. Instead, adopting widely used tools and libraries with community support, tend to cost much less, given the extensive support nature of communities, such as the open-source community.

4. **Focus on Innovation**

   (a) *Unique Value*: As stated before, by reusing existing solutions for common problems, developers can concentrate on creating unique features and innovations that add real value to the proposed web browser EE, since their focus is not on creating a new solution from scratch.

   (b) *Competitive Advantage*: In turn, developers can also focus on creating differentiators rather than rebuilding basic foundational components. This would aid in quickly realising truly innovative features for the proposed web browser EE, instead of delivering basic foundational components that does not help to advance the widespread usage of a new web browser EE.

5. **Standardisation**

   (a) *Interoperability*: Using standard well-established solutions ensures better compatibility and integration with other systems and tools, given they have been utilised extensively and have matured to such a degree that many layers of compatibility, together with many integrations, have been incorporated into them.

   (b) *Community Support*: Well-established solutions often have strong community support, including documentation, forums, and updates, especially those that are made available as open-source solutions. In fact, in some instances open-source solutions are considered to be better than some closed-source or proprietary solutions, given that they are not constrained by development resource limitations that are driven by profit.

6. **Learning and Growth**

   (a) *Knowledge Sharing*: Leveraging existing solutions allows developers to learn from the collective knowledge and experience of the community. Whereas, if one were to create a new solution then all developers would be equally novice and have very little knowledge of the solution, given its infancy.

   (b) *Best Practices*: Using standard existing solutions helps in adopting best practices based on their usage patterns, that have been refined over time. Comparatively, when creating a new solution that has yet to present established usage patterns from which to formulate any set of best practices, may take some time to realise.

7. **Risk Reduction**

   (a) *Reduced Risk*: Using a known, reliable and well-established solution mitigates the risks associated with developing and deploying untested newly developed solutions. This common understanding holds true, where maturity is commonly proportional to robustness of well-established solutions.
   (b) *Predictable Outcomes*: Established solutions provide more predictable results, making further development planning and execution smoother as compared to newly developed solutions that still need to formulate a baseline from which to establish a predictable pattern planning and execution outcomes.

Considering this, having a web browser EE that is capable of executing tried and tested natively compatible applications should be considered the best solution with which one would be able to fully take advantage of the benefits of CIAs.

4.1.2. Compilation point in time

AOT compilation and JIT compilation are two distinct methods used to translate source code into machine code that a computer system can execute (Bourgoin & Chailloux, 2015; Haßler & Maier, 2021; Herman et al., 2014; McAnlis et al., 2014a; McAnlis et al., 2014b; Ménétrey et al., 2021; Nießen et al., 2020; Park et al., 2017; Park et al., 2018; Salim et al., 2020; Serrano, 2018; Serrano, 2021; Šipek et al., 2019; Szewczyk et al., 2022; Titzer, 2022; Van Es et al., 2017; Wang, 2022; Zhuykov et al., 2015). Comparing the two methods:

1. **AOT Compilation**
   (a) *Definition*: AOT compilation refers to the process of compiling source code into machine code before the application is executed. This typically happens during the software build process.
   (b) *Performance*: Since the code is already compiled before execution, there is no compilation overhead at runtime, leading to potentially faster startup times. The compiler can also perform extensive optimisations because it has the entire codebase available.
   (c) *Deployment*: AOT compiled code is platform-specific, meaning the compiled binary must match the target architecture of the computer system.
   (d) *Portability*: Less portable compared to JIT because different binaries are needed for different ISAs and OSs.
   (e) *Error Detection*: Errors can be detected at compile time, providing a chance to catch and fix issues before deployment.
   (f) *Examples*: C, C++, and Rust are typically AOT compiled programming languages.

2. **JIT Compilation**
   (a) *Definition*: JIT compilation involves compiling source or intermediate code into machine code, typically at runtime, as the program is being executed.

(b) *Performance*: May have a slower startup time due to the compilation overhead during execution. Can optimise code based on runtime information, potentially leading to better optimisations when compared to AOT.

(c) *Deployment*: JIT compiled code is more adaptable because it compiles on the target machine, making it platform-agnostic.

(d) *Portability*: Highly portable since the same code can run on any platform with a compatible JIT compiler.

(e) *Error Detection*: Some errors may only be detected at runtime, which can make debugging more complex.

(f) *Examples*: Java, JS and WASM are typically JIT compiled programming languages.

Given the overhead imposed by JIT-based programming languages, one should thus seek to use AOT-based programming languages from a performance-based point of view (Haßler & Maier, 2021; Herman et al., 2014; McAnlis et al., 2014a; McAnlis et al., 2014b; Ménétrey et al., 2021; Park et al., 2017; Park et al., 2018; Salim et al., 2020; Serrano, 2018; Serrano, 2021; Šipek et al., 2019; Szewczyk et al., 2022; Titzer, 2022; Van Es et al., 2017; Wang, 2022; Zhuykov et al., 2015).

4.1.3. Process isolation

Employing a ZTA is both prudent and wise when using a web browser, given the level of malicious activity that takes place across the Internet (Belkin et al., 2019; Bhansali et al., 2022; Bian et al., 2019; Borisov & Kosolapov, 2020; Brito et al., 2022; De Macedo et al., 2021; Douceur et al., 2008; Fras & Nowak, 2019; Hilbig et al., 2021; Kharraz et al., 2019; Konoth et al., 2018; Lehmann et al., 2020; Mazaheri et al., 2022; Musch et al., 2019a; Musch et al., 2019b; Nießen et al., 2020; Rahimi, 2021; Rose et al., 2020; Sun et al., 2019; Sun & Ryu, 2018; Wang et al., 2019; Wen & Wang, 2007; Wirfs-Brock & Eich, 2020; Yin et al., 2015; Yu et al., 2020). The common experience is that not a day goes by that one does not hear of another entity or individual that has fallen victim to a malicious web site.

Given this, if one was to be able to download and execute natively compatible applications from the Internet, one would need to ensure that the application has limited access to resources available on the computer system (Rose et al., 2020; Wen & Wang, 2007). Failing to do so would allow malicious applications to take over a user's computer system, steal their data, and then use said data to perform all sorts of illegal activities.

One mechanism that one can use to limit an application's access to computer system resources is by isolating its processes and then only allowing a very limited and specific set of resources to be accessible (Rose et al., 2020; Wen & Wang, 2007). Using this foundation, the characteristic of process isolation can therefore be assumed to be crucial for several reasons, primarily revolving around security, stability, and performance, which can be articulated as follows:

1. **Security**
    (a) *Containment of Malicious Content*: By isolating web pages and their associated processes, web browsers can prevent malicious applications on one page from affecting others. This helps mitigate attacks such as Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF).
    (b) *Protection Against Vulnerabilities*: If a vulnerability is exploited in one process, the attacker is contained within that process and cannot easily access or control other processes. This limits the scope and impact of the attack.
    (c) *Sandboxing*: Each isolated process can run with limited privileges, reducing the risk of system-level exploits. This means that even if a process is compromised, the attacker's ability to harm the computer system is restricted.
2. **Stability**
    (a) *Crash Containment*: If a web page or its associated processes crashes, only the process handling that specific page or application is affected. This prevents the entire browser from crashing and ensures that other open tabs remain unaffected and functional.
    (b) *Resource Management*: Isolating processes helps manage resource usage more effectively. If one web page or their associated processes consumes excessive memory or CPU, it can be managed or terminated without impacting other web pages.
3. **Performance**
    (a) *Efficient Multithreading*: Modern CPUs have multiple cores, and process isolation allows web browsers to take advantage of these cores by running different processes concurrently. This can lead to better performance and responsiveness.
    (b) *Prioritisation*: Browsers can prioritise processes based on user interaction. For instance, the web page that the user is actively engaging with can be given a higher priority, ensuring a smoother experience.

Overall, process isolation enhances the user experience by making web browsers more secure, stable, and responsive, especially when executing natively compatible applications.

Let us now look at what a potential web browser EE architectural design may look like when employing these three characteristics.

4.2. Conceptual design

Reflecting on the design constraints while considering what a well-aligned architecture might look like, we can expand on and dig deeper into the design presented in figure 1.3, section 1.5. What has been derived at is a simple yet elegant conceptual design with which to solve the problem at hand, which is depicted in figure 4.1.

In the design, the containers represent the different parts of a complete web browser, the lines connecting the containers represents the interactions between them, such as communication protocols, API calls, and data flows to name a few, while the users depict the people or systems that interact with the containers.

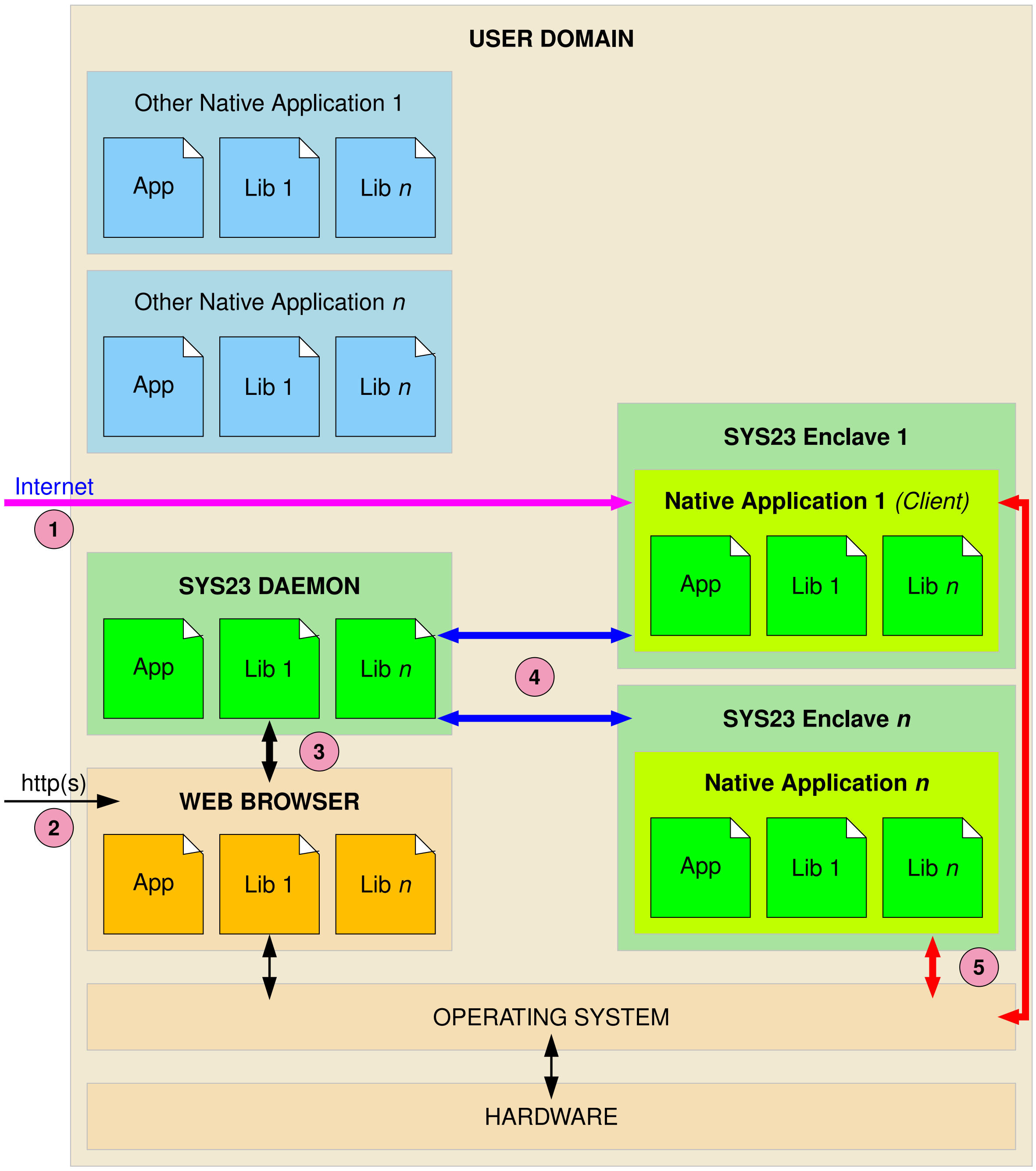

Figure 4.1: C4 Container (Level 2) Conceptual Design

The key component or crown jewel of the design is the SYS23 enclave, which is based on existing container technologies similar to that of Docker (Fras & Nowak, 2019; Kozlovičs, 2020; Manco et al., 2017; Randal, 2020; Szewczyk et al., 2022) and LXC (Manco et al., 2017; Randal, 2020). The simple idea is to use a specifically crafted container within which one can execute untrusted native applications.

Dissecting the design, one is presented with the following containers:

1. **Web Browser**: Facilitates the downloading of the natively compatible application. It will provide the web server with the relevant details with which it can determine which of the available applications, given the user's computer system ISA and OS, should be downloaded to the user's computer system.
2. **System23 Daemon**: Receives the downloaded application and orchestrates the isolation of the application into its own enclave, which restricts the functions of the applications and its processes. This daemon would manage the full lifecycle of the enclaves together with their various restrictions.
3. **System23 Enclave**: Is the actual enclave that is heavily restricted to what functions the application and its processes can perform.

In addition to the containers, the design also depicts the relationships or connections between the containers, which are numbered as follows:

1. This relationship depicts an example internet connection from an application executing in a SYS23 enclave. Those network connections would be managed by the network namespace as described later and would allow for example, client/server type applications.
2. This relationship depicts the stock standard http and https connections that are facilitated via one's web browser.
3. This relationship depicts how a web browser would integrate with the SYS23 daemon. The mechanism used could for example be that of a Multipurpose Internet Mail Extension (MIME) (Freed & Borenstein, 1996) type trigger.
4. This relationship depicts the interactions between the SYS23 daemon and all of its enclaves.
5. This relationship depicts the interactions between the SYS23 enclaves and the OS, which as will be described later, will be facilitated by the Secure Computing Mode (Seccomp) restrictions built into the SYS23 enclave.

This idea and design is similar to that of Wen and Wang's (2007) with their untrusted EE, but it comes without the overhead of a fully fledged VM or OS, which makes it smaller and more compact, thus requiring fewer resources. The prominent aspect of the SYS23 enclave is that it fully encapsulates the execution of an application, where the application can only perform functions that the SYS23 enclave allows. Also, the SYS23 executes native applications that have previously been built using an AOT compiler, thus it does not require any JIT compiler.

Let us now look at these aspects in more detail.

4.3. Prototype architecture

Building on the design considerations and the conceptual design, we can now look at what a high-performance web browser EE might look like together with what sort of technologies it would contain.

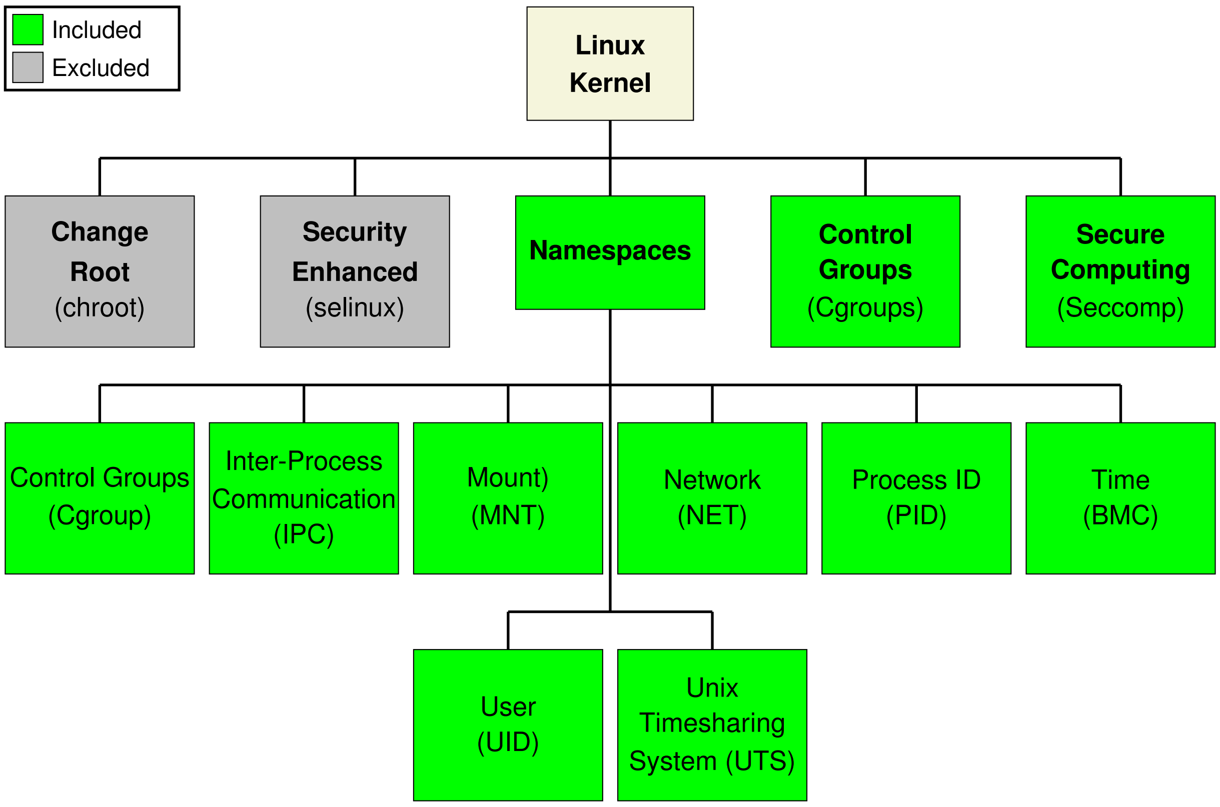

Figure 4.2: C4 Component (Level 3) Prototype Design

4.3.1. Native is good, native is fast

As argued in the design considerations, running native machine code is the most performance efficient way of executing CIAs, because executing native machine code is generally considered better than executing interpreted code for several reasons (Arteaga et al., 2020; Belkin et al., 2019; Bourgoin & Chailloux, 2015; Brito et al., 2022; Cho et al., 2015; Choi & Moon, 2019; De Macedo et al., 2021; De Macedo et al., 2022; Douceur et al., 2008; Haßler & Maier, 2021; Herrera et al., 2018; Hockley & Williamson, 2022; Jangda et al., 2019; Jansen & van Groningen, 2016; Jiang & Jin, 2017; Konoth et al., 2018; Koper & Woda, 2022; Lehmann et al., 2020; Letz et al., 2018; Liu et al., 2022; Lyu, 2021; Ma et al., 2019; Malle et al., 2018; McAnlis et al., 2014a; McAnlis et al., 2014b; Ménétrey et al., 2021; Musch et al., 2019a; Na et al., 2016; Nießen et al., 2020; Powers et al., 2017; Reiser & Bläser, 2017; Rossberg, 2022; Rossberg et al., 2018; Šipek et al., 2021; Spies & Mock, 2021; Stiévenart et al., 2022; Sun et al., 2019; Szewczyk et al., 2022; Titzer, 2022; Tushar & Mohan, 2022; Van Es et al., 2017; van Hasselt et al., 2022; Verdú & Pajuelo, 2016; Vilk & Berger, 2014; Wagner, 2017; Wang, 2021; Wang, 2022; Wang et al., 2019; Wen et al., 2020; Wen & Wang, 2007; Yin et al., 2015; Yu et al., 2020; Zakai, 2011;

Zakai, 2017; Zakai, 2018; Zhuykov & Sharygin, 2017). These reasons can be summarised as follows:

1. **Performance**
   1. (a) *Execution Speed*: Native machine code runs directly on the hardware without the need for an intermediary, making it significantly faster than interpreted code, which must be translated on-the-fly.
   2. (b) *Optimisation*: Compilers can optimise native machine code during the compilation process, taking advantage of specific hardware features and performing advanced optimisations that are not possible in an interpreted environment.
2. **Resource Utilisation**
   1. (a) *Efficiency*: Native machine code tends to use system resources more efficiently because it runs directly on the CPU and can be optimised for memory and CPU usage.
   2. (b) *Lower Overhead*: Interpreted code requires an interpreter, which adds overhead and consumes additional memory and CPU cycles.
3. **Predictability**
   1. (a) *Consistent Performance*: The performance of native machine code is more predictable because it does not depend on the interpreter's runtime performance, which can vary based on different factors.
   2. (b) *Deterministic Behaviour*: Native machine code execution is more deterministic, which is crucial for real-time and embedded systems where timing is critical.
4. **Security**
   1. (a) *Reduced Attack Surface*: Running native machine code reduces the attack surface associated with the interpreter. Interpreters can have vulnerabilities that native code execution avoids.
5. **Deployment**
   1. (a) *Standalone Executables*: Native machine code can be compiled into standalone executables, simplifying deployment since there is no need to distribute an interpreter with the application.
   2. (b) *Compatibility*: While native machine code is platform-specific, it can be highly tuned for a given environment, ensuring maximum performance and reliability for that platform.

Overall, the choice between native machine code and interpreted code depends on the specific requirements of the application, including performance, resource constraints, and portability needs. For SYS23 the C programming language was used with which to test and benchmark the prototype artefact.

4.3.2. Keep it simple, stupid

In the design considerations, specifically section 4.1.2., we described how AOT compiled programming languages are more desirable than JIT ones as they avoid the need for an interpreter. Currently, quite a few programming languages exist that are required to be interpreted at runtime, such as Python, Java, JS, and WASM to name a few (Ahn et al., 2014; Auler et al., 2014; Chandra et al., 2016; Cho et al., 2015; Choi et al., 2019; De Macedo et al., 2021; De Macedo et al., 2022; Douceur et al., 2008; Frankston, 2020; Gong et al., 2015; Grimmer et al., 2018; Haßler & Maier, 2021; Herrera et al., 2018; Jangda et al., 2019; Jansen & van Groningen, 2016; Konoth et al., 2018; Koper & Woda, 2022; Letz et al., 2018; Maas et al., 2017; Manco et al., 2017; Matsakis et al., 2014; McAnlis et al., 2014a; Na et al., 2016; Park et al., 2017; Powers et al., 2017; Reiser & Bläser, 2017; Rossberg et al., 2018; Salim et al., 2020; Selakovic & Pradel, 2016; Serrano, 2018; Serrano, 2021; Šipek et al., 2019; Southern & Renau, 2016; Spies & Mock, 2021; Sun & Ryu, 2018; Szabó & Nehéz, 2019; Titzer, 2022; Tushar & Mohan, 2022; Ueda & Ohara, 2017; Van Es et al., 2017; Wang, 2021; Wang, 2022; Wang et al., 2019; Wen et al., 2020; Wirfs-Brock & Eich, 2020; Yan et al., 2021; Zakai, 2011; Zhuykov et al., 2015; Zhuykov & Sharygin, 2017).

In order to interpret these programming languages, their runtime environments usually employ a VM. However, when employing a VM, it brings with it all of the overhead that is required to transform the programming language into native machine code so that the CPU can execute the application. This overhead can significantly impact the performance of the application.

Conversely, applications that do not require any interpretation at runtime are not constrained by any VM or the overhead that comes with it. Applications that have been developed in the C and C++ programming languages (Brito et al., 2022; De Macedo et al., 2021; De Macedo et al., 2022; DiPierro, 2018; Douceur et al., 2008; Fras & Nowak, 2019; Haßler & Maier, 2021; Jangda et al., 2019; Jiang & Jin, 2017; Koper & Woda, 2022; Lehmann et al., 2020; Lehmann & Pradel, 2022; Malle et al., 2018; McAnlis et al., 2014b; Nießen et al., 2020; Rossberg et al., 2018; Salim et al., 2020; Spies & Mock, 2021; Szabó & Nehéz, 2019; Szewczyk et al., 2022; Ueda & Ohara, 2017; Zakai, 2018) are often considered some of the fastest due to several key factors:

1. **Low-Level Access**: C and C++ provides direct access to memory and hardware through pointers, thereby enabling fine-tuned optimisations that are not possible in higher-level languages.

2. **Minimal Overhead**: The C and C++ languages have minimal runtime overhead. It lacks features like garbage collection and extensive runtime type checking, which can slow down execution in other languages.

3. **Efficient Compilation**: C and C++ compilers, such as GCC and Clang, are highly optimised and produce efficient machine code. These compilers have been refined over decades to generate code that runs very close to the hardware's potential speed.

4. **Simple Language Constructs**: C and C++ are relatively simple languages with a small number of keywords and constructs, which allows for more straightforward and faster execution.

5. **Portability**: C and C++ are highly portable and can be compiled for virtually any architecture, allowing the same code to be run efficiently on different hardware platforms.

6. **Control Over System Resources**: C and C++ gives programmers fine-grained control over system resources such as CPU and memory. This control allows for optimisations that can lead to significant performance improvements.

These characteristics make the C and C++ programming languages ideal choices for performance-critical applications where speed and performance are essential, such as with the SYS23.

4.3.3. Protect the innocent

Focussing specifically on the Linux OS and understanding the importance of application and process isolation, we can propose the following technologies, noting that similar technologies also exist within other OSs such as Windows. Containers such as the ones that are available through Docker are a form of process isolation, combined with resource management and security features (Biradar et al., 2018; Fras & Nowak, 2019; Manco et al., 2017; Randal, 2020; Yu et al., 2020). The concept of a container does not exist inside of Linux or any other currently existing OS.

A container is a term that is used to describe a combination of OS technologies, such as change root, namespaces, Control Groups (Cgroups), and Seccomp, which are features that one can use to isolate processes from one another. Isolating a process this way is not an all or nothing approach as you can be very selective over what resources are isolated and which ones are not. For SYS23, the intent is to use a combination of these Linux OS features. Let us now look at each technology and how they would form part of the SYS23 enclave.

4.3.3.1. Change root

Change root or `chroot` is a Linux shell command that allows one to set the root directory of a process to any other directory. This enables one to hide the computer system's actual root directory and provide any process with a fake root directory or jail. Hiding the computer system's actual root directory from a potentially malicious process has many security advantages, foremost of which is that in no way can the malicious process access any OS or user files (Ménétrey et al., 2021; Randal, 2020; Shepherd & Markantonakis, 2024; The Linux Foundation, 2024) thus rendering the malicious process harmless from a file access point of view.

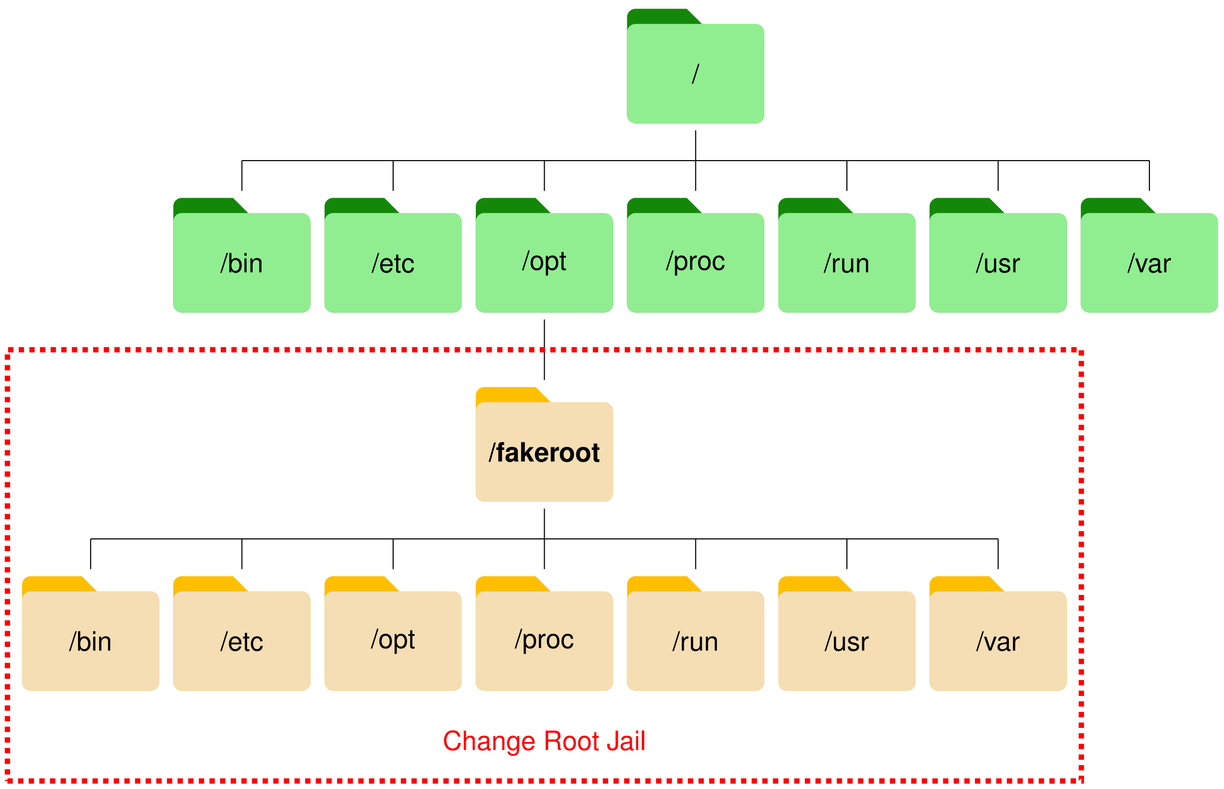

Figure 4.3: Change Root Example Directory Structure

The following shell command example and figure 4.3 shows how one would create a fake root called `/fakeroot` for a new shell process. Note that the new root directory does need to contain all of the required OS files and other libraries in order for the process to execute successfully.

Shell Command

```
$ mkdir /fakeroot && chroot /fakeroot bash
```

Change root is not a very secure mechanism to use and using it on its own is not going to provide one with a foolproof solution with which to protect one's files. Also using change root programmatically within say C programming code has a few caveats, such as the fact that when using the change root system call, the current working directory is not switched automatically (Randal, 2020; Shepherd & Markantonakis, 2024; The Linux Foundation, 2024). Also relative paths still refer to directories and files outside of the new root directory.

Change root is a useful tool for specific use cases, particularly when lightweight isolation or testing environments are needed. However, for more robust isolation, security, and ease of use, modern isolation technologies such as those provided by *control groups*, *namespaces*, and Seccomp are preferred by SYS23.

4.3.3.2. Security-enhanced Linux

A security architecture for Linux OS called Security-Enhanced Linux (SELinux) gives administrators additional control over who can access a Linux system by utilising access control security policies. It was originally created as a set of patches for the Linux kernel utilising Linux Security Modules (LSMs) by the National Security Agency (NSA) of the United States (Randal, 2020; Shepherd & Markantonakis, 2024; The Linux Foundation, 2005). In 2000, SELinux was made available to the open-source community and in 2003 it was merged into the upstream Linux kernel version 2.6.

A flexible Mandatory Access Control (MAC) solution, as provided by SELinux is integrated into the Linux kernel. An application or process operating as a user (UID or SUID) has the user's permissions to objects like files, connections, and other processes under standard Linux Discretionary Access Control (DAC). Utilising a MAC-based OS kernel shields the Linux system from faulty or malevolent programs that could harm or even destroy it.

Every user, application, process, and file on the Linux system has its access and transition permissions defined by SELinux. Then, using a security policy that defines how strict or lenient a particular Linux system should be, SELinux regulates the interactions of these entities (Shepherd & Markantonakis, 2024; The Linux Foundation, 2005). How strict a policy to implement for their Linux systems is to be considered by the system administrators. The policy is comprehensive and can be applied with varying degrees of severity, which gives SELinux complete fine-grained control over the Linux system.

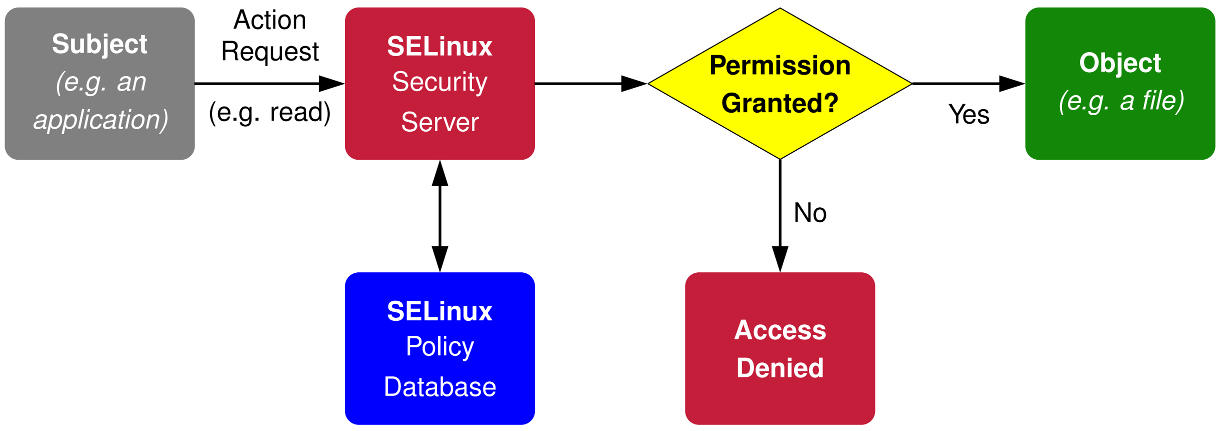

Figure 4.4: Security-Enhanced Linux Decision Process

When a subject, such as an application, attempts to access an object, such as a file, the SELinux policy enforcement server in the Linux kernel checks an access vector cache, where subject and object permissions are cached. If a decision cannot be made based on data in the access vector cache the request continues to the security server, which looks up the security context of the application and the file in a matrix, based on which permission is then granted or denied. This decision process is depicted in figure 4.4. The security context of subjects and objects are applied from the pre-installed policy, which also provides the information to populate the security server's matrix.

SELinux can operate in one of three modes: enforcing, permissive and disabled. Instead of running in enforcing mode, SELinux can run in permissive mode, where the access vector cache is checked and denials are merely logged, as SELinux does not enforce the policy. This can be useful for troubleshooting and for developing or fine-tuning SELinux policy. Disabled mode should be avoided, as the system does not enforce the SELinux policy and does not label any persistent objects such as files, consequently enabling SELinux in the future becomes quite difficult (Shepherd & Markantonakis, 2024; The Linux Foundation, 2005).

When SELinux is used together with SYS23, it secures all of the resources outside of the SYS23 enclave and does not play any part within the SYS23 enclave.

4.3.3.3. Secure computing mode

Seccomp is a feature within the Linux kernel that allows one to filter the hundreds of system calls from a process to the kernel. Seccomp provides fine-grained access control to system calls through one of two operating modes. The first, called *strict* mode, utilises a list of predefined system calls that are allowed, the second, called *filter* mode, utilises a user customisable filter based on the Berkeley Packet Filter (BPF) rule system (Randal, 2020; Shepherd & Markantonakis, 2024; The Linux Foundation, 2023j).

In *strict* mode, Seccomp restricts processes to a limited set of allowed system calls, namely `read`, `write`, `sigreturn`, and `exit`. The `read` and `write` system calls are only allowed for file descriptors that are already open, furthermore, all other system calls are automatically denied. Any process that attempts to initiate a denied system call while operating in *strict* mode will immediately be terminated by the Linux kernel. Figure 4.5 shows how an example *Application A* would be restricted.

In *filter* mode, Seccomp restricts processes to a custom set of allowed system calls as well as setting conditions as to when or how it should be restricted. If a process is allowed to use the `fork` or `clone` system calls, then the resulting child process will also be constrained to the same list of system calls as the parent process. As with *strict* mode, any process that attempts to initiate a denied system call can be immediately terminated or some other action can also be configured, such as logging the attempted breach or redirecting the call to another process for processing. Figure 4.5 shows how an example *Application B* would be restricted.

Seccomp is a useful mechanism with which one can limit the interactions of a process with the underlying OS. It can also provide a strong layer of additional protection to the SYS23 enclave, whereby it can limit the attack surface for any malicious applications that may execute within the enclave, for example by restricting system calls related to the reading and writing of files residing external to the enclave (Randal, 2020; Shepherd & Markantonakis, 2024; The Linux Foundation, 2023j). For these reasons, Seccomp forms a valuable part of the SYS23 solution.

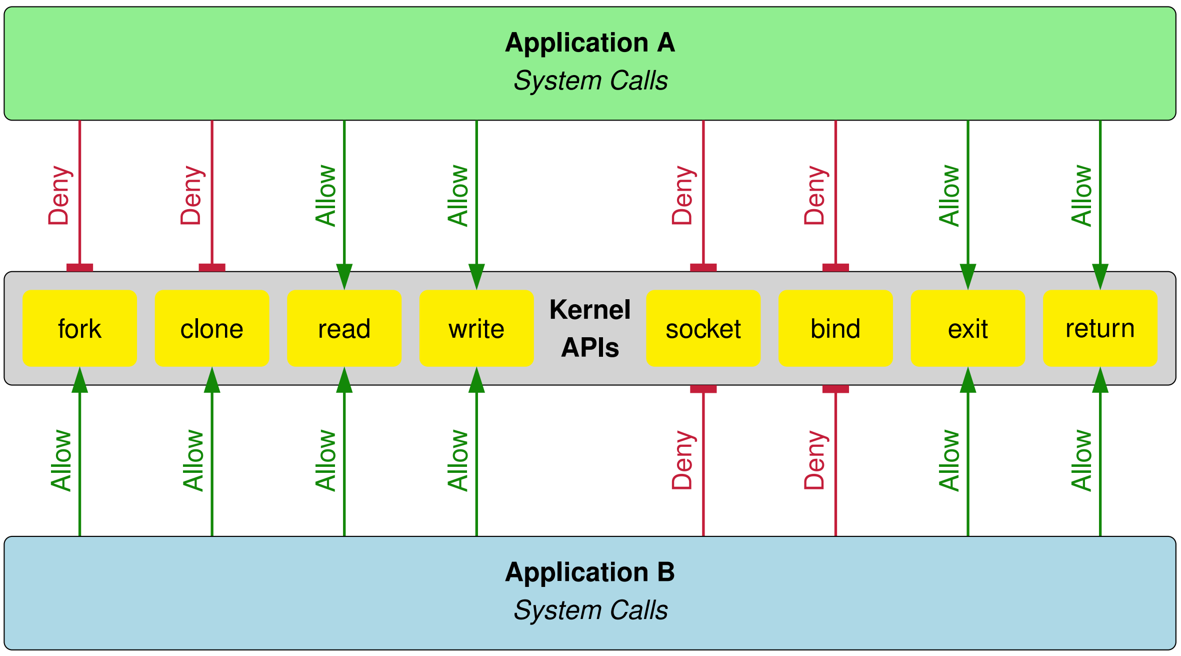

Figure 4.5: Secure Computing Mode

4.3.3.4. Control groups

Cgroups are a Linux kernel mechanism that can be used to impose usage limits on certain system resources, such as the maximum amount of CPU usage that can occur or the maximum amount of memory that can be used by a process, to name a few. By setting limits to certain system resources one can prevent any single process from using or claiming all available system resources, which in turn would cause the underlying computer system to become overloaded, unresponsive and unusable (Biradar et al., 2018; Fras & Nowak, 2019; Randal, 2020; Shepherd & Markantonakis, 2024; The Linux Foundation, 2023b).

Cgroups are hierarchical in structure where child Cgroups automatically inherit certain immutable attributes from their parent Cgroup. Every process within a Linux-based OS belongs to at least one Cgroup. The Linux kernel exposes Cgroups through a pseudo-filesystem called the *cgroupfs* which can be located within the Linux filesystem as `/sys/fs/cgroup`. Figure 4.6 depicts how various applications may utilise Cgroups to assist them in managing their system resource needs.

Furthermore, Cgroups are divided into the following set of high-level subsystem controllers, which allows one to manage the underlying system resources as required:

1. **Block IO**: Manages access to I/O on block devices.
2. **CPU**: Manages CPU usage and scheduling.
3. **CPUSET**: Manages specific CPU allocations to specific processes.
4. **Device**: Manages access to devices.
5. **Memory**: Manages memory usage and limits.

6. **Network**: Manages network usage and limits.

7. **Process**: Manages process limits.

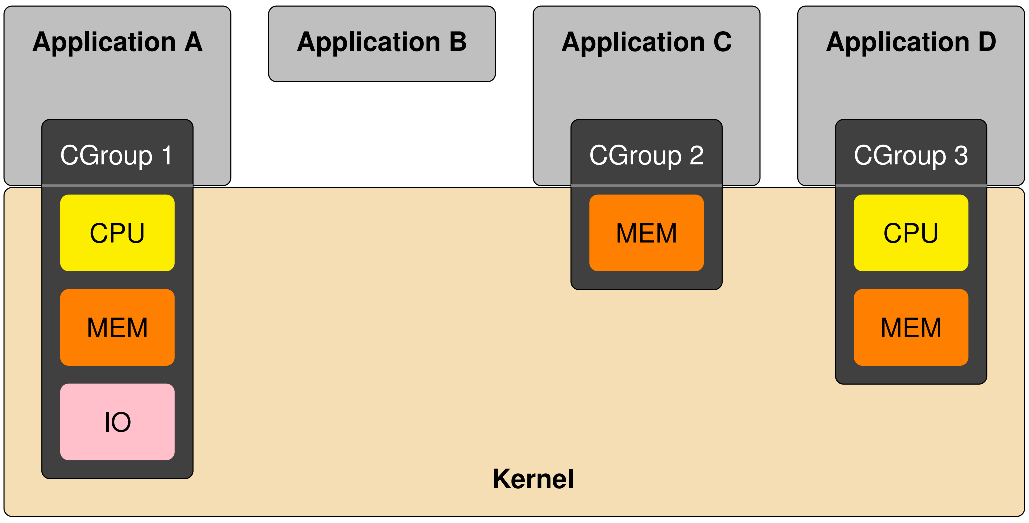

Figure 4.6: Control Groups

When utilised within the SYS23 enclave, one can quickly see the advantages that Cgroups will bring to the management of the enclave by providing a mechanism within which one is able to limit and restrict the amount of system resources an application executing within the enclave maybe able to gain access to (Biradar et al., 2018; Randal, 2020; Shepherd & Markantonakis, 2024; The Linux Foundation, 2023b). By doing so we can easily restrict a malicious application from trying to exhaust the system resources and render it unresponsive or unusable.

4.3.3.5. Namespaces overview

Change root stems from first generation Portable Operating System Interface (POSIX) computer systems and can be seen as a rudimentary form of namespaces. Namespaces are a Linux kernel feature that allows one to wrap system resources in an abstraction layer. Doing so allows one to make processes within the namespace believe that they have their own isolated instance of the system resource. Changes to the system resource are only visible to other processes that are members of the same namespace (Randal, 2020; Shepherd & Markantonakis, 2024; The Linux Foundation, 2023e).

Table 4.1 lists the currently available namespaces that one may utilise within Linux, either directly from the command-line or through the use of a programming interface when using a programming language such as C/C++.

Table 4.1: Linux OS Namespaces

Namespace	Description
Cgroup	Isolates the control group root directory
IPC	Isolates interprocess communication resources
MNT	Isolates filesystem mount points
NET	Isolates networking interfaces
PID	Isolates the process identification number space
Time	Isolates the system boot and other monotonic clocks
User	Isolates the user and group identifications
UTS	Isolates the hostname and NIS domain name

4.3.3.6. Control group namespace

A Cgroup namespace is a feature of the Linux kernel that isolates Cgroup hierarchies for processes executing within that namespace essentially, it is related to the concept of namespace isolation. The Cgroup namespace provides a virtualised view of a process's Cgroup hierarchy as described in section 4.3.3.4. This virtualised view enables processes executing within one Cgroup namespace to have a different view of its Cgroup hierarchy as compared to other processes and their Cgroup hierarchies running in their Cgroup namespaces (Biradar et al., 2018; The Linux Foundation, 2023a).

When creating a new Cgroup namespace, it will be initialised with the virtualised view of the Cgroup hierarchy that is based on the Cgroup of the parent process. Therefore processes executing in the new Cgroup namespace will be limited to viewing only the portion of the overall Cgroup hierarchy that is related to the parent process. An example of this virtualised view is given in figure 4.7, where both processes, each of which contains a memory related Cgroup has its own unique view of the Cgroup hierarchy.

As depicted in figure 4.7, we can see that processes within one Cgroup namespace see their own Cgroup as the root of their hierarchy. Therefore, from their perspective, their Cgroup appears as the root, and they are unable to see or access Cgroups outside their own hierarchy. Cgroup namespaces do not provide for the isolation of system resources, they only provide for the isolation of the related Cgroup hierarchies. Isolation of the system resources are enforced by Cgroups itself, as described earlier.

The obvious benefit of being able to isolate Cgroup hierarchies is that one process cannot change the details of Cgroups that they do not have access to, thus a malicious process would be unable to manipulate Cgroups systemwide. Given the inclusion of Cgroups within the SYS23 design, it stands to reason to include Cgroup namespaces as well.

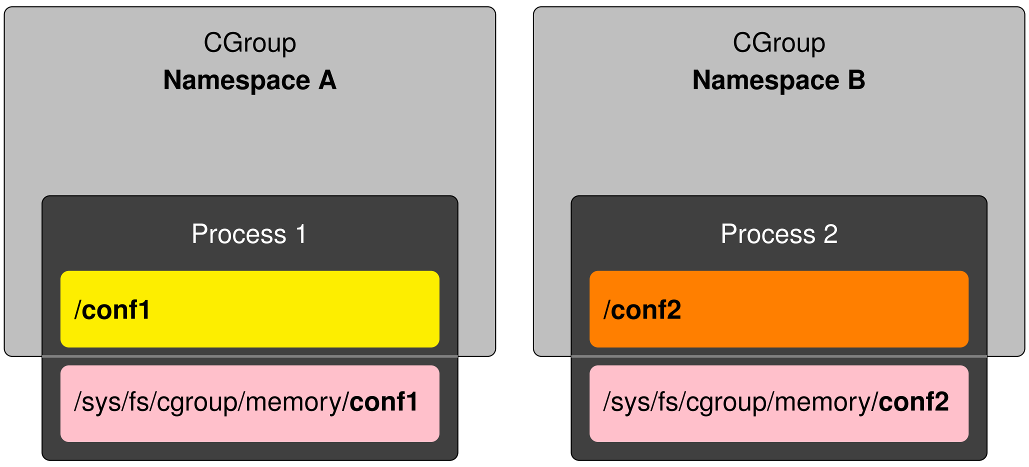

Figure 4.7: Control Groups Namespace

4.3.3.7. Inter-process communication namespace

An Inter-Process Communication (IPC) namespace is a feature of the Linux kernel that isolates IPC resources, specifically System V IPC objects and POSIX message queues, such as semaphores, shared memory segments, and message queues, to name a few. As with other types of namespaces, this isolation ensures that processes from one IPC namespace, can share System V IPC objects and POSIX queues within that namespace, but cannot directly access or engage with System V IPC objects and POSIX queues from other IPC namespaces (Biradar et al., 2018; The Linux Foundation, 2023c).

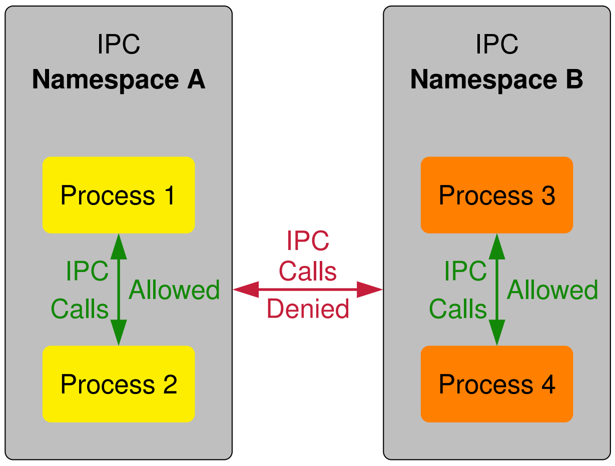

Figure 4.8: Inter-Process Communication Namespace

System V IPC and POSIX message queues are a set of mechanisms in Unix-like OSs that allows processes to communicate with each other and synchronise their operations. There are three primary System V IPC and POSIX message queue components supported by IPC namespaces (The Linux Foundation, 2023c):

1. **Semaphores**: Provide a process synchronisation mechanism, ensuring that multiple processes can safely access shared system resources, such as memory segments.

2. **Shared Memory**: Provides a mechanism that allows multiple processes to utilise the same physical memory segments.
3. **Message Queues**: Provide a mechanism whereby processes can send and receive messages between them, using a queue-like structure.

Other System V IPC and POSIX message queue components such as spinlocks (Waterman & Asanović, 2019) and mutexes (Ueda & Ohara, 2017; Waterman & Asanović, 2019) are currently not supported by IPC namespaces. Furthermore, both System V and POSIX-based message queues are supported by IPC namespaces. POSIX message queues are similar in purpose to System V message queues, but they are part of the POSIX standard, which aims to improve portability and consistency across different Unix-like systems (The Linux Foundation, 2023c).

Figure 4.8 shows the segregation of inter-process communications between processes, both within and outside of the IPC namespaces. IPC is a fundamental part of any well-designed and implemented application and as such, any SYS23 enclave would be less than optimal in design if such a mechanism was not supported by it. As such IPC namespaces form an integral part of the SYS23 enclave.

4.3.3.8. Mount namespace

A mount namespace is a feature of the Linux kernel that isolates the mount points of a filesystem so that processes in one mount namespace cannot access any mount points inside other mount namespace or from the rest of the Linux kernel. Each mount namespace has its own view of the underlying file system mount points and as with other namespaces, changes to a mount point within one mount namespace does not propagate to any other mount namespace, see figure 4.9 for an example visual representation. In turn, mount point changes in the rest of the Linux kernel do not propagate to any mount namespaces once created (Biradar et al., 2018; The Linux Foundation, 2023d).

Mount namespaces and `chroot` are both techniques used in Unix-like computer systems to isolate processes and their associated mount points. The mount point isolation created by mount namespaces also create a fake root directory or jail that is similar to what `chroot` provides, except that it is much more secure, as it does not include any of the limitations that `chroot` does.

Mount namespaces are generally considered more secure than `chroot` for the following primary reasons:

1. **Broader Isolation Scope**: While `chroot` changes the root directory of the current process and its child processes to a specific directory that can be escaped if they have elevated privileges, such as root access. Mount namespaces provide a more comprehensive level of isolation by creating an entirely separate view of the filesystem for the processes within that mount namespace (The Linux Foundation, 2024).

2. **Namespace Hierarchies**: While `chroot` only changes the apparent root directory for a process, it still shares the same global mount table with all other processes. Whereas each mount namespace has its own mount table, allowing for more fine-grained control over what filesystems and mount points are available (The Linux Foundation, 2024).

3. **Flexibility and Control**: While `chroot` creates simple jail environments, mount namespaces allow for more sophisticated configurations, such as shared or private mount points, making it possible to control exactly how filesystems and mount points are propagated between namespaces (The Linux Foundation, 2024).

4. **Security Vulnerabilities**: While `chroot` has several well-known methods for escaping its filesystem jail, especially if the process has root privileges, mount namespaces are part of a broader namespace mechanism that is designed with modern security in mind, reducing the chances of escape or unintended interactions with other mount namespaces or the rest of the Linux kernel (The Linux Foundation, 2024).

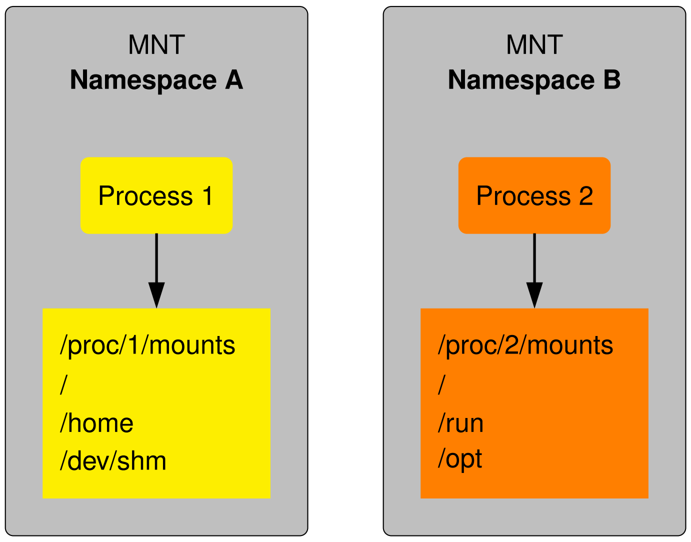

Figure 4.9: Mount Namespace

The mount namespace is one of the foundational namespaces, as it also underpins both the Cgroup and process namespaces and any other current or future namespaces where aspects of the filesystem are included as part of their implementation (The Linux Foundation, 2023d). For these reasons mount namespaces also form a foundational part of the SYS23 enclave as filesystems are a key component of any isolated environment, without which any application would find it nearly impossible to execute successfully.

4.3.3.9. Network namespace

A network namespace is a feature of the Linux kernel that isolates the networking environment such as the Internet Protocol (IP) version 4 and 6 protocol stacks, IP routing tables, firewall rules, port numbers or sockets, network devices and many other networking components. A network namespace is a virtualised instance of an entire networking environment with its own routing tables, firewall rules, port numbers, network devices and so forth. Furthermore, the UNIX domain abstract socket namespace is isolated by network namespaces (Biradar et al., 2018; The Linux Foundation, 2023f).

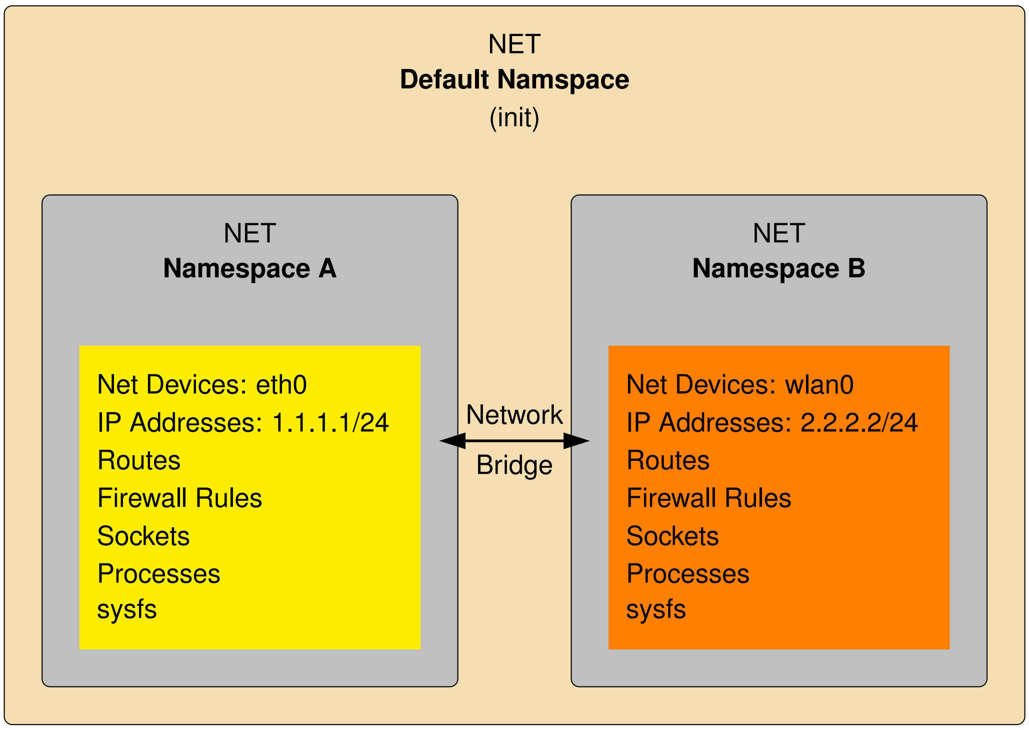

Figure 4.10: Network Namespace

Upon creation, processes default to inheriting their parent's network namespace. This default network namespace is shared with the `init` process or in simple terms, the first process that is created when a computer system is booted. When a process becomes a member of its own network namespace, it is provided with a virtual network device pair consisting of a physical network device and an equivalent virtual network device. Figure 4.10 presents a rudimentary view of how the various network namespaces may coexist.

A physical network device can only be a member of one network namespace at any point in time. When a network namespace ceases to exist, its network devices are allocated back to the default network namespace, not to the network namespace of its parent. Note that all physical network devices initially reside within the default network namespace. Also with the creation of a network namespace, it is automatically assigned its own loopback network device.

Virtual network device pairs provide a pipe-like communication abstraction that can be used to create tunnels between two network namespaces, which in turn can be used to create a bridge between the virtual and physical network devices at either end of the communications channel. Using this configuration one is able to facilitate networking between two network namespaces or between a network namespace and the default network namespace.

The use cases for a network namespace are significant, in that it can for instance allow one to do various sorts of network testing with different networking configurations without breaking the default network namespace. One can also execute untrusted applications in its own network namespace without fear that it may open up various unknown communications channels, where those communication channels will be isolated to resources residing only within that network namespace.

The network namespace is another foundational namespace that forms part of the SYS23 design. Giving the SYS23 enclave networking capabilities allows applications executing within it to communicate with other remotely located computer systems, which are fundamental to computer system design principles and concepts such as client/server, distributed computing, clustering and many more. Therefore, including network namespaces in the SYS23 design is almost expected otherwise the use cases of the SYS23 enclave would be limited to standalone applications only or applications that do not require any sort of networking.

4.3.3.10. Process namespace

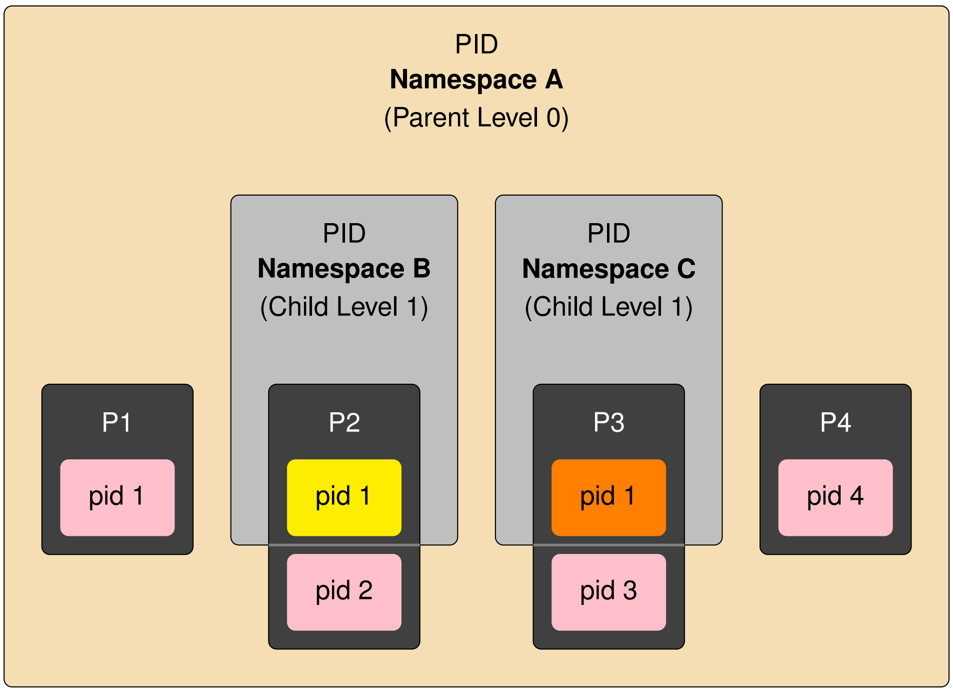

Figure 4.11: Process Namespace

The Process ID (PID) namespace is a feature of the Linux kernel that isolates the process identifiers, specifically the PID number space, by allowing each set of processes in different PID namespaces to have their own set of PIDs, without causing any conflicts across the different namespaces. The PID namespaces can be nested in a tree-like hierarchy several levels deep, up to a maximum of 32 levels. This indirectly implies that each PID namespace has a parent PID namespace, except for the `init` process, which is the only exception to this rule (Biradar et al., 2018; The Linux Foundation, 2023g). One further aspect to note is that the PID namespace is closely associated with the mount namespace, as will be shown later.

When a new PID namespace is created it will assign the PID of 1 to the first process allocated to that PID namespace, see figure 4.11 for a depiction. For all intents and purposes, one can think of this first process as the `init` process of that PID namespace. The behaviour of this first process is functionally similar to that of the `init` process of the Linux kernel, whereby all newly created processes within the same PID namespace will inherit the first process as its parent process.

Subsequently, when any first process is terminated, all descendant processes together with all descendant PID namespaces and their processes will also be terminated. Once a process is a member of a PID namespace, it can interact with not only any other processes within the same namespace, but also processes within any direct child or descendant PID namespaces. It cannot however interact with any processes within its parent PID namespaces.

Note that when a new PID namespace is created, the process that created it will not automatically be placed within that new PID namespace, only the children of that process will be allocated to the new PID namespace. This behaviour is slightly different to what happens when other namespaces are created. Also, processes are able to migrate from a parent PID namespace to a direct child or descendant PID namespace, but never to a parent PID namespace, thus migration is only ever available in one direction.

At the point in time when the first process is placed into the new PID namespace, the `/proc` filesystem is also shared with this new PID namespace. This is because the Linux kernel requires that a `/proc` directory exists, as it holds amongst other things, a numerical hierarchy and related information of each process currently executing within that PID namespace. Note that one can also overload this default allocated `/proc` directory when a mount namespace is created at the same time as the PID namespace, this will ensure that commands such as `ps` function appropriately.

The PID namespace is also another fundamental namespace that forms part of the SYS23 enclave design, as process management and isolation, is core to any application and executing it successfully. Being able to isolate an untrusted application and its individual processes are core to being able to safely execute the application that may potentially be malicious in nature, hence its inclusion in the SYS23 enclave design.

4.3.3.11. Time namespace

The time namespace is a recent feature of the Linux kernel that isolates some specific internal clocks, that are related to the monotonic and boottime clocks only. The reason why only those two clocks are virtualised is because of the specific use cases that they relate to or that revolves around live container migrations. Note also that the internal clocks are not necessarily changed, but an offset can be set which allows those locks to move forward or back in time (The Linux Foundation, 2023h).

Figure 4.12 shows which specific offsets can be set per time namespace and as previously stated only the offsets that relate to the monotonic and boottime clocks can be changed when a new time namespace is created. As with other namespaces, these offsets cannot be changed after the time namespace has been created, as they are immutable in nature. These offsets make it easier to migrate a live container or to restore one from hibernation, a checkpoint or snapshot.

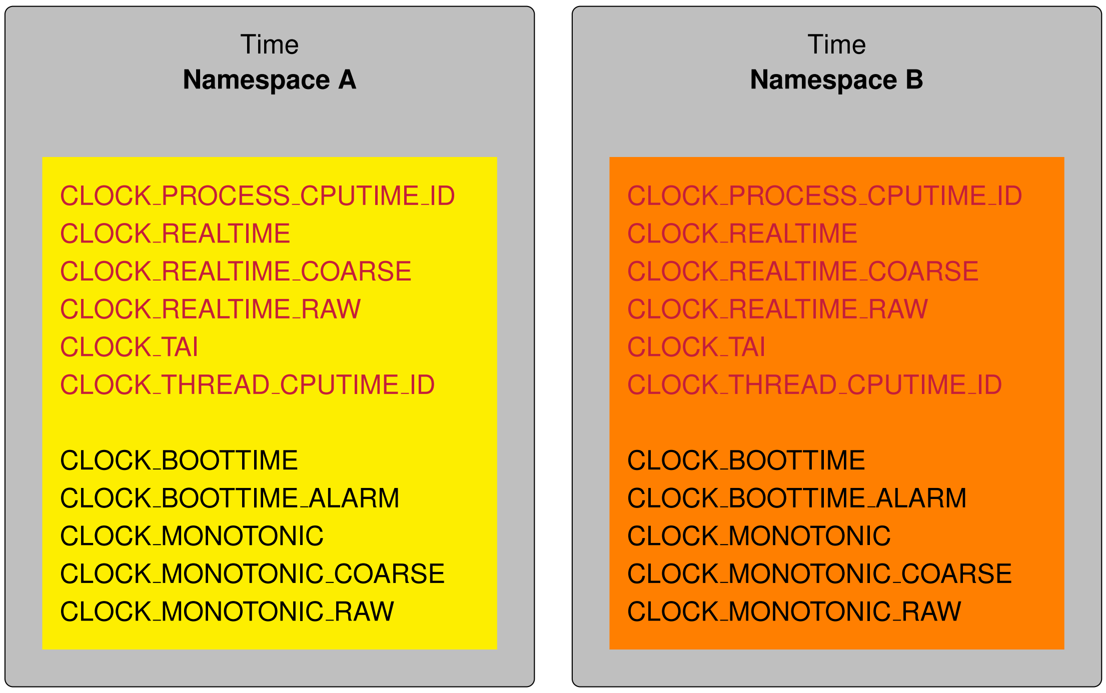

Figure 4.12: Time Namespace

Given that only those offsets relating to the monotonic and boottime clocks can be manipulated in isolation, it leaves numerous other clocks exposed to the time namespace for manipulation, such as those clocks relating to the real-time clock. Therefore if one was to set the system time within a time namespace, it will in fact also change the time outside of the namespace and across all other namespaces. This allows for caveats from a security point of view, which can be manipulated.

The time namespace is not necessarily fundamental to the use case underpinning SYS23 however, if any future use case presents itself that may take advantage of this feature, it will be readily available given its inclusion within the SYS23 design. For now though, one may consider it to be a dormant feature.

4.3.3.12. User namespace

The user namespace as per figure 4.13 is a feature of the Linux kernel that isolates User IDs (UIDs) and Group IDs (GIDs), which allows each user namespace to have its own set of user and group identifiers. User namespaces can be nested up to 32 levels deep, which is true for other namespaces too. Also, as with other namespaces, once a parent and child relationship has been created, it cannot be changed. This isolation allows processes executing in different user namespaces to have different ownership and privileges, even if they share the same UIDs and GIDs (Biradar et al., 2018; The Linux Foundation, 2023i).

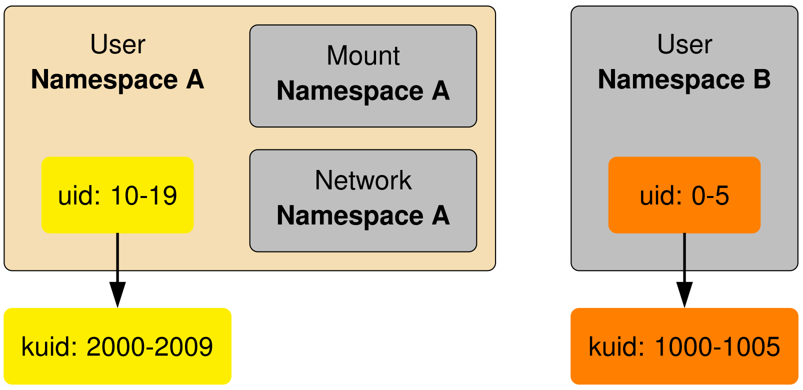

Figure 4.13: User Namespace

Allowing processes to have different ownership and privileges depending on the user namespace within which they are executing, is accomplished by assigning different UIDs and GID to different user namespaces. Given then that security privileges are linked to UIDs and GIDs, processes are limited to which operations they can execute given these UID and GID assignments. Noting then that each process can only belong to one user namespace, it allows one to set the security context of processes per user namespace.

A common use case of user namespaces is to allow processes executing within a user namespace to execute with the highest privileges, commonly called *root* privileges, while limiting its reach to within the namespace only. The process can thereby perform any operation that it requires to while in reality, those operations have no effect outside of the user namespace. As an example, when trying to read from a secured directory such as `/proc`, which requires *root* privileges, a *root* user within a user namespace would be able to access that directory, but they will not be able to see its full contents as it appears outside of the namespace.

Another useful and unique feature of the user namespace is that it is capable of owning all other types of namespaces within its 32 levels of nested hierarchy. When a new user namespace is created, the process that creates it can also create any of the other types of namespaces, within the context of this user namespace. Therefore, a process might start in a user namespace and then create its own network and mount namespaces. Within that context, the user namespace encapsulated processes will be able to, for instance, manage network interfaces and mount filesystems as if it were the *root* user, but only within the confines of the respective namespaces.

The user namespace forms an integral part of the SYS23 enclave and as such, it is part of the foundational layer of the enclave. The most obvious reason for its use is to provide adequate protection against untrusted applications executing within the enclave, that may require root privileges. Limiting the untrusted application's reach to within the enclave, allows one to be liberal with one's privilege access allocations within the user namespace, while still maintaining full security outside of the user namespace and across the rest of the computer system.

4.3.3.13. Unix time-sharing namespace

The Unix Time-Sharing (UTS) namespace is a feature of the Linux kernel that isolates two very specific system resources related to the host name and Network Information Service (NIS) domain name. The host name relates to the name of the computer system, which is typically used by various applications to identify the computer system on a network and the domain name is used by the NIS to group computer systems together into subsets when managing network service across large networks (Biradar et al., 2018; The Linux Foundation, 2022).

When a process creates a new UTS namespace, that namespace inherits the host and domain names from the parent process. As with other namespaces, changes within a specific UTS namespace after creation are restricted to that UTS namespace only. Figure 4.14 shows how different UTS namespaces would manage their own host and domain names.

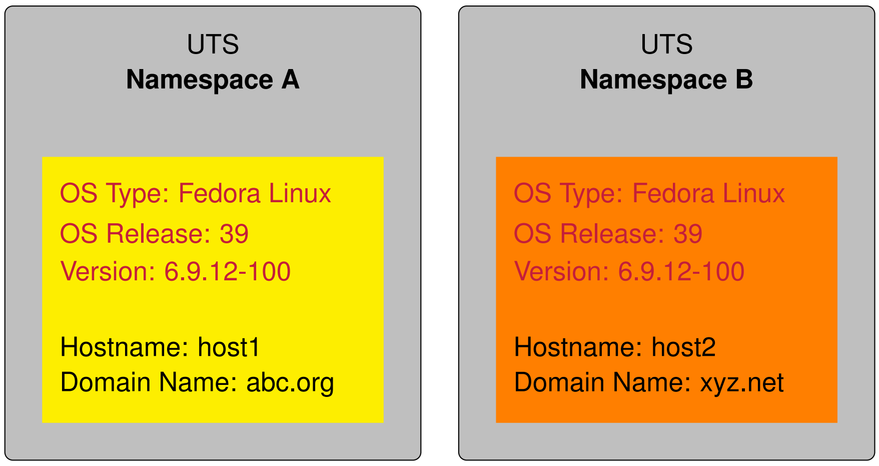

Figure 4.14: Unix Time-Sharing Namespace

UTS namespaces have the advantage that they allow a single computer system to appear as if it is many different computer systems, which is a foundational aspect of most virtualised and containerised computer systems. The UTS namespace pairs well with the network namespace, given the underlying networking aspects of both the host and domain names and how they pair with networking devices and their IP addresses.

As with the time namespace, the UTS namespace is also not necessarily fundamental to the underpinning use case of SYS23 however, for any future use case that may arise, it has been included in the SYS23 design. One aspect where UTS namespaces may provide a security mitigation is by hiding the underlying computer system's actual host and domain names. Thereby shielding its true identity from any malicious application that may have been able to take advantage of it.

4.4. Data collection

Wieringa (2014) asserts that a DSR project involves iterative cycles of designing and investigating. The design task is itself broken down into three specific subtasks, namely, problem investigation, treatment design, and treatment validation. These three subtasks are collectively referred to as the design cycle, as researchers repeat them multiple times throughout the project.

This design cycle is embedded within a broader cycle called the engineering cycle. In this larger cycle, the validated treatment produced by the design cycle is implemented, utilised, and assessed in the real world. In order to adequately evaluate the prototyped artefact, we need to collect performance data first during the design cycle and then during the engineering cycle.

The data collection design of this study intends to facilitate the gathering of performance data through the existing benchmarking harness as employed within the PolyBench/C benchmarking suite. With PolyBench/C algorithms that have an execution time in the order of a few nanoseconds, it is critical to validate any performance number by repeating the execution several times. An included companion script is available to perform a reasonable performance measurement of an algorithm.

Shell Command

```
$ gcc -O3 -I utilities -I linear-algebra/kernels/atax utilities/polybench.c
linear-algebra/kernels/atax/atax.c -DPOLYBENCH_TIME -o atax_time
```

When an algorithm is compiled with the `-DPOLYBENCH_TIME` option, as shown above for the atax algorithm, the script will execute the benchmark five times, while eliminating the two outmost times, and check that the deviation of the three remaining executions does not exceed a given threshold of 5%. For this study though, we additionally repeat the execution of this script 10 times and then take the mathematical mean of all 10 runs.

4.5. Data precision

Data precision design in research, particularly in computer science, refers to the level of detail and exactness with which data is collected, stored, measured, manipulated and presented. It is critical in ensuring the accuracy and reliability of computations, algorithms, and system processes. The precision impacts the results of numerical calculations, the behaviour of algorithms, and the overall performance of computer software (Bartz-Beielstein et al., 2020; Bryman, 2016; Bryman & Bell, 2011; Creswell & Creswell, 2018; Sanders et al., 2022; Saunders et al., 2019; Wang, 2021; Yu et al., 2020).

It is thus crucial to the reliability and validity of the research findings. Consider a scenario where a computer scientist is developing a simulation for climate modeling that requires calculating the temperature changes over time.

Using both high and low-precision measurements, one will observe the following:

1. **High Precision Measurements**: The scientist uses double-precision floating-point numbers (typically 64 bits) to represent temperature values.

   Example High Precision Data:
   15.123456789012345 ℃, 15.123456789012346 ℃, 15.123456789012347 ℃

2. **Low Precision Measurements**: The scientist uses single-precision floating-point numbers (typically 32 bits) to represent temperature values.

   Example Low Precision Data:
   15.123457 ℃, 15.123457 ℃, 15.123457 ℃

The implications of the two levels of precision within the study are:

1. **High Precision Measurements**: The double-precision representation allows for more decimal places, providing a more detailed and accurate representation of temperature changes. This is crucial for simulations that require high fidelity over long periods or involve small variations.

2. **Low Precision Measurements**: The single-precision representation has fewer decimal places, which may be sufficient for less detailed simulations but can lead to rounding errors and less accurate results over time.

In this example, high-precision measurements ensure that the simulation can accurately model the minute temperature changes, which is essential for making reliable predictions about climate patterns. Low-precision measurements might be faster and use less memory but could result in less accurate models, potentially leading to incorrect conclusions or predictions. For this reason, this study aims to provide a high level of accuracy and reliability for the findings by employing high-precision measurements during its data collection phases.

4.6. Summary

In this chapter, we furnished a prototype architecture and design as outlined in figure 4.1, for a new web browser EE that is based on existing Linux OS features, such as namespaces, Cgroups and Seccomp. The prototype incorporates native application compatibility to both the CPU and OS, together with execution isolation, which will deliver improved performance for CIAs. Bringing all of the various components or building blocks together, one can depict a summarised view of this new web browser EE, as per figure 4.15.

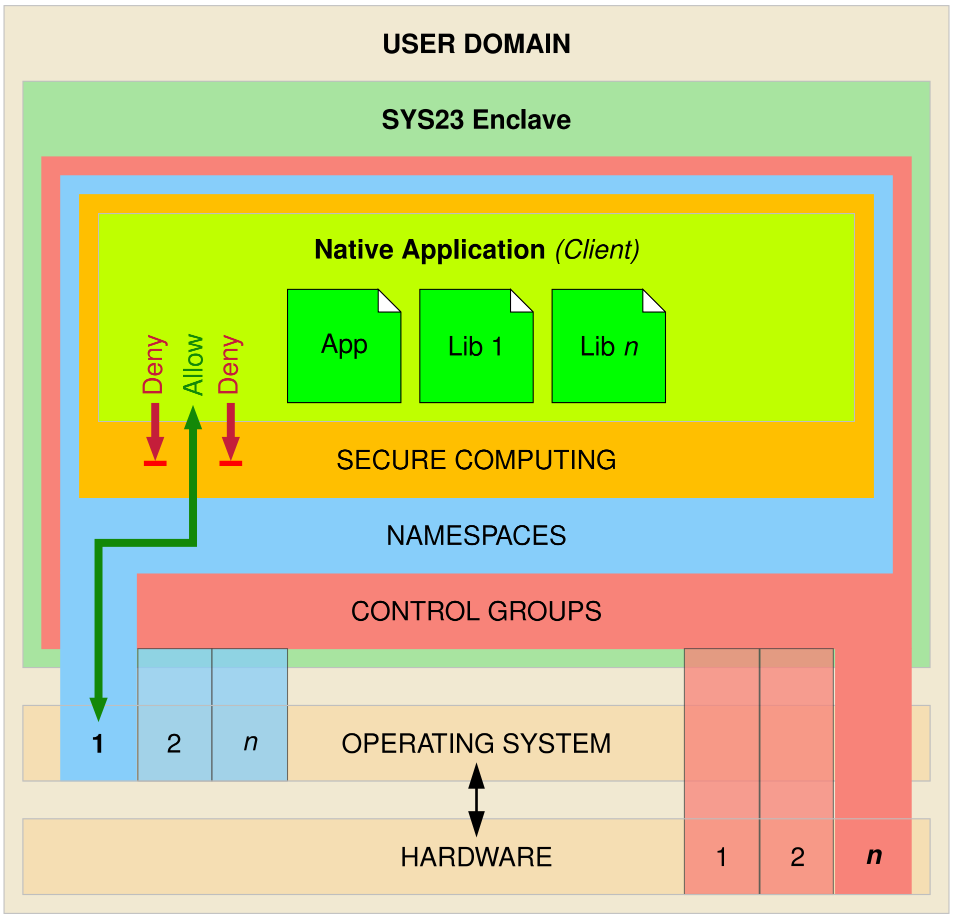

Figure 4.15: C4 Component (Level 3) Conceptual Design

Furthermore, given that the proposed architecture and design are based on existing architectural patterns and OS features, it allows for rapid and straightforward implementation within existing compatible devices. While also continuing to ensure that the user's devices that incorporate this architecture and design remain safe from malicious downloaded applications by ring-fencing the downloaded application.

We also briefly discussed the data collection design that utilises the PolyBench/C benchmarks and reasoned as to why its use as a benchmarking suite is acceptable, but also appropriate. Following that, we also briefly examined the aspect of data precision, the mathematical difference between low and high precision measurements, and concluding with how a high precision data collection implementation benefits this study and adds to the rigour of the presented findings.

In the next chapter, we will look at how the prototype performs against both native implementations of the benchmarking algorithms and implementations based on existing web browser EEs. A critical discussion is then undertaken based on the interpretation of the findings.

CHAPTER 5

FINDINGS AND DISCUSSION

This chapter evaluates the performance of the new web browser EEs by comparing it to existing ones using various metrics. This chapter also covers this research's objective four, where the researcher endeavours to:

Objective 4

Evaluate and benchmark the performance of the prototype web browser EE against those discovered through the SLR, thereby testing the empirical hypothesis of this study.

This chapter reports on the data collected by looking at the following:

Section 5.1. *Prototype evaluation*

Section 5.2. *Real-world evaluation*

Section 5.3. *Critical analysis*

Section 5.4. *Discussion*

Section 5.5. *Synopsis*

5.1. Prototype Evaluation

This section summarises the performance of the benchmarks when executed within the SYS23 prototype as compared to that of the native desktop. As expected, the native desktop outperformed the SYS23 prototype for the most part. However, a few surprising results of some benchmarks stand out, where the SYS23 performance for the *3mm*, *adi*, *jacobi-1d*, and *syrk* benchmarks outperformed their native equivalents. A later section ponders the possible reasons for these anomalies.

Table 5.1: System23 Prototype vs Native Benchmarks

Benchmark	System23	Native	Differential[1]	Speed Up
2mm	2.370166196 s	2.363493631 s	**-0.006672565 s**	-0.2823 %
3mm	4.012183164 s	4.012725930 s	**0.000542766 s**	$\boxed{0.0135\ \%}$
adi	9.960964131 s	9.961349930 s	**0.000385799 s**	$\boxed{0.0039\ \%}$
atax	0.005988229 s	0.005970464 s	**-0.000017765 s**	-0.2975 %
bicg	0.009938297 s	0.009923197 s	**-0.000015100 s**	-0.1522 %
cholesky	1.515222131 s	1.514085262 s	**-0.001136869 s**	-0.0751 %
correlation	5.860367264 s	5.843906095 s	**-0.016461169 s**	-0.2817 %
covariance	5.860343597 s	5.843348830 s	**-0.016994767 s**	-0.2908 %
deriche	0.242219765 s	0.240564130 s	**-0.001655635 s**	-0.6882 %
doitgen	0.549359296 s	0.548080830 s	**-0.001278466 s**	-0.2333 %
durbin	0.002974097 s	0.002968196 s	**-0.000005901 s**	-0.1988 %
fdtd-2d	1.404423562 s	1.401325230 s	**-0.003098332 s**	-0.2211 %
floyd-warshall	14.059311031 s	14.023624363 s	**-0.035686668 s**	-0.2545 %
gemm	0.623021030 s	0.620031664 s	**-0.002989366 s**	-0.4821 %
gemver	0.023476263 s	0.023426596 s	**-0.000049667 s**	-0.2120 %
gesummv	0.004354730 s	0.004346029 s	**-0.000008701 s**	-0.2002 %
gramschmidt	10.761396297 s	$\boxed{10.678539132\ s}$	**-0.082857165 s**	-0.7759 %
heat-3d	2.097793696 s	2.072167530 s	**-0.025626166 s**	-1.2367 %
jacobi-1d	0.000650531 s	0.000654328 s	**0.000003797 s**	$\boxed{0.5803\ \%}$
jacobi-2d	1.403825897 s	1.396950464 s	**-0.006875433 s**	-0.4922 %
lu	5.434344996 s	5.359398498 s	**-0.074946498 s**	-1.3984 %
ludcmp	5.216614332 s	5.174912364 s	**-0.041701968 s**	-0.8058 %
mvt	0.020398962 s	0.020365464 s	**-0.000033498 s**	-0.1645 %
nussinov	5.530813497 s	5.520036464 s	**-0.010777033 s**	-0.1952 %
seidel-2d	19.749028795 s	19.718044265 s	**-0.030984530 s**	-0.1571 %
symm	2.588202763 s	2.568486063 s	**-0.019716700 s**	-0.7676 %
syr2k	3.805913932 s	3.745486996 s	**-0.060426936 s**	-1.6133 %
syrk	0.831422029 s	0.840600996 s	**0.009178967 s**	$\boxed{1.0920\ \%}$
trisolv	0.002476862 s	0.002464864 s	**-0.000011998 s**	-0.4868 %
trmm	2.400019431 s	2.398117464 s	**-0.001901967 s**	-0.0793 %

Execution time is measured in seconds to within a nanosecond.
[1]The differential between the *SYS23 prototype* and the *native* benchmarks.

5.2. Real-world evaluation

This section summarises the performance of the benchmarks when executed within the SYS23 prototype as compared to that of WASM. As expected, the SYS23 prototype outperformed WASM for the most part. However, one surprising result of the benchmarks stands out, where the WASM performance for the *gramschmidt* benchmark outperformed both the SYS23 and native equivalents. A later section considers the possible reasons for these anomalies.

Table 5.2: System23 Prototype vs Real-World Benchmarks

Benchmark	**System23**	**WASM**	**Differential**[1]	**Speed Up**
2mm	2.370166196 s	5.169499997 s	**2.799333801 s**	54.1510 %
3mm	4.012183164 s	8.291966664 s	**4.279783500 s**	51.6136 %
adi	9.960964131 s	12.045733330 s	**2.084769199 s**	17.3071 %
atax	0.005988229 s	0.012099999 s	**0.006111770 s**	50.5105 %
bicg	0.009938297 s	0.014033333 s	**0.004095036 s**	29.1808 %
cholesky	1.515222131 s	2.726199996 s	**1.210977865 s**	44.4200 %
correlation	5.860367264 s	6.405766663 s	**0.545399399 s**	8.5142 %
covariance	5.860343597 s	6.259666664 s	**0.399323067 s**	6.3793 %
deriche	0.242219765 s	0.283299997 s	**0.041080232 s**	14.5006 %
doitgen	0.549359296 s	1.092633329 s	**0.543274033 s**	49.7215 %
durbin	0.002974097 s	0.007466664 s	**0.004492567 s**	60.1683 %
fdtd-2d	1.404423562 s	4.351133329 s	**2.946709767 s**	67.7228 %
floyd-warshall	14.059311031 s	46.623133330 s	**32.563822299 s**	69.8448 %
gemm	0.623021030 s	2.575133330 s	**1.952112300 s**	75.8063 %
gemver	0.023476263 s	0.036733330 s	**0.013257067 s**	36.0900 %
gesummv	0.004354730 s	0.006000000 s	**0.001645270 s**	27.4212 %
gramschmidt	10.761396297 s	$\boxed{10.300333331\ \text{s}}$	**-0.461062966 s**	$\boxed{-4.4762\ \%}$
heat-3d	2.097793696 s	9.462366665 s	**7.364572969 s**	77.8301 %
jacobi-1d	0.000650531 s	0.003699998 s	**0.003049467 s**	82.4181 %
jacobi-2d	1.403825897 s	5.507033330 s	**4.103207433 s**	74.5085 %
lu	5.434344996 s	9.728866664 s	**4.294521668 s**	44.1421 %
ludcmp	5.216614332 s	9.064333330 s	**3.847718998 s**	42.4490 %
mvt	0.020398962 s	0.027066666 s	**0.006667704 s**	24.6344 %
nussinov	5.530813497 s	11.583899996 s	**6.053086499 s**	52.2543 %
seidel-2d	19.749028795 s	23.018199997 s	**3.269171202 s**	14.2025 %
symm	2.588202763 s	4.407899996 s	**1.819697233 s**	41.2826 %
syr2k	3.805913932 s	5.991033331 s	**2.185119399 s**	36.4732 %
syrk	0.831422029 s	2.552399997 s	**1.720977968 s**	67.4259 %
trisolv	0.002476862 s	0.004033333 s	**0.001556471 s**	38.5902 %
trmm	2.400019431 s	2.842799998 s	**0.442780567 s**	15.5755 %

Execution time is measured in seconds to within a nanosecond.
[1]The differential between the *SYS23 prototype* and the *WASM* benchmarks.

5.3. Critical analysis

5.3.1. Data mining

5.3.1.1. correlation

The *correlation* benchmark as depicted in figure 5.1 calculates the correlation coefficients (Pearson's) representing the normalised form of covariance (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A minor, within 20% performance improvement of the SYS23 benchmark over that of the WASM one.

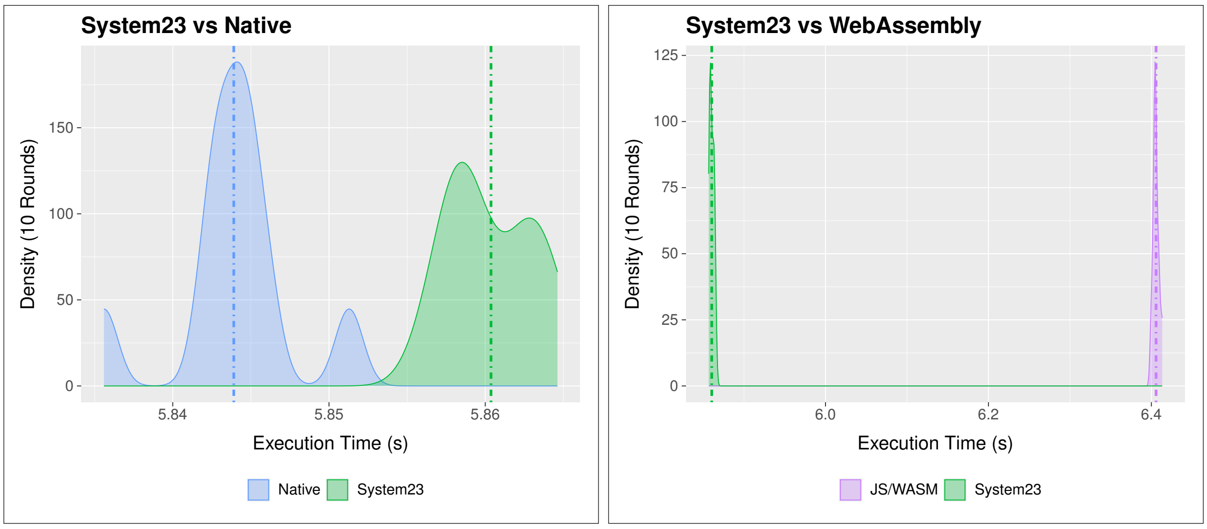

Figure 5.1: correlation Benchmarks

5.3.1.2. covariance

The *covariance* benchmark as depicted in figure 5.2 calculates the covariance, which is a statistical measure that indicates the degree to which two variables are linearly related (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A minor, within 20% performance improvement of the SYS23 benchmark over that of the WASM one.

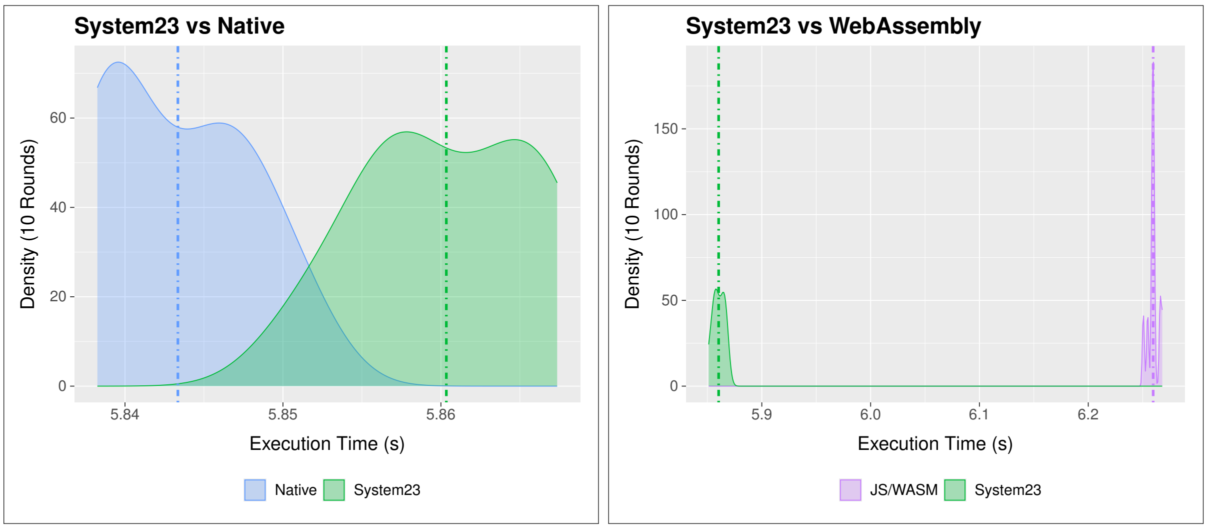

Figure 5.2: covariance Benchmarks

5.3.2. Basic linear algebra

5.3.2.1. gemm

The *gemm* benchmark as depicted in figure 5.3 is a generalised matrix multiplication calculation (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A significant, within 90% performance improvement of the SYS23 benchmark over that of the WASM one.

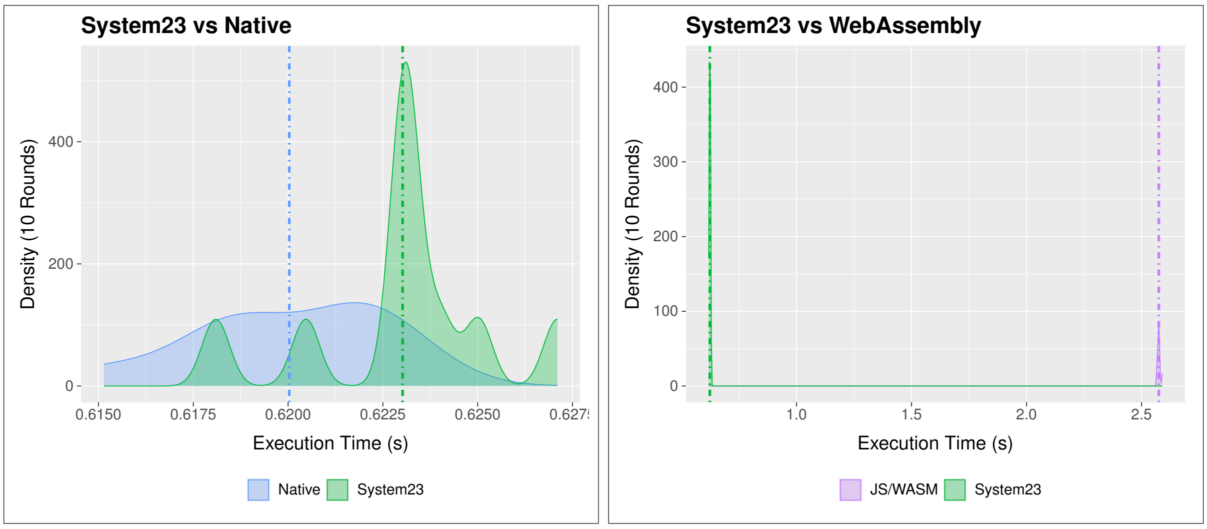

Figure 5.3: gemm Benchmarks

5.3.2.2. gemver

The *gemver* benchmark as depicted in figure 5.4 is a multiple matrix-vector multiplication calculation (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A substantial, within 50% performance improvement of the SYS23 benchmark over that of the WASM one.

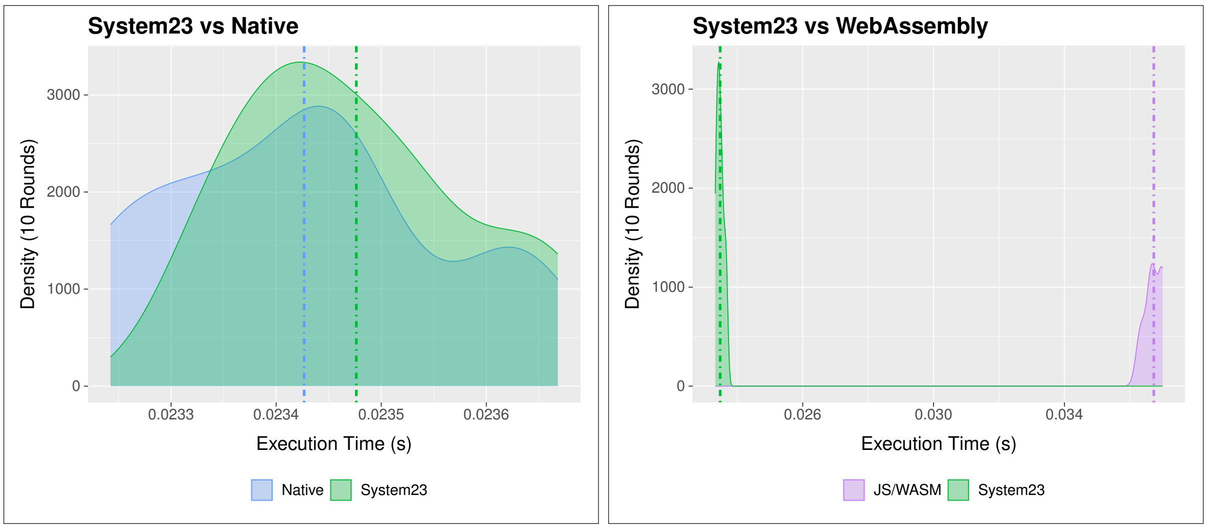

Figure 5.4: gemver Benchmarks

5.3.2.3. gesummv

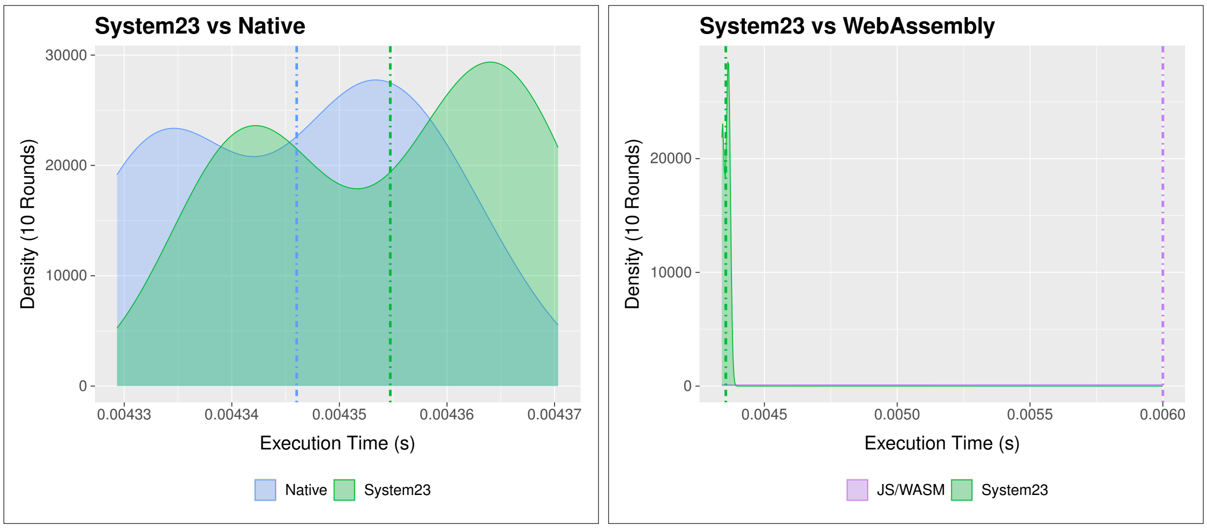

Figure 5.5: gesummv Benchmarks

The *gesummv* benchmark as depicted in figure 5.5 is a summed matrix-vector multiplication calculation (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A substantial, within 50% performance improvement of the SYS23 benchmark over that of the WASM one.

5.3.2.4. symm

The *symm* benchmark as depicted in figure 5.6 is a symmetric matrix multiplication calculation (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A substantially close performance result of less than 1% of the SYS23 benchmark to that of the native one.
- A substantial, within 50% performance improvement of the SYS23 benchmark over that of the WASM one.

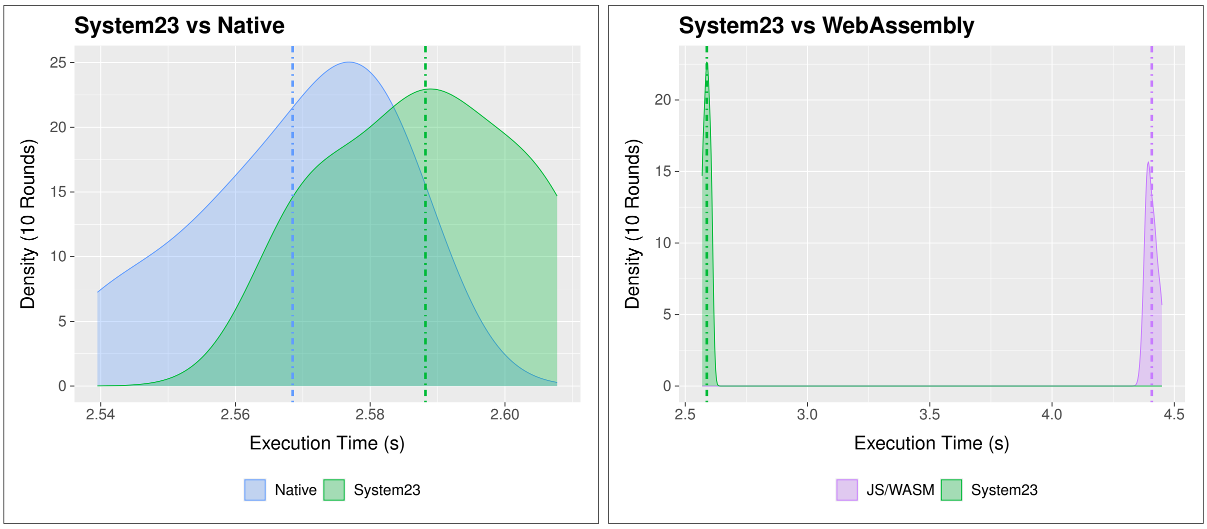

Figure 5.6: symm Benchmarks

5.3.2.5. syr2k

The *syr2k* benchmark as depicted in figure 5.7 is a symmetric rank $2k$ update calculation (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A minorly close performance result of less than 2% of the SYS23 benchmark to that of the native one.

- A substantial, within 50% performance improvement of the SYS23 benchmark over that of the WASM one.

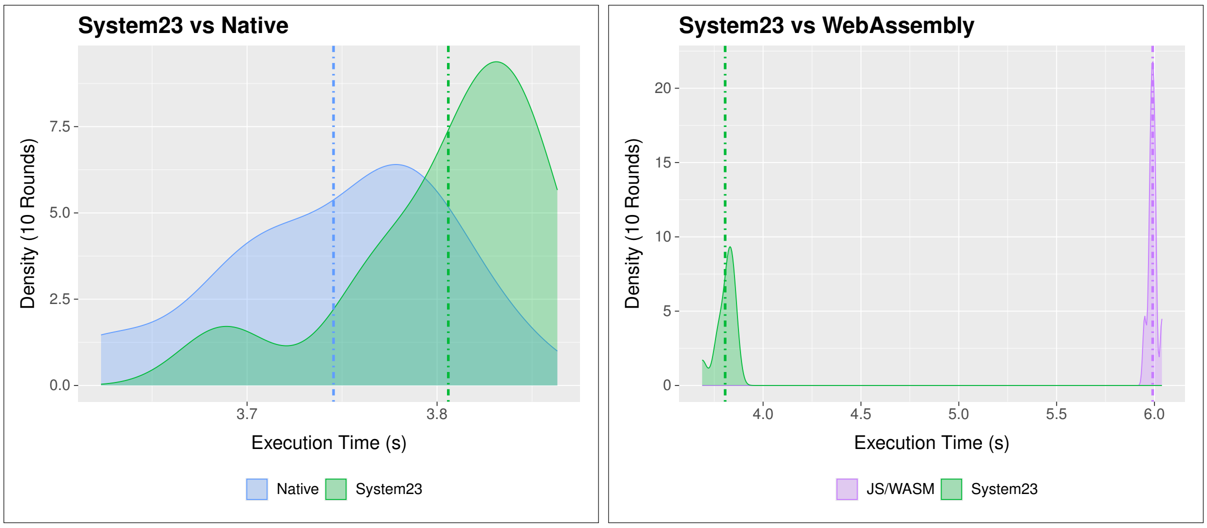

Figure 5.7: syr2k Benchmarks

5.3.2.6. syrk

The *syrk* benchmark as depicted in figure 5.8 is a symmetric rank k update calculation (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- An interestingly reverse result, whereby the SYS23 benchmark outperformed that of the native one.
- A significant, within 90% performance improvement of the SYS23 benchmark over that of the WASM one.

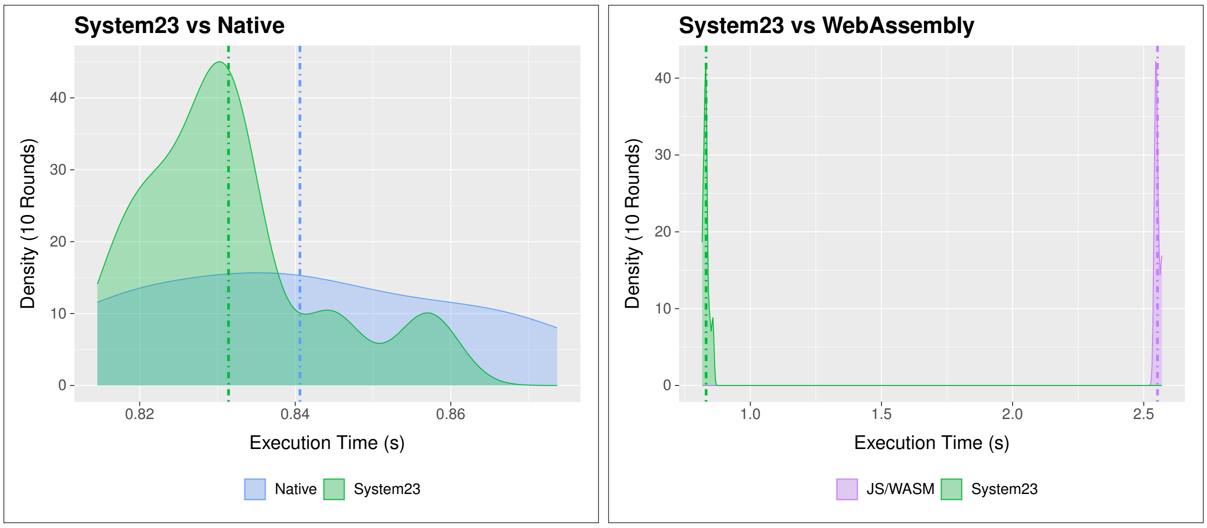

Figure 5.8: syrk Benchmarks

5.3.2.7. trmm

The *trmm* benchmark as depicted in figure 5.9 is a triangular matrix multiplication calculation (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A minor, within 20% performance improvement of the SYS23 benchmark over that of the WASM one.

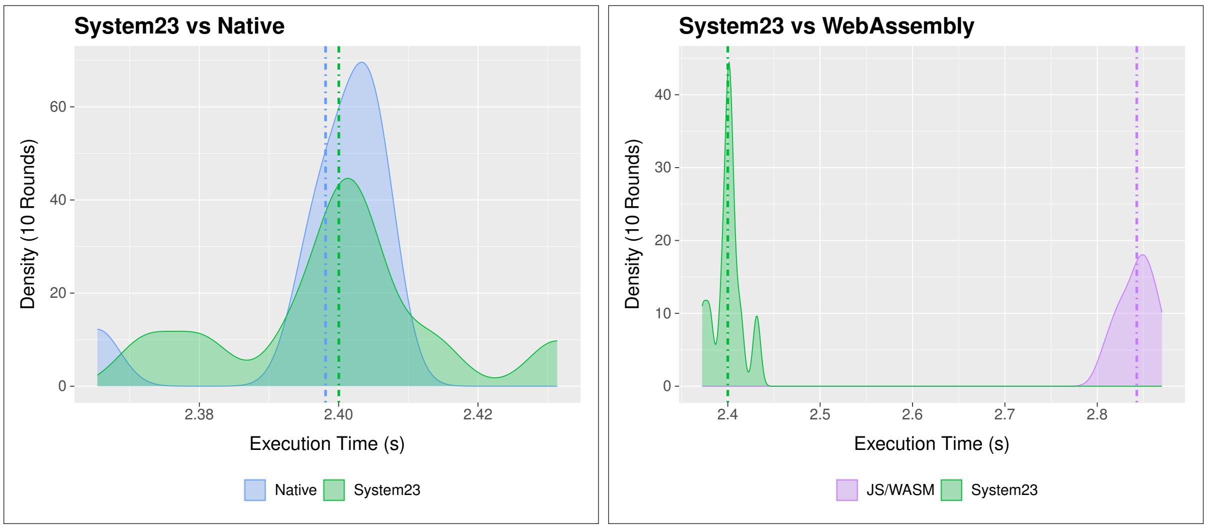

Figure 5.9: trmm Benchmarks

5.3.3. Linear algebra transform

5.3.3.1. 2mm

The *2mm* benchmark as depicted in figure 5.10 is a linear algebra transform operation that involves performing two matrix multiplications (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A significant, within 90% performance improvement of the SYS23 benchmark over that of the WASM one.

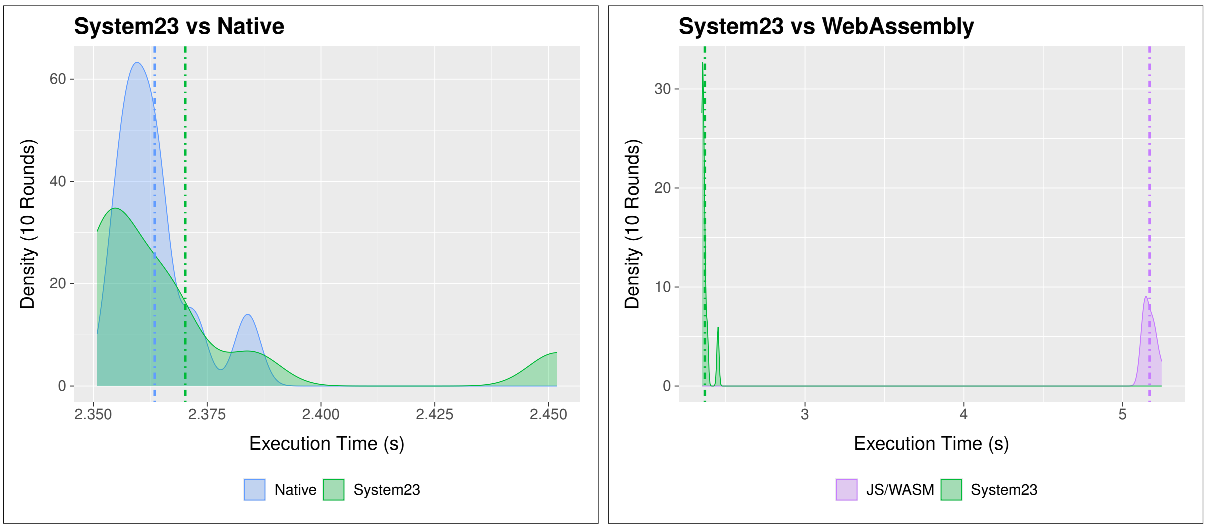

Figure 5.10: 2mm Benchmarks

5.3.3.2. 3mm

The *3mm* benchmark as depicted in figure 5.11 is another linear algebra transform operation that involves performing three matrix multiplications (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- An almost even result, whereby the SYS23 benchmark is almost exactly the same as the native one.
- A significant, within 90% performance improvement of the SYS23 benchmark over that of the WASM one.

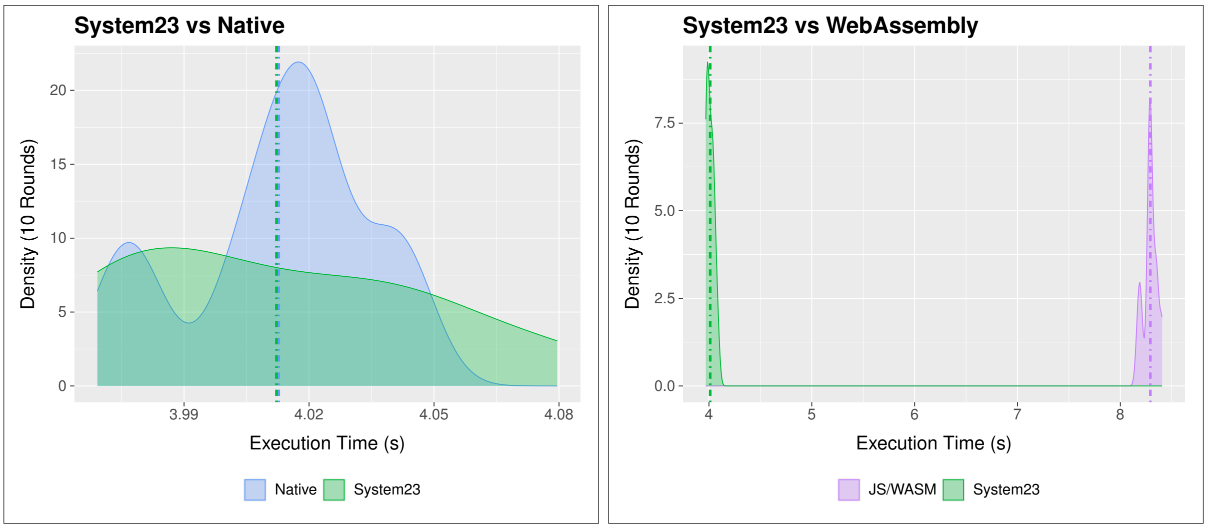

Figure 5.11: 3mm Benchmarks

5.3.3.3. atax

The *atax* benchmark as depicted in figure 5.12 calculates $A^T \times Ax$ (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A significant, within 90% performance improvement of the SYS23 benchmark over that of the WASM one.

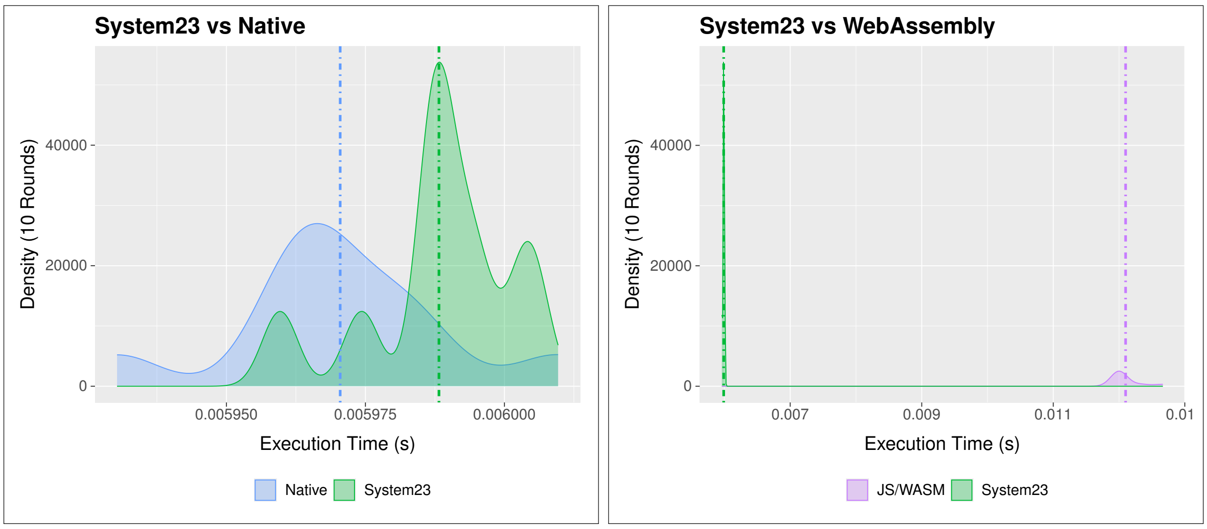

Figure 5.12: atax Benchmarks

5.3.3.4. bicg

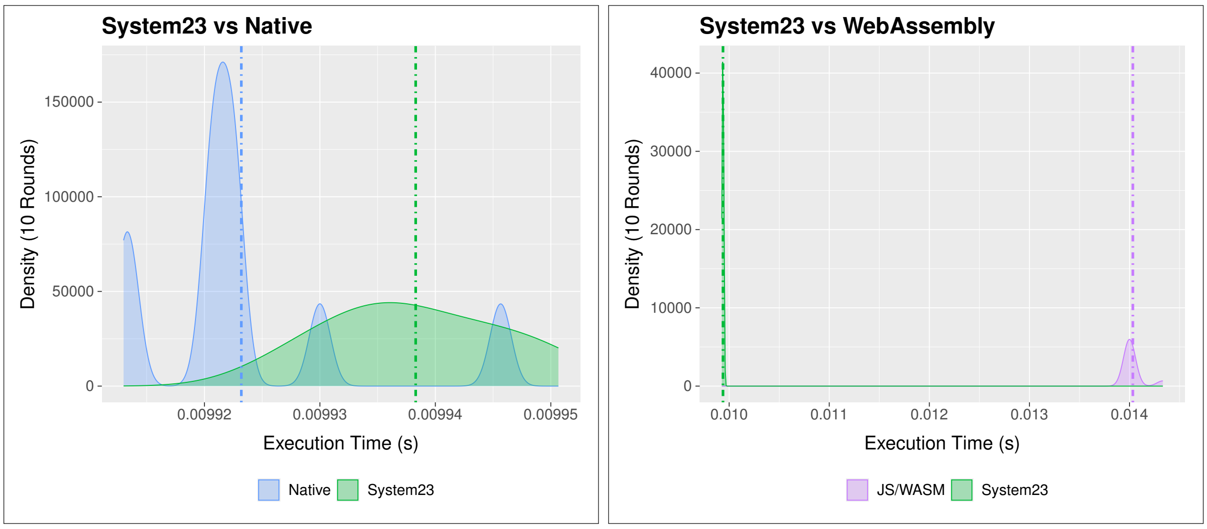

Figure 5.13: bicg Benchmarks

The *bicg* benchmark as depicted in figure 5.13 is a calculation of the *BiConjugate Gradient STABilized* method (BiCGSTAB) (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A substantial, within 50% performance improvement of the SYS23 benchmark over that of the WASM one.

5.3.3.5. doitgen

The *doitgen* benchmark as depicted in figure 5.14 is a calculation of the *Multiresolution ADaptive NumErical Scientific Simulation* (MADNESS) (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A substantial, within 50% performance improvement of the SYS23 benchmark over that of the WASM one.

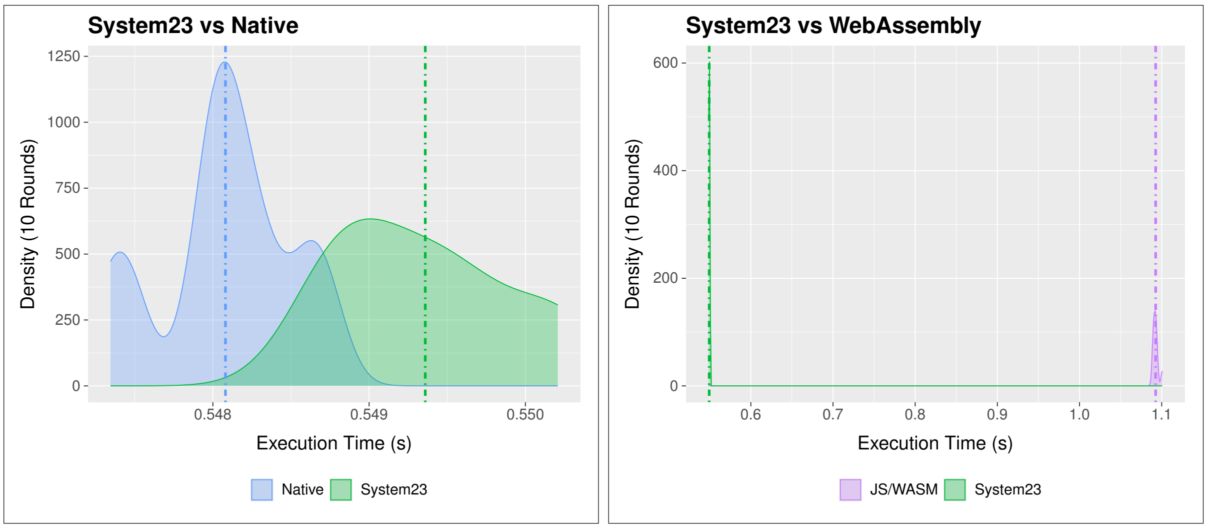

Figure 5.14: doitgen Benchmarks

5.3.3.6. mvt

The *mvt* benchmark as depicted in figure 5.15 calculates the matrix vector multiplication followed by another matrix vector multiplication using the transposed matrix (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A substantial, within 50% performance improvement of the SYS23 benchmark over that of the WASM one.

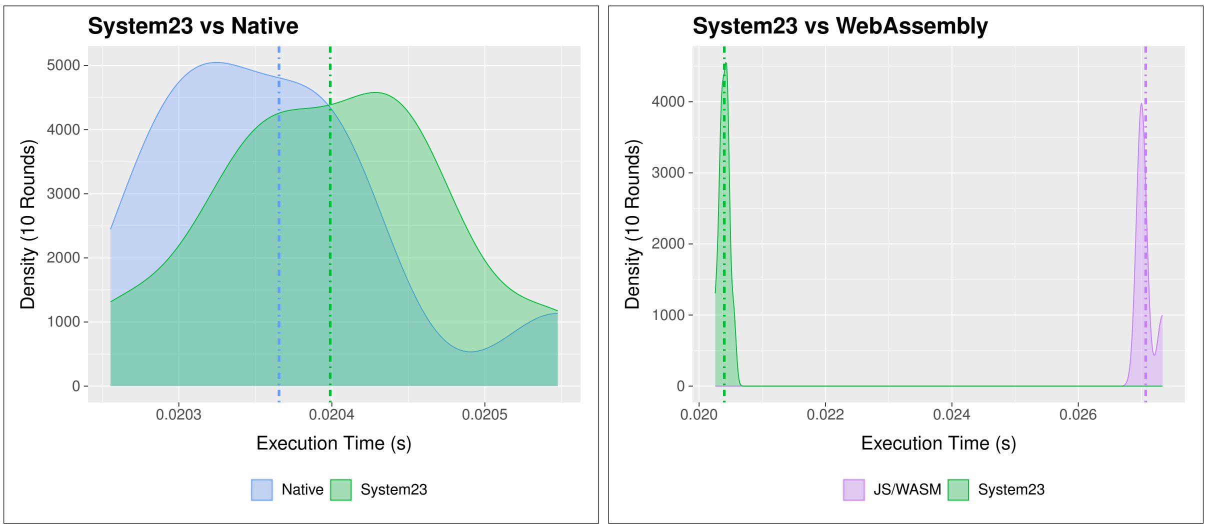

Figure 5.15: mvt Benchmarks

5.3.4. Linear algebra solver

5.3.4.1. cholesky

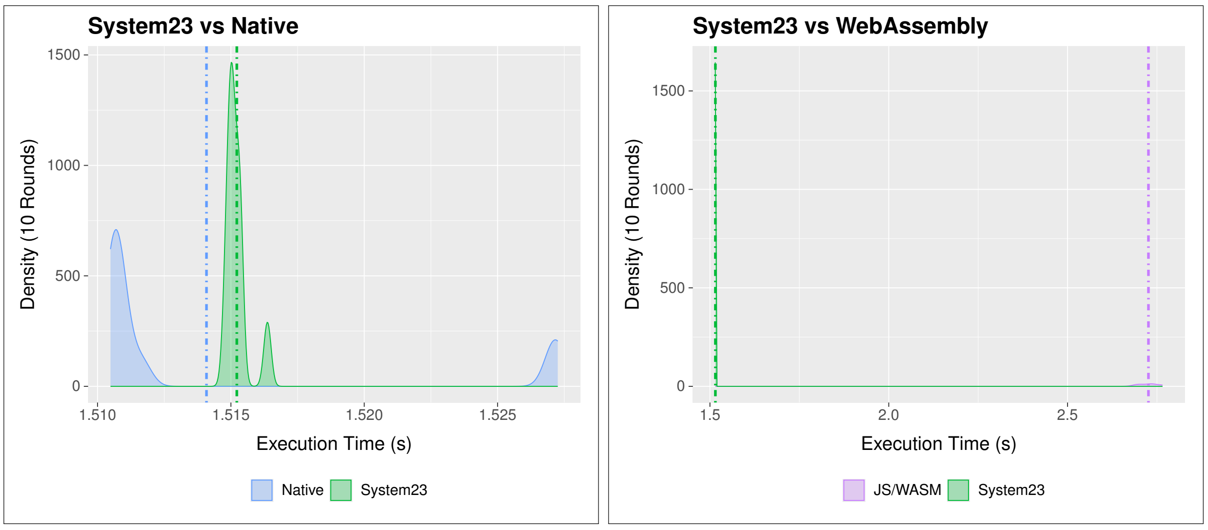

Figure 5.16: cholesky Benchmarks

The *cholesky* benchmark as depicted in figure 5.16 is an algorithm that decomposes a matrix into triangular matrices (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A substantial, within 50% performance improvement of the SYS23 benchmark over that of the WASM one.

5.3.4.2. durbin

The *durbin* benchmark as depicted in figure 5.17 is a special case of *Toelitz* systems that utilises an algorithm for solving *Yule-Walker* equations (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A significant, within 90% performance improvement of the SYS23 benchmark over that of the WASM one.

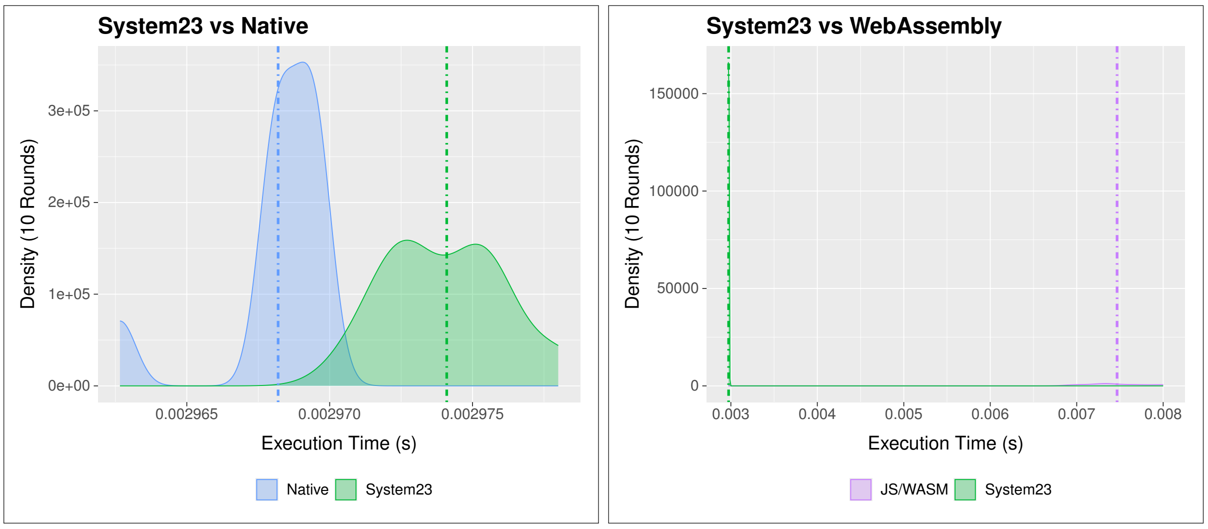

Figure 5.17: durbin Benchmarks

5.3.4.3. gramschmidt

The *gramschmidt* benchmark as depicted in figure 5.18 calculates QR decomposition with *Modified Gram Schmidt* (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A substantially close performance result of less than 1% of the SYS23 benchmark to that of the native one.

- An interestingly reverse result, whereby the WASM benchmark outperformed that of the SYS23 one.

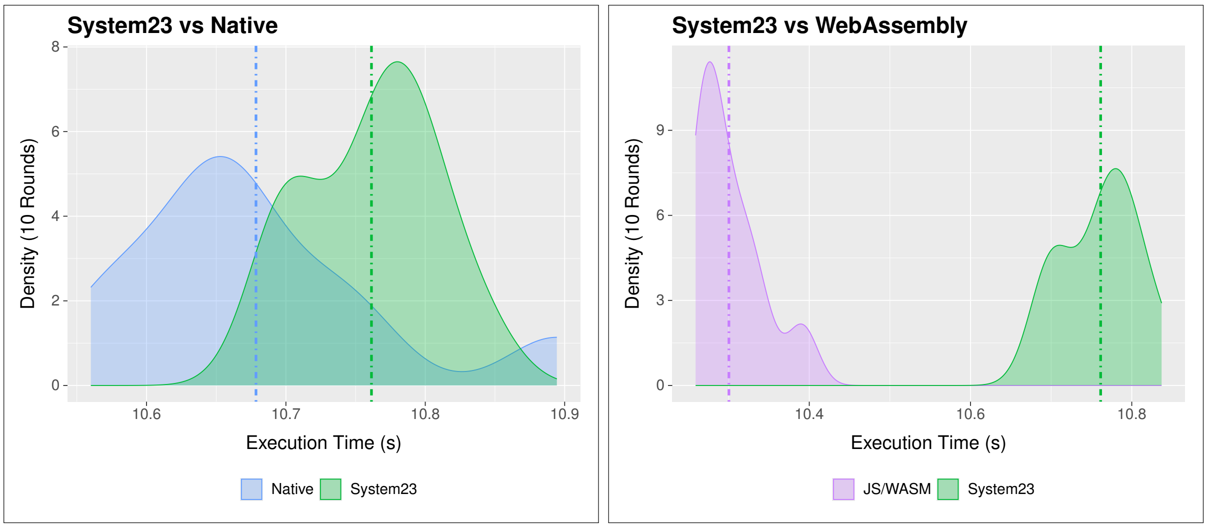

Figure 5.18: gramschmidt Benchmarks

5.3.4.4. lu

The *lu* benchmark as depicted in figure 5.19 calculates LU decomposition without pivoting (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A minorly close performance result of less than 2% of the SYS23 benchmark to that of the native one.
- A substantial, within 50% performance improvement of the SYS23 benchmark over that of the WASM one.

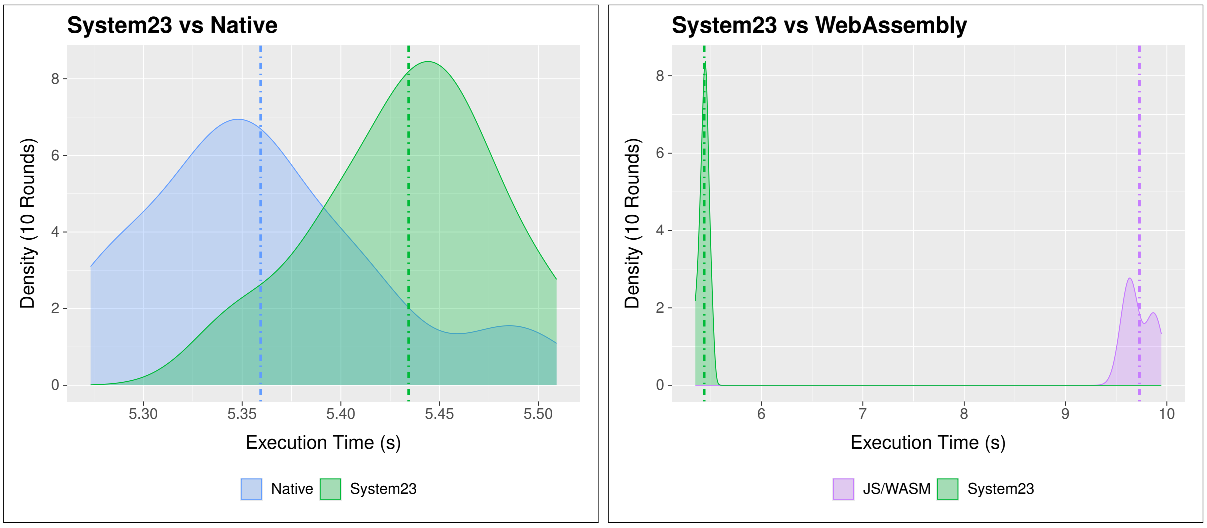

Figure 5.19: lu Benchmarks

5.3.4.5. ludcmp

The *ludcmp* benchmark as depicted in figure 5.20 is a system of linear equations using LU decomposition followed by forward and backward substitutions (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A substantially close performance result of less than 1% of the SYS23 benchmark to that of the native one.
- A substantial, within 50% performance improvement of the SYS23 benchmark over that of the WASM one.

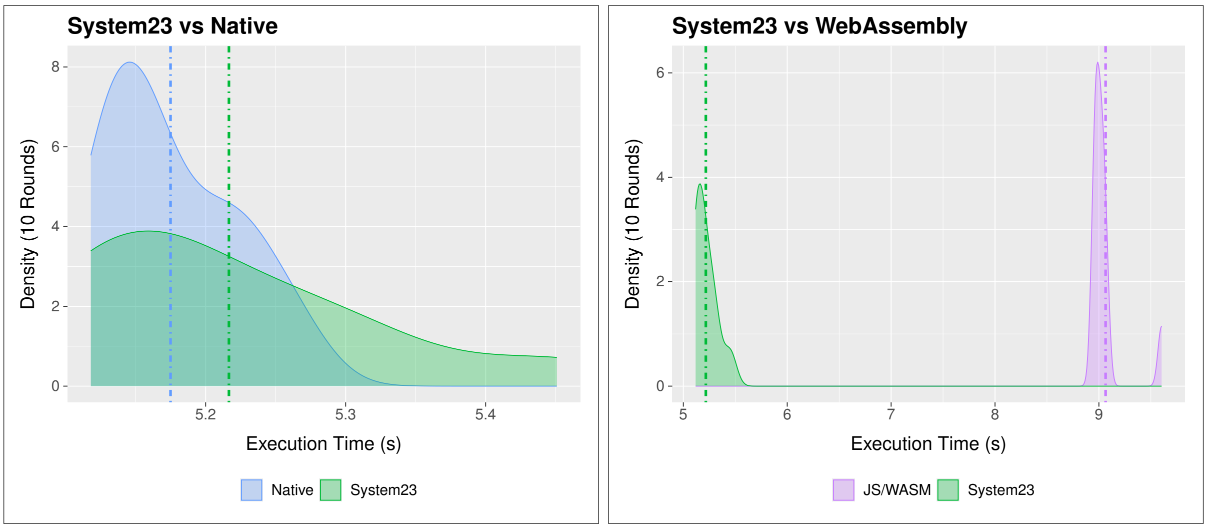

Figure 5.20: ludcmp Benchmarks

5.3.4.6. trisolv

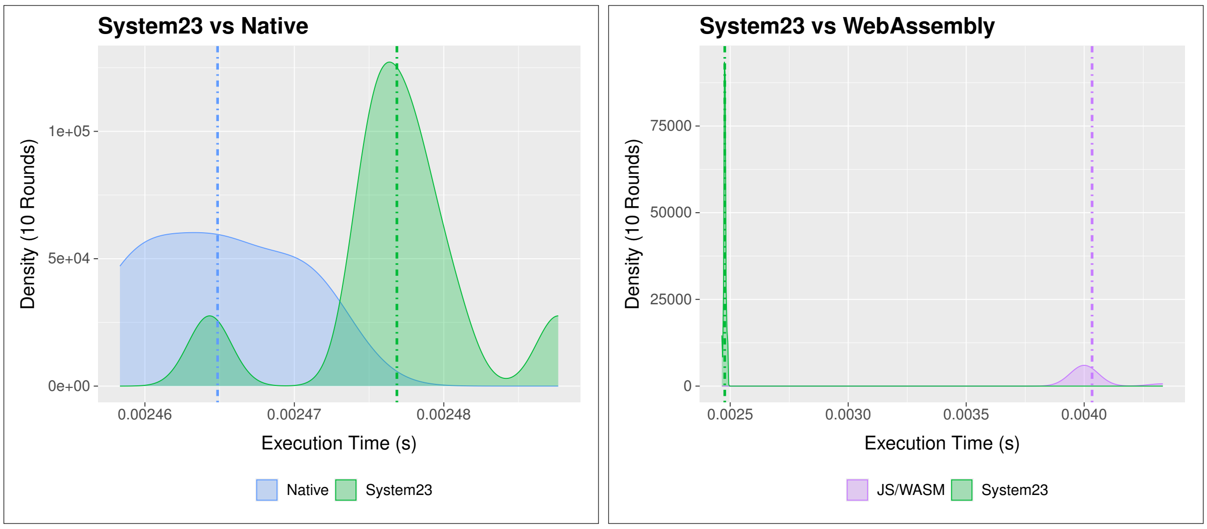

Figure 5.21: trisolv Benchmarks

The *trisolv* benchmark as depicted in figure 5.21 is a triangular matrix solver using forward substitution (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A substantial, within 50% performance improvement of the SYS23 benchmark over that of the WASM one.

5.3.5. Medley

5.3.5.1. deriche

The *deriche* benchmark as depicted in figure 5.22 is an implementation of the *Deriche* recursive filter, which is a generic filter that can be used for both smoothing and edge detection (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A substantially close performance result of less than 1% of the SYS23 benchmark to that of the native one.
- A minor, within 20% performance improvement of the SYS23 benchmark over that of the WASM one.

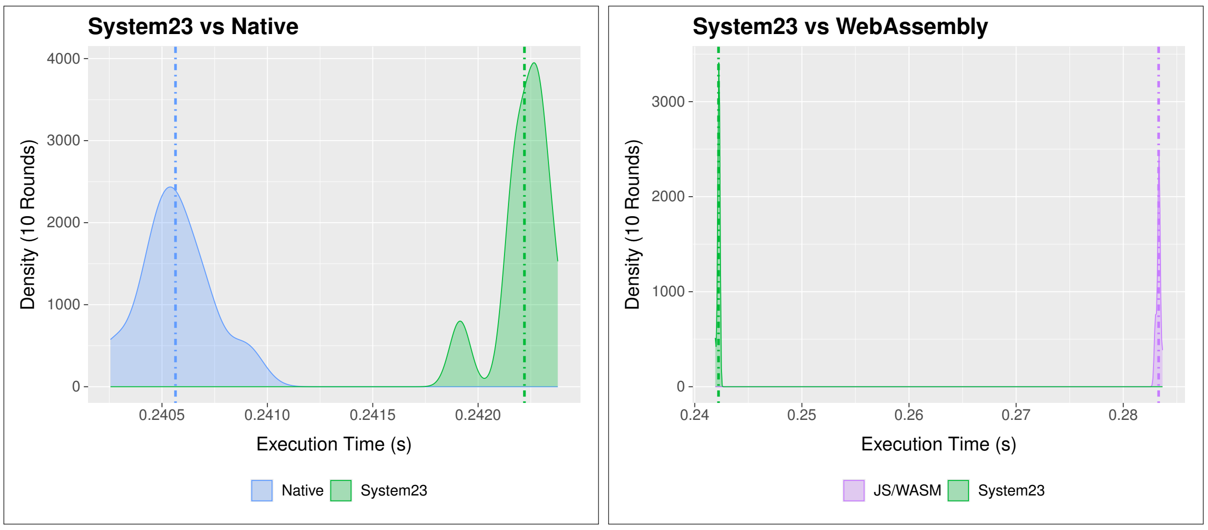

Figure 5.22: deriche Benchmarks

5.3.5.2. floyd-warshall

The *floyd-warshall* benchmark as depicted in figure 5.23 calculates the shortest paths between each pair of nodes in a graph (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A significant, within 90% performance improvement of the SYS23 benchmark over that of the WASM one.

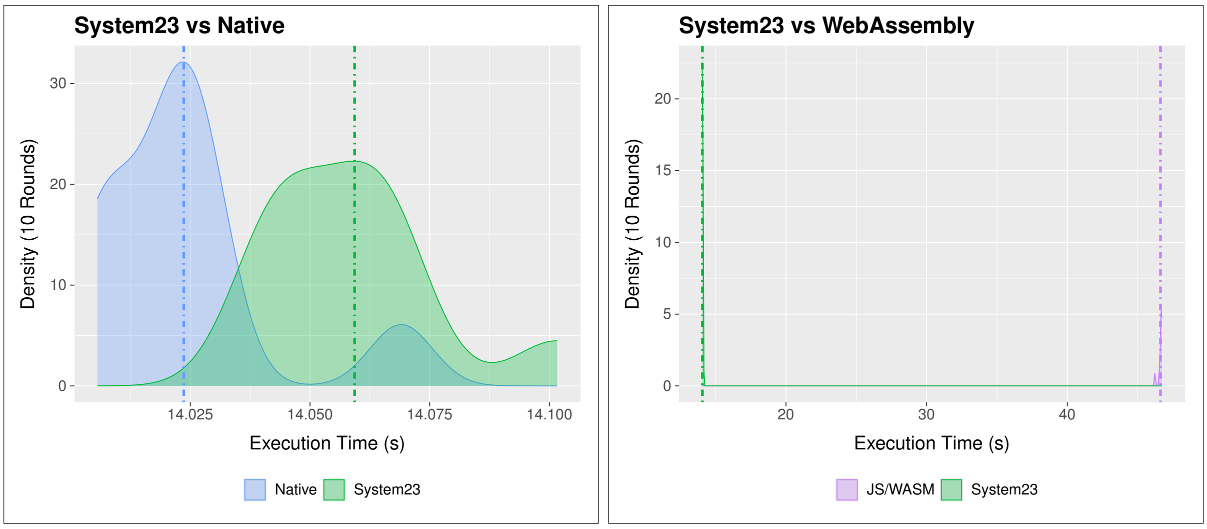

Figure 5.23: floyd-warshall Benchmarks

5.3.5.3. nussinov

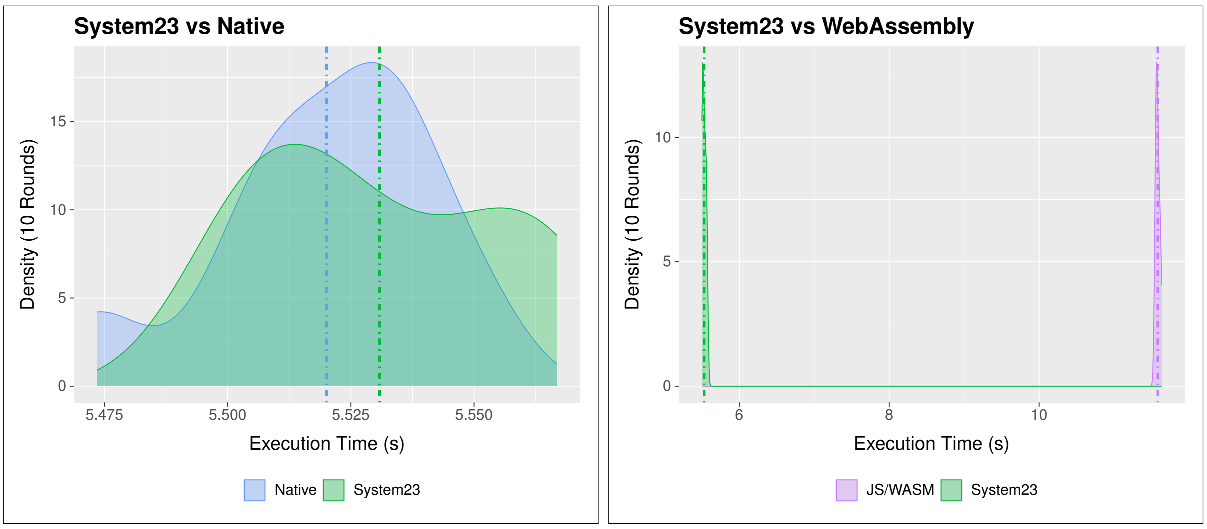

Figure 5.24: nussinov Benchmarks

The *nussinov* benchmark as depicted in figure 5.24 is an instance of dynamic programming where the algorithm is used for predicting RNA folding (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A significant, within 90% performance improvement of the SYS23 benchmark over that of the WASM one.

5.3.6. Stencils

5.3.6.1. adi

The *adi* benchmark as depicted in figure 5.25 is a calculation of the *Alternating Direction Implicit* method for two-dimensional heat diffusion (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- An almost even result, whereby the SYS23 benchmark is almost exactly the same as the native one.
- A minor, within 20% performance improvement of the SYS23 benchmark over that of the WASM one.

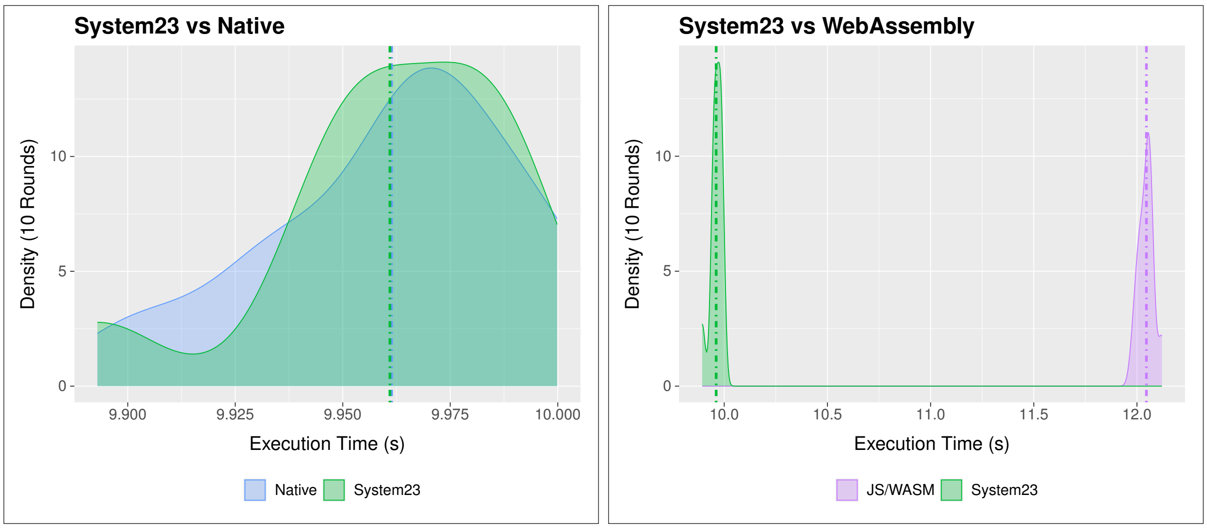

Figure 5.25: adi Benchmarks

5.3.6.2. fdtd-2d

The *fdtd-2d* benchmark as depicted in figure 5.26 is a calculation of the Simplified *Finite-Difference Time-Domain* method for two-dimensional data, which based on Maxwell's equation, models electric and magnetic fields (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.

- A significant, within 90% performance improvement of the SYS23 benchmark over that of the WASM one.

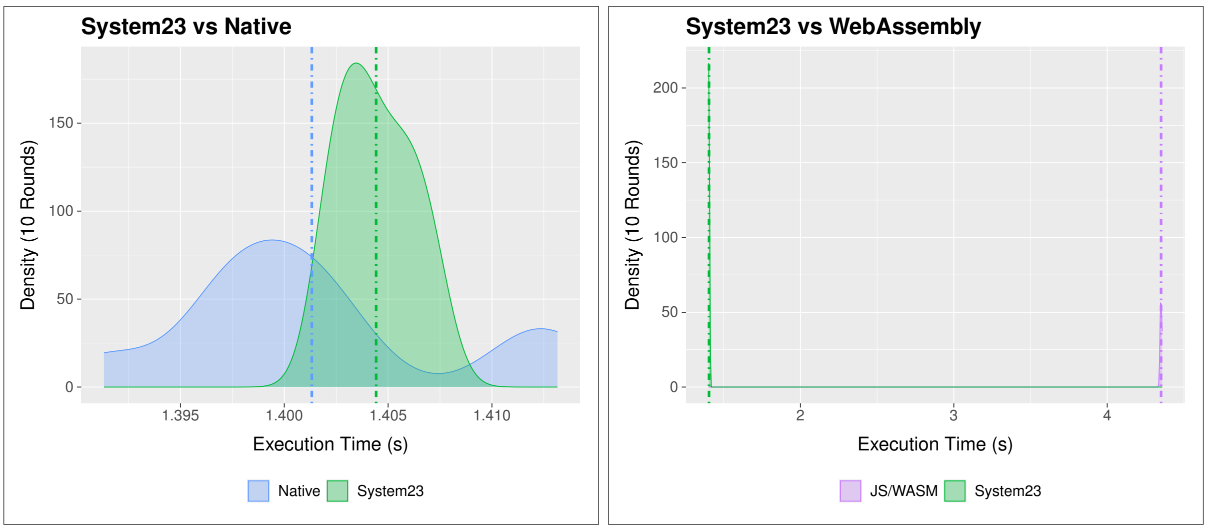

Figure 5.26: fdtd-2d Benchmarks

5.3.6.3. heat-3d

The *heat-3d* benchmark as depicted in figure 5.27 is a calculation using the *Heat* equation over three-dimensional space (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A minorly close performance result of less than 2% of the SYS23 benchmark to that of the native one.
- A significant, within 90% performance improvement of the SYS23 benchmark over that of the WASM one.

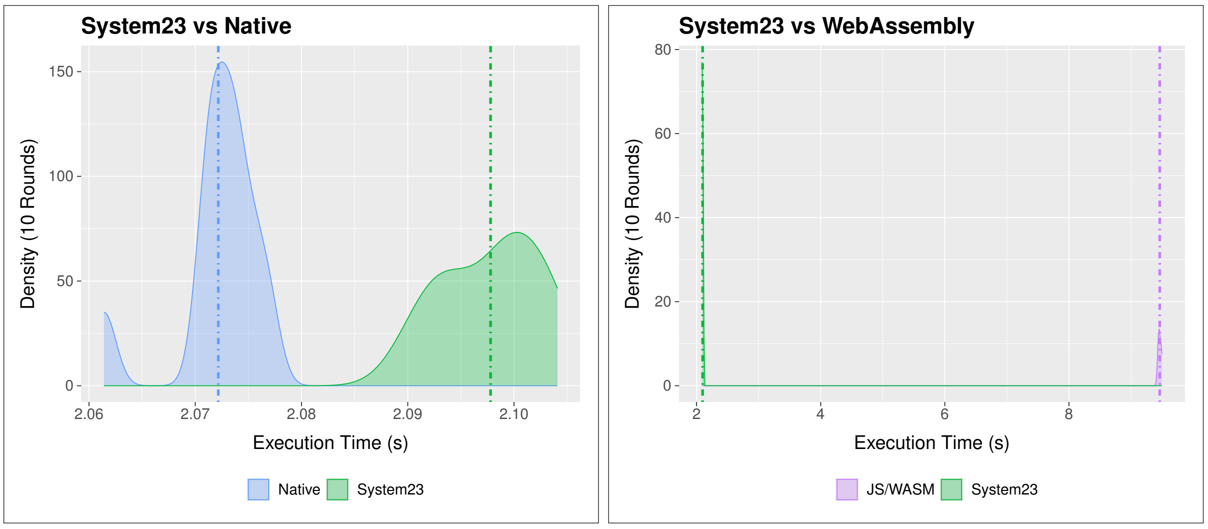

Figure 5.27: heat-3d Benchmarks

5.3.6.4. jacobi-1d

The *jacobi-1d* benchmark as depicted in figure 5.28 is a Jacobi-style stencil computation over one-dimensional data using a three-point stencil pattern (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- An interestingly reverse result, whereby the SYS23 benchmark outperformed that of the native one.
- A significant, within 90% performance improvement of the SYS23 benchmark over that of the WASM one.

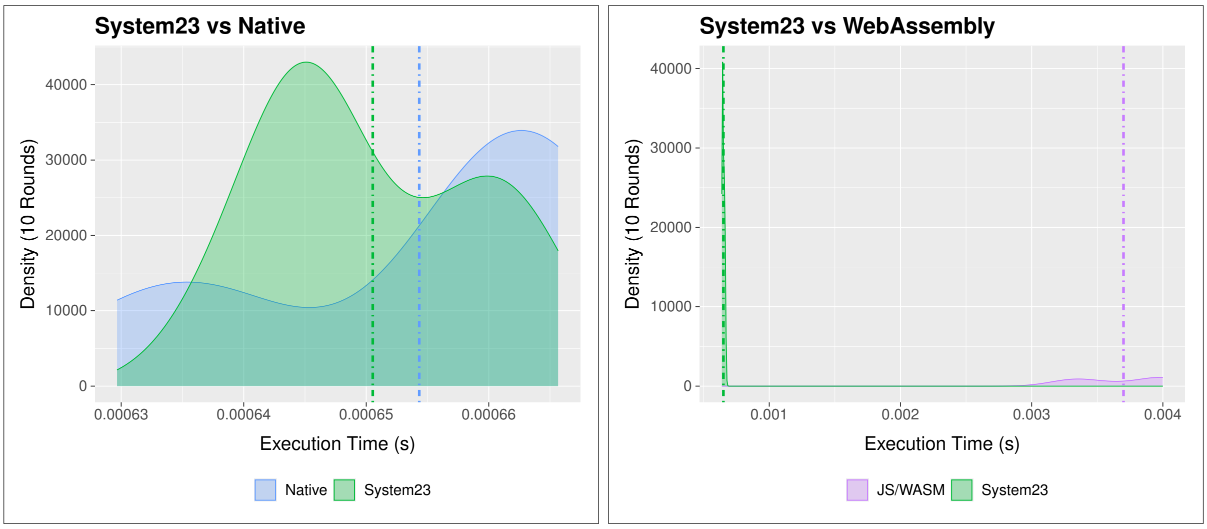

Figure 5.28: jacobi-1d Benchmarks

5.3.6.5. jacobi-2d

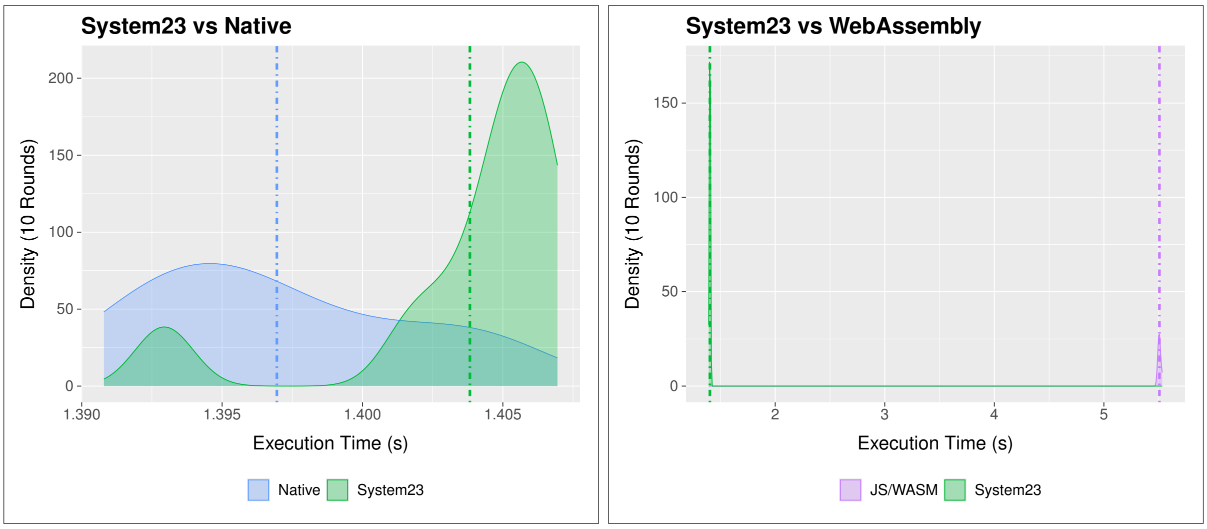

Figure 5.29: jacobi-2d Benchmarks

The *jacobi-2d* benchmark as depicted in figure 5.29 is a Jacobi-style stencil computation over two-dimensional data using a five-point stencil pattern (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A significant, within 90% performance improvement of the SYS23 benchmark over that of the WASM one.

5.3.6.6. seidel-2d

The *seidel-2d* benchmark as depicted in figure 5.30 is a Gauss-Seidel style stencil computation over two-dimensional data using a nine-point stencil pattern. Similar to Jacobi-style stencils, the Gauss-Seidel style stencil is derived from the Gauss-Seidel method for solving systems of linear equations (Yuki & Pouchet, 2016). The benchmarks have the following characteristics depicted in green:

- A significantly close performance result of less than 0.5% of the SYS23 benchmark to that of the native one.
- A minor, within 20% performance improvement of the SYS23 benchmark over that of the WASM one.

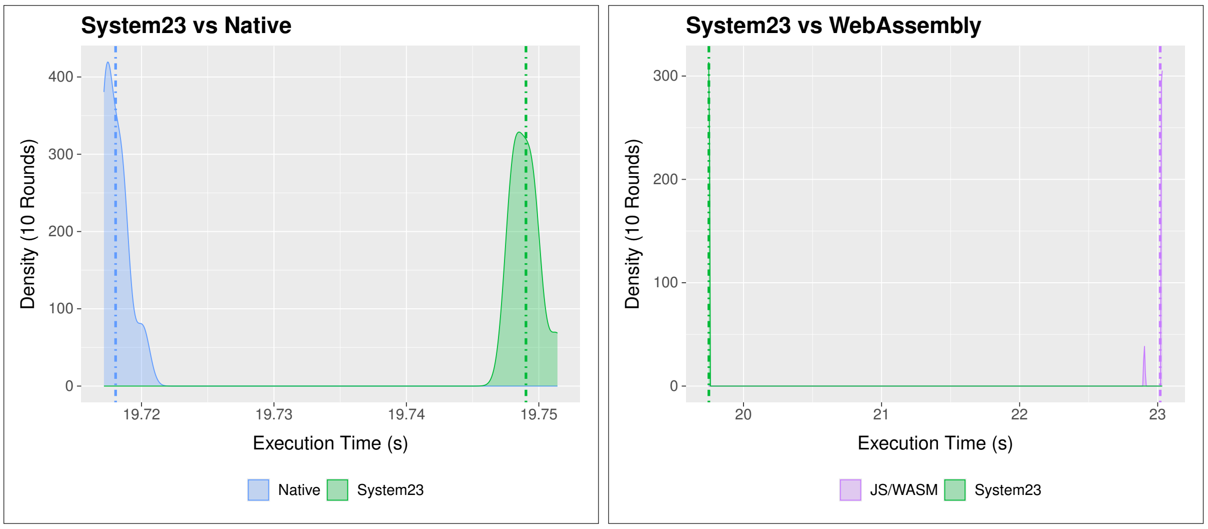

Figure 5.30: seidel-2d Benchmarks

5.4. Discussion

At the start of this study, we stated that the goal or overall objective was to develop a new web browser EE, which can be used by any web browser to host and execute CIAs with native desktop performance. In conjunction with that goal, this study endeavoured to answer a singular research question using the framework proposed by Kofod-Petersen (2015). That singular research question was:

Research Question

What software architecture and design does a new web browser EE need to comprise of in order to be able to host and execute CIAs with native desktop performance?

Together with the research question, a hypothesis was formed which stated that:

Hypothesis

A web browser EE that can host and execute CIAs on any device with native desktop performance can be created.

In the broader scope of our research, being able to derive a solution that could aid in the execution of CIAs with native desktop performance, while also being able to embed it within a web browser, would allow our hypothesis to be scientifically tested. Furthermore the results of this hypothesis test would demonstrate whether our hypothesis is or is not supported by the evidence, as was generated by our benchmarking.

Let's now review the key findings of not just objective four, but all four objectives of this study.

Objective 1

Identify the features and limitations of existing web browser EEs that can host and execute applications.

While working through objective one, we compiled an extensive list of literature that dealt with the performance of web browser EEs. From that list of literature we were able to identify all of the historical and existing web browser EEs, as well as the unique features of each. Furthermore, we were also able to ascertain the limitations of each, especially where it pertained to the aspect of performance, relative to that of a native desktop.

We were then able to draw inferences from the primary literature, which indicated that further investigation into web browser EEs was warranted. Together with that, we were also able to identify where the most rewarding opportunities were for improvements to the performance of web browser EEs. Those opportunities revolved around finding a safe way to execute natively compatible applications within a web browser.

To that end, we then focussed our efforts on experimenting with innovative ways of executing native applications within a web browser, while ensuring that the application could not harm the rest of the computer system. In turn, given the high-performance capabilities of executing native applications, we would be able to meet and even possibly exceed the *Doherty Threshold*.

Objective 2

Employ a suitable research methodology by determining which ones are best aligned to deliver on the requirements and goal of this study.

While deriving a suitable methodology that this study could be grounded on that would cover objective two, we again used the learnings from objective one. We analysed the literature to understand what methodologies those other studies used and then proceeded to formulate a set of methodologies that would best suit this study.

While compiling a well-aligned set of methodologies for this study, we also became aware of other aspects of the methodology that may need refining, such as enhancing existing benchmarking suites so that they are made up of samples that more accurately represent use cases of current times. Another aspect that was also missed by almost every other study except one, was the existence of OS noise and how to mitigate against it.

As part of this study, we then proceeded to investigate the effect of OS noise on benchmarking. The results of those tests confirmed that OS noise tends to degrade benchmarking results and as a consequence, one may find that one's results are not as accurate as one may desire or require. Given that, we would go as far as to suggest that OS noise mitigations must always be deployed whenever one needs to benchmark any computer software.

Objective 3

Develop a prototype web browser EE that is capable of hosting and executing CIAs, with native desktop performance.

For our third objective, we fully immersed ourselves in current technological trends and possibilities to devise a truly innovative solution to a challenging problem. Our investigations revealed that the Linux OS has several features that one can easily combine into a high-performance and secure web browser EE.

Using these Linux OS features together with other ideas that relate to container and VM technologies, we were able to create a container-like structure that we refer to as an enclave, because of its process isolation characteristics. This enclave would allow one to execute a native application without compromising on security, as the application executing within it would be completely isolated from the rest of the computer system.

By utilising this enclave we were able to present a possible solution to the problem of being able to execute CIAs within a web browser. Obviously, the proposed solution is merely a very rudimentary prototype however, it provides a sound basis for a fundamental shift in thinking, as to how we go about executing CIAs within a web browser.

Objective 4

Evaluate and benchmark the performance of the prototype web browser EE against those discovered through the SLR, thereby testing the empirical hypothesis of this study.

For objective four, the key findings of our research indicate that our proposed SYS23 solution would provide a significant improvement in performance as compared to existing web browser EEs such as WASM. To this end, we can infer from the results that the proposed SYS23 solution would on average be 42% faster in performance than its WASM counterpart. Together with that, the proposed SYS23 solution should also perform to within 0.45% of the performance of its native equivalent.

When looking at the results of the SYS23 prototype versus those of WASM in more detail, one can quickly form the view that the performance improvements presented by SYS23 are fairly consistent, even though the degree of said performance improvement may vary, in part due to the underlying nature of the algorithm being benchmarked. Suffice it to say though given the varying nature of the benchmarking algorithms one could form a generalised view supporting the claim that in general, the SYS23 prototype should always outperform WASM. The extent to which the SYS23 prototype would outperform its WASM equivalent could be as low as 6% and as high as 82%.

One anomaly that did present itself was with the *gramschmidt* benchmark, whereby WASM outperformed the SYS23 prototype. This is not surprising, given the fact that WASM outperformed the native equivalent benchmark as well. If one had to infer a reason for this anomaly, one could only assume that the WASM application code had undergone some sort of optimisation that is not present within either the native equivalent or the SYS23 prototype. One could further speculate that the optimisation in question relates to how the applications manage their memory allocation and management, given that the *gramschmidt* benchmark makes use of large blocks of memory for the decomposition calculation.

That said, instead of an optimisation being in play, which could be causing the anomaly, it could also be the remaining OS noise which is playing a part in that specific benchmark. Further and deeper analysis is required to really shed some light on this anomaly. When looking at how the SYS23 prototype performs against its native equivalent, we can see similar performance anomalies with *3mm*, *adi*, *jacobi-1d* and *syrk*, whereby the SYS23 prototype outperformed its native equivalent. Given that the application code is 100% the same, one can only infer again that some OS noise is to blame for this, as the assumption would be that the SYS23 prototype would always be at least a fraction of a second slower than its native counterpart given the extra CPU cycles required with which to manage the enclave.

In comparison to the literature reviewed as part of the SLR, we can now suggest based on the findings that the path to even greater performance improvements within a web browser, especially where it relates to CIAs, is by pursuing solutions that reuse what already exists today and by utilising native applications as compared to creating ever-increasing numbers of new interpreted languages with which to solve the problem. It stands to reason that an interpreted language could never outperform its native equivalent, simply because it will always be handicapped by the overhead required for its JIT compilation and optimisation features.

Other studies and research in this area seem to have for the most part focused on optimising the existing EEs and their interpreted languages, being JS and WASM, instead of going back to basics and trying to find a way in which to reuse what already exists, namely executing native applications in a secure manner. By utilising what already exists natively, we would also automatically gain all of the decades worth of optimisations, refinements and expertise that have gone into these native technologies, such as Assembler, C and C++, to name but a few.

Also by utilising these native technologies, not only do we gain on the technology side, but we also gain on the human capital side, whereby we get to reuse those existing skills that many programmers, developers and software engineers already possess, instead of having a whole new cycle by which we have a cohort of people learning the latest and trendiest language that is trying to solve the problem. There is something to be said about having a foundational understanding of software engineering and then building on that foundation, instead of being a *"Jack of all trades, master of none"*.

The obvious counter-arguments to this proposed SYS23 prototype is that it is not easily ported to all existing OSs, which we touch on briefly in the following section. Another counter-argument is that it would be challenging to decide on what is a perfectly and universally usable configuration for the various components like *Cgroups*, *namespaces* and *Seccomp*. However just as both JS and WASM have evolved over time to allow for more and more features, one could start from a similar baseline with SYS23, whereby it enforces the most severe restrictions and over time the industry can allow for less restrictions once they have been proven to favour usability with no or minimal sacrifice to security.

From a practical point of view, one could implement a SYS23 like solution in specific use cases where the author/provider of the native application and the consumers are part of the same trusted group or organisation. In such a use case, one could substantially lessen or even remove the execution restrictions in favour of usability over security, since the applications would be implicitly trusted. From a theoretical point of view, the SYS23 prototype serves as a foundation on which to build upon, from additional research to help refine the proposed solution to advocating to technology providers like Apple to build the required safeguards into future versions of their OSs, thereby making it easier to port the proposed solution to their OSs.

5.5. Proofs

In this section, we seek to formalise the concepts of Linux Seccomp, Cgroups, and namespaces in mathematical form using set theory, characteristic functions, and proposition logic, we then propose lemmas for each one, and then prove them using a theoretical framework.

5.5.1. Secure computing mode

Linux Seccomp operates like a security sandbox, allowing processes to declare a limited set of system calls they can invoke, thereby reducing the attack surface of malicious processes.

Lemma 1.1 (Process Isolation Through System Call Access Restriction Using Set Theory and Characteristic Functions)**.** *Given a set of system calls $S = \{s_1, s_2, \ldots, s_n\}$, and a policy O which allows a subset $S' \subseteq S$ of system calls, any process P that adheres to the policy O can only invoke elements from S'. If P attempts to invoke any system call $s \in S \setminus S'$, the invocation will be denied or result in termination.*

Proof of Process Isolation Using Secure Computing. We can formalise the proof related to the concept of restricting system call access based on predefined rules by proving that under the Seccomp policy, the process P can only invoke system calls from the allowed subset S' and no other.

Let $S = \{s_1, s_2, \ldots, s_n\}$ represent the set of all possible system calls available in a Linux environment. Let O be the Seccomp policy that defines a subset $S' \subseteq S$ as the allowed system calls for a process P. Specifically, policy O blocks any system call $s \in S \setminus S'$.

Assume process P is executing in a Linux environment with Seccomp enabled and policy O applied. Process P can only invoke system calls that belong to the set S', as defined by policy O. If process P attempts to invoke a system call $s \in S \setminus S'$, the Seccomp filter will intercept this call and block it, either by returning an error or terminating the process.

The Seccomp system functions as a filter that enforces policy O. This can be modelled as a characteristic function $f_O(s)$ defined as:

$$f_O(s) = \begin{cases} 1 & \text{if } s \in S' \\ 0 & \text{if } s \in S \setminus S' \end{cases} \tag{5.1}$$

The function $f_O(s)$ evaluates to 1 if the system call s is allowed or $s \in S'$, and 0 if the system call is blocked or $s \in S \setminus S'$.

Suppose process P attempts to invoke a system call $s_d \in S \setminus S'$. By definition of Seccomp, the policy O blocks the invocation of any system call not in S', so:

$$f_O(s_d) = 0 \tag{5.2}$$

This implies that the invocation of s_d will be denied, either by terminating the process or by returning an error. Therefore, for any process P running under the Seccomp policy O, it is impossible to successfully invoke a system call $s \in S \setminus S'$. The process can only invoke system calls in the allowed subset S', which proves that Seccomp restricts the system call interface as intended. ■

In conclusion, if a process P is constrained by Seccomp with policy O, the process can only invoke system calls from the allowed subset $S' \subseteq S$, and any attempt to invoke system calls outside this set will be blocked or return an error. This behaviour enforces the security sandbox limitations of Seccomp.

Lemma 1.2 (Process Isolation Through System Call Access Restriction Using Proposition Logic)**.** *If a Seccomp filter F is applied to a process P to restrict certain system calls $S(s)$, then any unauthorised system call attempted by the process will result in termination or an error.*

Proof of Process Isolation Using Secure Computing. We can formalise the proof related to the concept of restricting system call access based on predefined rules by proving that under the Seccomp policy, the process P can only invoke system call s from the allowed system calls $S(s)$. The call will be successful if the system call s is within the allowed Seccomp filter set $F(s)$ otherwise the process will be blocked $B(s)$.

Given that, we can assume the following two premises, firstly if a system call s is in the filter set $F(s)$, it will not be blocked:

$$\forall s(F(s) \rightarrow \neg B(s)) \tag{5.3}$$

Secondly if a system call s is not in the filter set $F(s)$, it will be blocked:

$$\forall s(\neg F(s) \rightarrow B(s)) \tag{5.4}$$

Therefore, when using the two premises and given that when a process P attempts a system call $S(s)$ and the system call is not in the filter set $\neg F(s)$, it will be blocked $B(s)$:

$$S(s) \wedge \neg F(s) \rightarrow B(s) \tag{5.5}$$

This implies then that if $S(s) \wedge \neg F(s)$, then $B(s)$, as required, either by terminating the process or returning an error. ■

In conclusion, if any system call $S(s)$ attempted by a process P that is not explicitly permitted by the Seccomp filter $F(s)$ will be blocked $B(s)$. This aligns with the function of Seccomp to enforce security by restricting processes to a predefined set of system calls.

5.5.2. Control groups

Lemma 2.1 (Process Resource Consumption Limit Enforcement Using Set Theory)**.** *In a Linux system, Cgroups ensure that the total resource usage of processes within a Cgroup does not exceed the specified limits, thereby maintaining system stability and performance.*

Proof of Process Resource Consumption Limit Enforcement Using Control Groups. Let R be the total available resource, such as CPU time and memory to name a few. Let C be a Cgroup with a resource limit L such that $L \leq R$. Let $P_1, P_2, \ldots, P_n$ be the processes within Cgroup C. Let r_i be the resource usage of process P_i.

By definition, the Cgroup C enforces that the sum of the resources used by all processes within the Cgroup does not exceed the limit L. This can be expressed as:

$$\sum_{i=1}^{n} r_i \leq L \tag{5.6}$$

The Linux kernel monitors the resource usage of each process P_i within the Cgroup C. If the sum of the resource usage $\sum_{i=1}^{n} r_i$ approaches L, the kernel intervenes to ensure the limit is not exceeded. This can involve throttling processes, denying additional resource allocation, or terminating processes.

Cgroups are organised hierarchically, meaning that resource limits set at a parent Cgroup level are propagated to child Cgroups. If C is a parent Cgroup with child Cgroups $C_1, C_2, \ldots, C_m$, each with their own limits $L_1, L_2, \ldots, L_m$ such that $\sum_{j=1}^{m} L_j \leq L$, the resource usage within each child Cgroup must also adhere to these limits:

$$\begin{gathered} \sum_{i \in C_j} r_i \leq L_j \quad \text{for all } j \\ \therefore \forall j (\sum_{i \in C_j} r_i \leq L_j) \end{gathered} \tag{5.7}$$

The isolation provided by Cgroups ensures that resource-intensive processes in one Cgroup do not affect the performance of processes in other Cgroups, which is crucial for maintaining system stability. ■

In conclusion, by enforcing resource limits and leveraging the hierarchical structure of Cgroups, Linux systems ensure that the total resource usage within a Cgroup does not exceed the specified limits. This guarantees that processes within the Cgroup are contained, thereby maintaining system stability and performance.

Lemma 2.2 (Process Resource Consumption Limit Enforcement Using Proposition Logic)**.** *In a Linux system, if a process belongs to a Cgroup and the Cgroup has a specific resource limit, then the process is subject to that resource limit.*

Proof of Process Resource Consumption Limit Enforcement Using Control Groups. Let $P(x, g)$ denote that a process x belongs to a Cgroup g. Let $R(g, r)$ denote that Cgroup g has a resource limit of r. Let $S(x, r)$ denote that a process x is subject to a resource limit of r.

Assuming then that when process x belongs to Cgroup g, and Cgroup g has a resource limit of r, it can be expressed as:

$$P(x, g) \wedge R(g, r) \tag{5.8}$$

Given that, since x belongs to $g(P(x, g))$ and g has a resource limit of $r(R(g, r))$, process x must be subject to r. Using this reasoning, the implication is that $S(x, r)$ and thus:

$$\begin{gathered} P(x, g) \wedge R(g, r) \rightarrow S(x, r) \quad \text{for all } x, g, r \\ \therefore \forall x \forall g \forall r (P(x, g) \wedge R(g, r) \rightarrow S(x, r)) \end{gathered} \tag{5.9}$$

The simplified model of Cgroups as mechanisms with which one can organise and manage system resources assist in ensuring that processes do not exceed their allowed resource limits. ■

In conclusion, by applying resource limits and utilising the hierarchical structure of Cgroups, Linux systems ensure that the total resource consumption within a Cgroup remains within defined boundaries. This containment ensures that processes within the Cgroup are managed effectively, contributing to overall system stability and performance.

5.5.3. Namespaces

Lemma 3.1 (Process Isolation Through Namespaces Using Set Theory)**.** *In an OS with multiple Linux namespaces, processes that are assigned to different namespaces do not share OS resources unless explicitly permitted through mechanisms such as shared namespaces or resource linking.*

Proof of Process Isolation Using Control Group Namespaces. Each Cgroup namespace provides a unique view of a Cgroup linked to the contained processes. Let N_1 and N_2 be two Cgroup namespaces with processes P_1 and P_2. The Cgroup linked to P_1 are isolated from those linked to P_2, unless there is an explicitly shared Cgroup.

Therefore Cgroups are distinct across namespaces unless explicitly shared:

$$\begin{gathered}\text{For all Cgroups } G,\\ G_{N_1} \cap G_{N_2} = \emptyset \\ \text{unless explicitly shared}\end{gathered} \tag{5.10}$$

■

Proof of Process Isolation Using Inter-Process Communication Namespaces. We can think of Linux namespaces as creating disjoint sets of resources for processes in the OS. Each namespace separates certain aspects of the OS resources, and processes in different namespaces cannot access resources outside of their namespace unless explicitly configured to do so. We can formalise the proof by breaking down the different namespaces and showing that the resources they isolate form disjoint sets.

IPC resources, such as message queues, semaphores, and shared memory, are isolated between namespaces. Processes in one namespace N_1 cannot interact with IPC mechanisms in another namespace N_2, unless explicitly shared.

IPC resources are therefore distinct across namespaces:

$$\begin{gathered}\text{For all IPC resources } R,\\ R_{N_1} \cap R_{N_2} = \emptyset \\ \text{unless explicitly shared}\end{gathered} \tag{5.11}$$

■

Proof of Process Isolation Using Mount Namespaces. Let P_1 and P_2 be two processes in different mount namespaces N_1 and N_2, respectively. The file system views of P_1 and P_2 are different, as N_1 and N_2 maintain independent mount points. Thus, P_1 cannot access files or directories mounted in N_2, and P_2 cannot access those in N_1 unless there is an explicit shared mount.

Therefore no process in one mount namespace can access the file hierarchy of another unless shared:

$$\begin{gathered}\text{For all filesystems } F,\\ F_{N_1} \cap F_{N_2} = \emptyset\\ \text{unless explicitly linked}\end{gathered} \tag{5.12}$$

■

Proof of Process Isolation Using Network Namespaces. Processes in different network namespaces have independent network interfaces, routing tables and so forth. Let N_1 and N_2 be two network namespaces with processes P_1 and P_2. The networking resources used by P_1 are isolated from those used by P_2, unless explicit bridges or virtual Ethernet pairs are set up.

Therefore networking resources are isolated across namespaces unless explicitly connected:

$$\begin{gathered}\text{For all network interfaces } I,\\ I_{N_1} \cap I_{N_2} = \emptyset\\ \text{unless a bridge or virtual Ethernet pairs exists}\end{gathered} \tag{5.13}$$

■

Proof of Process Isolation Using Process Namespaces. Processes in different PID namespace hierarchies are only aware of processes in the same or descendant PID namespaces. Let P_1 and P_2 be two processes that belong to PID namespaces N_1 and N_2, respectively. Each PID namespace has its own independent set of PIDs, and processes in N_1 cannot see or interact with the PIDs in N_2, unless parent-child namespace relationships exist.

Therefore processes in one namespace can only view processes in descendant namespaces:

$$\begin{gathered}\text{For all processes } P,\\ P_{N_1} \cap P_{N_2} = \emptyset\\ \text{unless one namespace is a descendant of another}\end{gathered} \tag{5.14}$$

■

Proof of Process Isolation Using Time Namespaces. Time namespaces allow one to have different system monotonic and boot-time clocks per namespace. Let N_1 and N_2 be two time namespaces with processes P_1 and P_2. The system monotonic and boot-time clocks used by P_1 are isolated from those used by P_2, unless explicitly duplicated.

Therefore system monotonic and boot-time clocks isolated unless explicitly duplicated:

$$\text{For all system monotonic and boot-time clocks } C, \\ C_{N_1} \cap C_{N_2} = \emptyset \\ \text{unless values are explicitly duplicated} \tag{5.15}$$

■

Proof of Process Isolation Using User Namespaces. User namespaces allow different processes to have different views of GIDs and UIDs. In namespace N_1, process P_1 can have UID u_1, while in namespace N_2, process P_2 could have the same UID u_1, but their permissions are independent due to namespace isolation. Thus, UIDs are mapped separately in different user namespaces.

Therefore UIDs are isolated unless explicitly mapped:

$$\text{For all UIDs } U, \\ U_{N_1} \cap U_{N_2} = \emptyset \\ \text{unless explicit user mapping is performed} \tag{5.16}$$

■

Proof of Process Isolation Using Unix Time-Sharing Namespaces. UTS namespaces allow one to have different hostnames and NIS domain names per namespace. Let N_1 and N_2 be two UTS namespaces with processes P_1 and P_2. The UTS system identifiers used by P_1 are isolated from those used by P_2, unless explicitly duplicated.

Therefore, hostnames and NIS domain names are isolated unless explicitly duplicated:

$$\text{For all UTSs system identifiers } I, \\ I_{N_1} \cap I_{N_2} = \emptyset \\ \text{unless values are explicitly duplicated} \tag{5.17}$$

■

In conclusion, each namespace provides a level of isolation, meaning that processes in different namespaces cannot share or interfere with each other's resources unless explicitly allowed.

Lemma 3.2. *Process Isolation Through Namespaces Using Proposition Logic] In a Linux system that comprises of multiple namespaces, processes in separate namespaces do not share OS resources unless explicitly allowed through mechanisms like resource linking or shared namespaces.*

Proof of Process Isolation Using Namspaces. Let $P(x, n)$ and $P(y, n)$ denote that a process x and y belong to a namespace n. Let $R(n, t)$ denote that namespace n governs resources of type t. Let $V(x,t)$ and $V(y,t)$ denote that a process x and y has a consistent view of resources of type t.

Assuming then that when process x belongs to namespace n, process y belongs to namespace n, and namespace n governs resources of type t, it can be expressed as:

$$P(x,n) \wedge P(y,n) \wedge R(n,t) \tag{5.18}$$

Given that, since both x and y belong to $n(P(x,n))$ and $n(P(y,n))$, and n governs resources of type $t(R(n,t))$, process x and y must share a consistent view of the resources of type t. Using this reasoning, the implication is that $V(x,t) \leftrightarrow V(y,t)$ and thus:

$$\begin{aligned} P(x,n) \wedge P(y,n) \wedge R(n,t) \rightarrow (V(x,t) \leftrightarrow V(y,t)) \quad \text{for all } x,y,n,t \\ \therefore \forall x \forall y \forall n \forall t (P(x,n) \wedge P(y,n) \wedge R(n,t) \rightarrow (V(x,t) \leftrightarrow V(y,t))) \end{aligned} \tag{5.19}$$

■

In conclusion, processes within the same namespace have a consistent and isolated perspective on resources of the type governed by that namespace, ensuring both coherence and isolation between namespaces.

5.6. Limitations

This study sought to answer a singular research question which has now been thoroughly analysed and discussed in the preceding sections. During the analysis and discussion, certain limitations have been brought to light. First and foremost, our hypothesis has been tested and the findings do indeed support it. That said, it stands to reason that the benchmarking that was used, is but a small subset of all possible scenarios that may exist. To that end, the viewpoint that we have formed needs to be contextualised to the use cases or benchmarking algorithms presented, even though one may be tempted to form a generalised opinion.

Then, one also has to be sensitive to the fact that we based our SYS23 prototype on the Linux OS and its kernel functions, as well as using it to test our hypothesis on. So even though similar mechanisms do indeed exist in other OSs, such as the Windows-based OSs from Microsoft, one would struggle to find similar functions on the OS X based OSs from Apple. Within the Windows-based OSs similar functions for *Cgroups* are *job objects*, *namespaces* are similar to *object manager* and *silos*, and *Seccomp* is similar to *host compute service*.

The deficiencies of the OS X OS also hold true for its mobile iOS OSs, while the mobile equivalents of the Linux-based OS, such as the Android ones from Google and Samsung contain the same functions as found in their desktop counterparts. Those OSs cover the mainstream world, but other non-mainstream OSs will need to be further evaluated to see if they contain mechanisms that could be used to create a secure enclave, as proposed by the SYS23 prototype. Some such OSs that come to mind are those based on Solaris from Oracle (previously Sun Microsystems) and BSD, which find favour in the scientific community, where the ability to execute compute-intensive applications through a web browser is sought after.

An interesting proposition would be to propose a design whereby if the required functions are not available within the given OS, one could provide such mechanisms as an add-on to the OS, which would allow one to port that design to OS X, iOS, and any other OS that may not provide such functions out of the box. All that one would need to ensure is that the design adheres to the three primary aspects of the SYS23 design, namely being able to execute native applications, being able to be precompiled with an AOT compiler, and most importantly, provide for the creation of a secure enclave within which the native application can execute safely.

A final aspect to also keep in mind is that the proposed SYS23 prototype only tries to solve the problem that exists on the client side of the end-to-end web ecosystem. It does not propose how one would go about integrating the server-side or the distribution of the native applications that would be downloaded. Those functions would equally need to be uplifted to accommodate the requirements of the SYS23 platform as a whole and any shortcomings would need to be dealt with in order to have a fully functional end-to-end web server to web browser system together with an efficient user experience.

Given these limitations, we are presented with areas that require further research. We delve into the specifics of those areas of future research, which relate directly to this study in the next chapter.

5.7. Synopsis

In this chapter, we analysed the performance of a prototype implementation of our SYS23 system design against both that of native ISA and OS compatible applications as well as WASM-based applications. Initially, we compared WASM against their native counterparts and obviously we can observe that the native benchmarks outperformed the WASM ones. This can simply be explained given the overhead that WASM performance suffers from as an interpreted language.

Thereafter, we compared WASM against the SYS23-based applications. Here we also observed that the SYS23-based applications had superior performance, again this can simply be explained given that the SYS23-based applications require no interpretation at time of execution and hence their performance would be quite close to that of native ISA and OS compatible applications. Even by executing within the SYS23 enclave, the SYS23-based applications still outperform their WASM-based counterparts.

In the next and concluding chapter, we will briefly summarise this study and show how all of its objectives have been dealt with, together with further areas of research that can be undertaken in this regard.

CHAPTER 6

CONCLUSION

In this chapter, we ponder the conclusions that have been derived at as well as to look at avenues that other researchers may wish to look into, as subsequent studies to the study undertaken herein.

This chapter summarises the findings and looks to the future through the following:

6.1. Recap

In this study, we set out to investigate the shortcomings within existing web browser EEs, specifically where they relate to being able to host and execute CIAs with native desktop performance. We examined the current state of web browser EEs and formed an initial hypothesis which was that:

Hypothesis

A web browser EE that can host and execute CIAs on any device with native desktop performance can be created.

This in turn led us to formulate a singular research question that this study endeavoured to answer, which was:

Research Question

What software architecture and design does a new web browser EE need to comprise of in order to be able to host and execute CIAs with native desktop performance?

Subsequent to this we presented four objectives that underpinned this study, which would allow us to answer the singular research question. These four objectives also correlated to the work presented in chapters 2 through 5. In addition, we also released those four chapters to the public domain through various conferences and journal publications, with the aim of stress testing their content with the general scientific community.

In the following sections, we will briefly re-examine the four objectives before presenting our final conclusion.

6.2. Objectives

Objective 1

Identify the features and limitations of existing web browser EEs that can host and execute applications.

With objective one we identified the current state of web browser EEs, where we delved into both their strengths as well as their weaknesses and found that existing web browser EEs are not currently equipped with the ability to host and execute CIAs with native desktop performance.

Objective 2

Employ a suitable research methodology by determining which ones are best aligned to deliver on the requirements and goal of this study.

With objective two we furnished a methodology that formed the foundation of our study which was deemed sufficient and adequate to align with the outcomes desired from this study. Together with that we also delivered a contribution relating to OS noise mitigations, which assisted in improving the benchmarking undertaken with objective four.

Objective 3

Develop a prototype web browser EE that is capable of hosting and executing CIAs, with native desktop performance.

With objective three we presented a system design for a web browser EE that we propose will be able to host and execute CIAs with native desktop performance. That design shows how one would be able to use existing technologies based on the Linux OS to build an isolated EE within which one would be able to execute untrusted but natively compatible applications that were downloaded from a web server.

Objective 4

Evaluate and benchmark the performance of the prototype web browser EE against those discovered through the SLR, thereby testing the empirical hypothesis of this study.

With objective four we benchmarked our presented system design and evaluated it against the most recently developed web browser EE, that being the WASM-based EE. We also showed how our system design compared to that of the WASM EE, from where we drew certain favourable conclusions.

6.3. Contributions

From a contribution point of view, which is best viewed through three lenses, being methodological, theoretical and practical, we put forward the following.

The methodology contributions presented in this study mark a significant advancement in the area of benchmarking, where existing approaches have fallen short in mitigating against OS noise. This critical gap was addressed by this study through the proposed mitigations, which aid in ensuring that the desired benchmarking is more accurate. The novelty of these mitigations is that they are not complex to implement and are supported by numerous OSs. Previous studies have also mostly ignored these OS noise and the effect that they may ultimately have on the accuracy of the benchmarking results.

A rigorous process was employed with which to implement this methodology and it also underwent a robust and extensive validation process by utilising PolyBench/C to test the performance implications and effectiveness of the OS noise mitigations. Practical applications of this methodology reach far beyond the scope of this study and are suited to all sorts of computer software performance benchmarking. One limitation that needs to be noted is that not all OS noise were discoverable and thus not all OS noise could be mitigated against. Future research would aid in increasing the list of know OS noise mitigations.

This study significantly contributes to the theoretical understanding of application isolation or application enclaves as referred to in this study. This study proposes to bridge existing gaps related to the far-reaching use cases of application enclaves and that they are not just limited to the execution of trusted applications, as per conventional understanding. Theoretical innovations presented in this study provide a fresh perspective as to how one might use application enclaves for the specific use case of this study, thereby expanding the theoretic boundaries of possible use cases of application enclaves.

The development of these theoretical contributions was based on a strong methodology that involved extensive literature reviews, iterative design and model testing. This process facilitated an in-depth exploration of existing theories while uncovering their limitations, specifically where they relate to the execution of CIAs. The implications of these theoretical contributions are extensive. They not only push forward academic discussions on the execution of CIAs but also have potential applications across various types of computer software. Future theoretical research could build on these foundations and propose further enhancements to aspects such as security, AOT compilation and native execution of applications as they relate to application enclaves.

Substantial practical contributions are made by this study, specifically where it focused on answering the singular research question by proposing the SYS23 prototype. This prototype provides a possible solution to the problem of being able to execute CIAs within web browsers with performance that is equivalent to that of native desktop performance. These contributions were rigorously tested and benchmarked against both existing solutions and their native equivalent. The results of the implementation of the SYS23 prototype suggest that it would indeed far surpass the performance of existing solutions.

The implications of these practical contributions suggest that its use extends well beyond the focus area of this study. The long-term advantages of this prototype could be significant, especially in terms of cost savings and learning curve, whereby one would be utilising existing knowledge and there would be no need to completely rewrite applications, simply to allow them to execute within a web browser. Although the practical implementation developed in this study mark a significant advancement, there are still opportunities for further development. While the approach has proven effective with regard to EEs, it would be valuable to assess its use across a broader range of benchmarks and use cases.

6.4. Conclusion

In the previous section, we summarised how the objectives of this study were dealt with, which allowed us to examine the subject area of web browser EEs that can host and execute CIAs with native desktop performance. We examined the current state of existing web browser EEs and showed how they are currently not up to the task of hosting and executing CIAs with native desktop performance.

That conclusion was derived at after we completed an exhaustive SLR together with also identifying that WASM-based EEs are currently the foremost web browser EE, delivering substantial performance improvements over its JS-based EEs counterpart. After that, we present the methodology underpinning our study, together with improvements for benchmarking computer software performance.

Subsequent to that we then proposed our system design that we believe can be used by web browsers to host and execute CIAs with equivalent native desktop performance. Finally, we then benchmarked and evaluated our system design and showed how our design performed favourably against the foremost existing web browser EE, particularly because it has much less overhead when executing ISA and OS natively compatible applications.

In conclusion, within this context we believe that our study has achieved its goals of answering our singular research question by presenting a well-designed system architecture, which can be expanded upon. We also believe that our system design has sufficient merit to warrant further evaluation. We also acknowledge that our system design is limited and merely a prototype, where in the following section we suggest further directions for research to improve upon our system design.

6.5. Future directions

Previously in section 2.4.4.7. we identified areas of research unrelated to this study that require further investigation. In this section we provide areas of research that relate directly to this study, that also require further investigation. These future studies would complement this study with the aim of formulating a full end-to-end solution to the subject area originally investigated.

The related studies that requires further investigation can be summarised as:

1. **Binary Artefact Creation**: Investigating the compilation of applications so that they are binary compatible across multiple ISAs and OSs. This may require that certain syntactic sugar be added to an application's source code base, so as to allow it to be compiled more efficiently depending on the target ISA and OS. This would also assume that an application has one source code base, much like the Linux kernel, instead of multiple different source code bases.

2. **Binary Artefact Repository**: Investigating the storage and/or archiving of applications in their binary state. This storage facility would allow for retaining multiple lifecycle versions of any one application together with binary compatible versions for multiple ISAs and OSs. This storage facility would also need to be compatible with various web servers, such as the open-source Apache web server and the Microsoft Internet Information Services (IIS) web server.

3. **Binary Artefact Distribution**: Investigating the distribution mechanisms that could be employed to more efficiently distribute the applications in their binary state. This could be facilitated through content delivery networks (CDNs) such as those by Akamai, Cloudflare and Fastly, to name a few. One could also employ other mechanisms such as those provided by torrent and other peer-to-peer like technologies.

   This obviously does not remove the age-old mechanism of simply downloading an application in its binary state and installing it on one's computer system. The only caveat would be that the execution of said application would still need to occur within the SYS23 enclave.

4. **Binary Artefact Fast Downloading**: Investigating the downloading mechanism that could be employed to efficiently download the application in their binary state. This could be achieved by only downloading the sections of the application that have changed, an

example of this would be to use a checksum method to perform a bit-level data transfer. That mechanism is similar to the one that the `rsync` command uses to transfer files, which allows it to dramatically decrease the amount of data that needs to be transferred.

5. **Enclave Adoption**: Investigating the broader adoption of the SYS23 enclave on more ISAs and OSs, such as those based on Apple hardware and its OS X line of OSs, furthermore adoption on mobile architectures such as those based on Apple mobile devices and their iOS line of OSs as well as Google's Android. Adoption on the mobile architecture based on Google's Android should be more straight forward because the Android OS is primarily based on Linux, which is also the foundation on which SYS23 has been prototyped in this study.

6. **Enclave Benchmarking**: Investigation into a more comprehensive suite of benchmarks, which are specifically designed to stress test SYS23 enclaves. One could use the PolyBench/C benchmarking suite as a baseline and expand upon it or one could design a purpose-built suite of benchmarks. Specific areas that would need benchmarking are those related to all aspects of CIAs, but also those that require enormous amounts of processing power such as Artificial Intelligence (AI), 3D rendering, especially in fast-paced gaming, and high-volume data processing, like pattern matching, to name a few.

7. **Enclave Enhancements**: Investigating further enhancements to this study and its SYS23 prototype especially with regard to further hardening its security and reducing its attack surface. Other aspects relating to the SYS23 enclave can also be expanded upon, such as improving the management of downloaded applications, caching them for future reuse and enabling multiple downloaded applications from being able to interact with one another.

These are only a few possible areas of research that one could investigate further, although many more may exist. Over time many more possible areas of research may also present themselves.

BIBLIOGRAPHY

Ahn, W., Choi, J., Shull, T., Garzarán, M.J. & Torrellas, J. 2014. Improving JavaScript Performance by Deconstructing the Type System. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. New York: Association for Computing Machinery: 496–507. doi: 10.1145/2594291.2594332.

Aponte, M. 2020. Create Your Single-Page Application. In *Building Single Page Applications in .NET Core 3 : Jumpstart Coding Using Blazor and C#*, Berkeley: Apress, Berkeley, CA, 33–71. doi: 10.1007/978-1-4842-5747-0_3.

Arteaga, J.C., Donde, S., Gu, J., Floros, O., Satabin, L., Baudry, B. & Monperrus, M. 2020. Superoptimization of WebAssembly bytecode. *2020: Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*: 36–40. doi: 10.1145/3397537.3397567.

Auler, R., Borin, E., de Halleux, P., Moskal, M. & Tillmann, N. 2014. Addressing JavaScript JIT Engines Performance Quirks: A Crowdsourced Adaptive Compiler. In *Compiler Construction*, vol. 8409, Berlin: Springer, 218–237. doi: 10.1007/978-3-642-54807-9_13.

Babbie, E.R. 2016. *The Practice of Social Research*. 14th ed. Boston: Cengage Learning.

Baltes, S. & Ralph, P. 2022. Sampling in Software Engineering Research: A Critical Review and Guidelines. *Empirical Software Engineering*, 27(4): 1–38. doi: 10.1007/s10664-021-10072-8.

Barry, D., Jagode, H., Danalis, A. & Dongarra, J. 2023. Memory Traffic and Complete Application Profiling with PAPI Multi-Component Measurements. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Florida: Institute of Electrical and Electronics Engineers: 393–402. doi: 10.1109/IPDPSW59300.2023.00070.

Bartz-Beielstein, T., Doerr, C., Berg, D.v.d., Bossek, J., Chandrasekaran, S., Eftimov, T., Fischbach, A., Kerschke, P., La Cava, W., López-Ibáñez, M., Malan, K.M., Moore, J.H., Naujoks, B., Orzechowski, P., Volz, V., Wagner, M. & Weise, T. 2020. Benchmarking in Optimization: Best Practice and Open Issues. *arXiv*, abs/2007.03488: 1–50. doi: 10.48550/arXiv.2007.03488.

Belkin, A., Gelernter, N. & Cidon, I. 2019. The Risks of WebGL: Analysis, Evaluation and Detection. In *Computer Security – ESORICS 2019*, Cham: Springer International Publishing, 545–564. doi: 10.1007/978-3-030-29962-0_26.

Berners-Lee, T. 1990. *Information Management: A Proposal*. CERN, Geneva.

Berners-Lee, T., Cailliau, R., Groff, J. & Pollermann, B. 1992. World-Wide Web: The Information Universe. *Electronic Networking*, 2(1): 52–58.

Bhansali, S., Aris, A., Acar, A., Oz, H. & Uluagac, A.S. 2022. A First Look at Code Obfuscation for WebAssembly. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '22. New York: Association for Computing Machinery: 140–145. doi: 10.1145/3507657.3528560.

Bian, W., Meng, W. & Wang, Y. 2019. Poster: Detecting WebAssembly-Based Cryptocurrency Mining. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. New York: Association for Computing Machinery: 2685–2687. doi: 10.1145/3319535.3363287.

Biradar, S.M., Shekhar, R. & Reddy, A.P. 2018. Build Minimal Docker Container Using Golang. In *Proceedings of the Second International Conference on Intelligent Computing and Control Systems*. ICICCS 2018. Madurai: Institute of Electrical and Electronics Engineers: 1–4. doi: 10.1109/ICCONS.2018.8663172.

Borisov, P.D. & Kosolapov, Yu.V. 2020. On the Automatic Analysis of the Practical Resistance of Obfuscating Transformations. *Automatic Control and Computer Sciences*, 54(7): 619–629. doi: 10.3103/S0146411620070044.

Bormann, C. 2018. Well-Known URIs for the WebSocket Protocol. Retrieved from https://rfc-editor.org/rfc/rfc8307 [15 April 2024].

Bourgoin, M. & Chailloux, E. 2015. High-level accelerated array programming in the web browser. *ARRAY 2015: Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*: 31–36. doi: 10.1145/2774959.2774964.

Brito, T., Lopes, P., Santos, N. & Santos, J.F. 2022. Wasmati: An efficient static vulnerability scanner for WebAssembly. *Computers & Security*, 118: 102745. doi: 10.1016/j.cose.2022.102745.

Brown, S. 2022. The C4 Model for Visualising Software Architecture. Retrieved from https://c4model.com [15 April 2024].

Bruyat, J., Champin, P.A., Médini, L. & Laforest, F. 2021. WasmTree: Web Assembly for the Semantic Web. In *The Semantic Web*, vol. 12731, Cham: Springer International Publishing, 582–597. doi: 10.1007/978-3-030-77385-4_35.

Bryman, A. 2016. *Social Research Methods*. 5th ed. Oxford: Oxford University Press.

Bryman, A. & Bell, E. 2011. *Business Research Methods*. 3rd ed. Oxford: Oxford University Press.

Calegari, P., Levrier, M. & Balczyński, P. 2019. Web Portals for High-Performance Computing:

A Survey. *ACM Transactions on the Web*, 13(1): 1–36. doi: 10.1145/3197385.

Cao, B., Shi, M. & Li, C. 2017. The solution of web font-end performance optimization. In *2017 10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*. Shanghai: Institute of Electrical and Electronics Engineers: 1–5. doi: 10.1109/CISP-BMEI.2017.8302083.

Casteleyn, S., Garrigós, I. & Mazón, J.N. 2014. Ten Years of Rich Internet Applications: A Systematic Mapping Study, and Beyond. *ACM Transactions on the Web*, 8(3): 1–46. doi: 10.1145/2626369.

Chandra, S., Gordon, C.S., Jeannin, J.B., Schlesinger, C., Sridharan, M., Tip, F. & Choi, Y. 2016. Type Inference for Static Compilation of JavaScript. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. New York: Association for Computing Machinery: 410–429. doi: 10.1145/2983990.2984017.

Cho, M., Han, Y., Kim, M. & Kim, S.W. 2015. O2WebCL: an automatic OpenCL-to-WebCL translator for high performance web computing. *Journal of Supercomputing*, 71(6): 2050–2065. doi: 10.1007/s11227-014-1260-4.

Choi, J., Shull, T. & Torrellas, J. 2019. Reusable Inline Caching for JavaScript Performance. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. New York: Association for Computing Machinery: 889–901. doi: 10.1145/3314221.3314587.

Choi, M.H. & Moon, I.Y. 2019. Development of Branch Processing System Using WebAssembly and JavaScript. *Journal of information and communication convergence engineering*, 17(4): 234–238. doi: 10.6109/jicce.2019.17.4.234.

Creswell, J.W. & Creswell, J.D. 2018. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. 5th ed. Los Angeles: SAGE Publications.

De Macedo, J., Abreu, R., Pereira, R. & Saraiva, J. 2021. On the Runtime and Energy Performance of WebAssembly: Is WebAssembly superior to JavaScript yet? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. Melbourne: Institute of Electrical and Electronics Engineers: 255–262. doi: 10.1109/ASEW52652.2021.00056.

De Macedo, J., Abreu, R., Pereira, R. & Saraiva, J. 2022. WebAssembly versus JavaScript: Energy and Runtime Performance. In *2022 International Conference on ICT for Sustainability (ICT4S)*. Plovdiv: Institute of Electrical and Electronics Engineers: 24–34. doi: 10.1109/ICT4S55073.2022.00014.

de Oliveira, D.B., Casini, D. & Cucinotta, T. 2023. Operating System Noise in the Linux Kernel. *IEEE Transactions on Computers*, 72(1): 196–207. doi: 10.1109/TC.2022.3187351.

Dingledine, R., Mathewson, N. & Syverson, P. 2004. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*. vol. 4, 303–320.

DiPierro, M. 2018. The Rise of JavaScript. *Computing in Science & Engineering*, 20(1): 9–10. doi: 10.1109/MCSE.2018.011111120.

Dodig-Crnkovic, G. 2002. Scientific Methods in Computer Science. In *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden*. Skövde, 126–130.

Doherty, W.J. & Thadani, A.J. 1982. *The Economic Value of Rapid Response Time (IBM Technical Report GE20-0752-0)*. IBM Corporation, New York.

Dot, G., Martínez, A. & González, A. 2015. Analysis and Optimization of Engines for Dynamically Typed Languages. In *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Florianopolis: Institute of Electrical and Electronics Engineers: 41–48. doi: 10.1109/SBAC-PAD.2015.20.

Douceur, J.R., Elson, J., Howell, J. & Lorch, J.R. 2008. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX OSDI 2008. San Diego: USENIX Association: 339–354. doi: 10.5555/1855741.1855765.

Eich, B. 2015. From ASM.JS to WebAssembly. Retrieved from https://brendaneich.com/2015/06/from-asm-js-to-webassembly/ [15 April 2024].

Fernando, N., Loke, S.W. & Rahayu, W. 2013. Mobile cloud computing: A survey. *Future generation computer systems*, 29(1): 84–106. doi: 10.1016/j.future.2012.05.023.

Fette, I. & Melnikov, A. 2011. The WebSocket Protocol. Retrieved from https://rfc-editor.org/rfc/rfc6455 [15 April 2024].

Fink, G. & Flatow, I. 2014. Introducing Single Page Applications. In *Pro Single Page Application Development: Using Backbone.js and ASP.NET*, Berkeley: Apress, Berkeley, CA, 3–13. doi: 10.1007/978-1-4302-6674-7_1.

Fleming, P.J. & Wallace, J.J. 1986. How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results. *Communications of the ACM*, 29(3): 218–221. doi: 10.1145/5666.5673.

Frankston, B. 2020. The JavaScript Ecosystem. *IEEE Consumer Electronics Magazine*, 9(6): 84–89. doi: 10.1109/MCE.2020.3009457.

Fras, K. & Nowak, Z. 2019. WebAssembly - Hope for Fast Acceleration of Web Applications Using JavaScript. In *Information Systems Architecture and Technology: Proceedings of 40th Anniversary International Conference on Information Systems Architecture and Technology - ISAT 2019*, Cham: Springer, 275–284. doi: 10.1007/978-3-030-30440-9_26.

Fraternali, P., Rossi, G. & Sánchez-Figueroa, F. 2010. Rich Internet Applications. *IEEE Internet Computing*, 14(3): 9–12. doi: 10.1109/MIC.2010.76.

Freed, N. & Borenstein, N. 1996. Multipurpose internet mail extensions (MIME) part one: Format of internet message bodies. Retrieved from https://rfc-editor.org/rfc/rfc2045 [15 April 2024].

Fukuda, H. & Yamamoto, Y. 2008. A System for Supporting Development of Large Scaled Rich Internet Applications. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. ASE '08. L'Aquila: Institute of Electrical and Electronics Engineers: 459–462. doi: 10.1109/ASE.2008.73.

Gaj, P., Skrzewski, M., Stój, J. & Flak, J. 2015. Virtualization as a Way to Distribute PC-Based Functionalities. *IEEE Transactions on Industrial Informatics*, 11(3): 763–770. doi: 10.1109/TII.2014.2360499.

Garrett, J.J. 2007. Ajax: A new approach to web applications. Tech. rep., Adaptive Path.

Gong, L., Pradel, M. & Sen, K. 2015. JITProf: Pinpointing JIT-Unfriendly JavaScript Code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo: Association for Computing Machinery: 357–368. doi: 10.1145/2786805.2786831.

Gonzalez, N.M., Morari, A. & Checconi, F. 2017. Jitter-Trace: a low-overhead OS noise tracing tool based on Linux Perf. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*. ROSS '17. Washington: Association for Computing Machinery: 1–8. doi: 10.1145/3095770.3095772.

Gregor, S. 2021. Reflections on the Practice of Design Science in Information Systems. In *Engineering the Transformation of the Enterprise: A Design Science Research Perspective*, Cham: Springer, 101–113. doi: 10.1007/978-3-030-84655-8_7.

Grimmer, M., Schatz, R., Seaton, C., Würthinger, T., Luján, M. & Mössenböck, H. 2018. Cross-Language Interoperability in a Multi-Language Runtime. *ACM Transactions on Programming Languages and Systems*, 40(2): 1–43. doi: 10.1145/3201898.

Gusenbauer, M. 2019. Google Scholar to overshadow them all? Comparing the sizes of 12 academic search engines and bibliographic databases. *Scientometrics*, 118(1): 177–214. doi: 10.1007/s11192-018-2958-5.

Gusenbauer, M. & Haddaway, N.R. 2020. Which academic search systems are suitable for systematic reviews or meta-analyses? Evaluating retrieval qualities of Google Scholar, PubMed, and 26 other resources. *Research Synthesis Methods*, 11(2): 181–217. doi: 10.1002/jrsm.1378.

Hardie, T. 2016. Clarifying Registry Procedures for the WebSocket Subprotocol Name Registry. Retrieved from https://rfc-editor.org/rfc/rfc7936 [15 April 2024].

Harper, R. 2013. *Practical Foundations for Programming Languages*. New York: Cambridge University Press.

Hassani, H. 2017. Research Methods in Computer Science: The Challenges and Issues. *arXiv*, abs/1703.04080: 1–16.

Haßler, K. & Maier, D. 2021. WAFL: Binary-Only WebAssembly Fuzzing with Fast Snapshots. In *ROOTS '21: Reversing and Offensive-oriented Trends Symposium*, New York: Association for Computing Machinery, 23–30. doi: 10.1145/3503921.3503924.

Hatchuel, A. & Weil, B. 2003. A New Approach of Innovative Design: An Introduction to C-K Theory. In *DS 31: Proceedings of ICED 03, the 14th International Conference on Engineering Design*. ICED. Stockholm: Design Society: 1–15.

Hatchuel, A., Weil, B. & Le Masson, P. 2013. Towards an Ontology of Design: Lessons from C–K Design Theory and Forcing. *Research in Engineering Design*, 24(2): 147–163. doi: 10.1007/s00163-012-0144-y.

Heo, J., Woo, S., Jang, H., Yang, K. & Lee, J.W. 2016. Improving JavaScript performance via efficient in-memory bytecode caching. In *2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*. Seoul: Institute of Electrical and Electronics Engineers: 1–4. doi: 10.1109/ICCE-Asia.2016.7804810.

Herman, D., Wagner, L. & Zakai, A. 2014. The asm.js Specification. Retrieved from http://asmjs.org/spec/latest/ [15 April 2024].

Herrera, D., Chen, H., Lavoie, E. & Hendren, L. 2018. Numerical computing on the web: benchmarking for the future. *DLS 2018: Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, 53(8): 88–100. doi: 10.1145/3393673.3276968.

Hevner, A.R. 2021. Pedagogy for Doctoral Seminars in Design Science Research. In *Engineering the Transformation of the Enterprise: A Design Science Research Perspective*, Cham: Springer, 185–198. doi: 10.1007/978-3-030-84655-8_12.

Hevner, A.R. & Chatterjee, S. 2010. Design Science Research in Information Systems. *Design Research in Information Systems: Theory and Practice*, 22: 9–22. doi: 10.1007/978-1-4419-5653-8_2.

Hevner, A.R., March, S.T., Park, J. & Ram, S. 2004. Design Science in Information Systems Research. *MIS Quarterly*, 28(1): 75–105. doi: 10.2307/25148625.

Hilbig, A., Lehmann, D. & Pradel, M. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021*. WWW '21. New York: Association for Computing Machinery: 2696–2708. doi: 10.1145/3442381.3450138.

Ho, X., de Joya, J.M. & Trevett, N. 2017. State-of-the-Art WebGL 2.0. *SA 2017: SIGGRAPH*

Asia 2017 Courses: 1–51. doi: 10.1145/3134472.3134479.

Hockley, D. & Williamson, C. 2022. Benchmarking Runtime Scripting Performance in Wasmer. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. ICPE '22. New York: Association for Computing Machinery: 97–104. doi: 10.1145/3491204.3527477.

Hopcroft, J.E., Motwani, R. & Ullman, J.D. 2006. *Introduction to Automata Theory, Languages, And Computation*. 3rd ed. London: Addison-Wesley.

Huber, S., Demetz, L. & Felderer, M. 2022. A comparative study on the energy consumption of Progressive Web Apps. *Information Systems*, 108(C): 1–13. doi: 10.1016/j.is.2022.102017.

Iivari, J. & Venable, J.R. 2009. Action research and design science research -Seemingly similar but decisively dissimilar. In *European Conference on Information Systems (ECIS) 2009 Proceedings*. Verona: AIS Electronic Library (AISeL): 1–13.

International Organization for Standardization. 2017. The JSON Data Interchange Syntax - ISO/IEC 21778:2017. Retrieved from https://www.iso.org/standard/71616.html [15 April 2024].

Jangda, A., Powers, B., Berger, E.D. & Guha, A. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC 2019. Renton: USENIX Association: 107–120.

Jansen, J.M. & van Groningen, J. 2016. A Portable VM-based implementation Platform for non-strict Functional Programming Languages. In *IFL 2016: Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages*, New York: Association for Computing Machinery, 1–14. doi: 10.1145/3064899.3064903.

Jiang, C. & Jin, X. 2017. Quick Way to Port Existing C/C++ Chemoinformatics Toolkits to the Web Using Emscripten. *Journal of Chemical Information and Modeling*, 57(10): 2407–2412. doi: 10.1021/acs.jcim.7b00434.

Kataoka, T., Ugawa, T. & Iwasaki, H. 2018. A framework for constructing javascript virtual machines with customized datatype representations. *SAC 2018: Proceedings of the 33rd Annual ACM Symposium on Applied Computing*: 1238–1247. doi: 10.1145/3167132.3167266.

Kharb, L., Chahal, D. & Vagisha. 2021. Smart Mobility: Understanding Handheld Device Adoption. In *Advances in Communication and Computational Technology*, Singapore: Springer, 13–29. doi: 10.1007/978-981-15-5341-7_2.

Kharraz, A., Ma, Z., Murley, P., Lever, C., Mason, J., Miller, A., Borisov, N., Antonakakis, M. & Bailey, M. 2019. Outguard: Detecting In-Browser Covert Cryptocurrency Mining in the Wild. In *The World Wide Web Conference*. WWW '19. New York: Association for Computing Machinery: 840–852. doi: 10.1145/3308558.3313665.

Kienle, H.M. & Distante, D. 2014. Evolution of Web Systems. In *Evolving Software Systems*, Berlin: Springer, 201–228. doi: 10.1007/978-3-642-45398-4_7.

Kivunja, C. 2018. Distinguishing between Theory, Theoretical Framework, and Conceptual Framework: A Systematic Review of Lessons from the Field. *International Journal of Higher Education*, 7(6): 44–53. doi: 10.5430/ijhe.v7n6p44.

Kluge, J., Kargl, F. & Weber, M. 2007. The Effects of the Ajax Technology on Web Application Usability. In *Proceedings of the Third International Conference on Web Information Systems and Technologies - Volume 1: WEBIST*. Barcelona: SciTePress: 289–294. doi: 10.5220/0001286102890294.

Kofod-Petersen, A. 2015. How to do a Structured Literature Review in computer science. *ResearchGate*.

Konoth, R.K., Vineti, E., Moonsamy, V., Lindorfer, M., Kruegel, C., Bos, H. & Vigna, G. 2018. MineSweeper: An In-Depth Look into Drive-by Cryptocurrency Mining and Its Defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. New York: Association for Computing Machinery: 1714–1730. doi: 10.1145/3243734.3243858.

Koper, D. & Woda, M. 2022. Performance Analysis and Comparison of Acceleration Methods in JavaScript Environments Based on Simplified Standard Hough Transform Algorithm. In *New Advances in Dependability of Networks and Systems*, Cham: Springer, 131–142. doi: 10.1007/978-3-031-06746-4_13.

Kozlovičs, S. 2020. webAppOS: Creating the Illusion of a Single Computer for Web Application Developers. In *Web Information Systems and Technologies*, vol. 399, Cham: Springer International Publishing, 1–21. doi: 10.1007/978-3-030-61750-9_1.

Kumar, R. 2018. *Research Methodology: A Step-by-Step Guide for Beginners*. 5th ed. London: SAGE Publications.

Lehmann, D., Kinder, J. & Pradel, M. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *Proceedings of the 29th USENIX Security Symposium*. Virtual: USENIX Association: 217–234.

Lehmann, D. & Pradel, M. 2022. Finding the dwarf: recovering precise types from WebAssembly binaries. In *PLDI 2022: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, New York: Association for Computing Machinery, 410–425. doi: 10.1145/3519939.3523449.

Letz, S., Orlarey, Y. & Fober, D. 2018. FAUST Domain Specific Audio DSP Language Compiled to WebAssembly. In *Companion Proceedings of the The Web Conference 2018*. WWW 2018. Lyon: International World Wide Web Conferences Steering Committee: 701–709. doi: 10.1145/3184558.3185970.

Liu, A.C. & You, Y.P. 2022. Offworker: An Offloading Framework for Parallel Web Applications.

In *Web Information Systems Engineering – WISE 2022*, Cham: Springer, 170–185. doi: 10.1007/978-3-031-20891-1_13.

Liu, W., Yang, X., Lin, H., Li, Z. & Qian, F. 2022. Fusing Speed Index during Web Page Loading. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6: 1–23. doi: 10.1145/3511214.

Liu, Y. 2019. JSOptimizer: An Extensible Framework for JavaScript Program Optimization. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. Montreal: Institute of Electrical and Electronics Engineers: 168–170. doi: 10.1109/ICSE-Companion.2019.00069.

Lubbers, P. & Greco, F. 2010. HTML5 Websocket: A Quantum Leap in Scalability for the Web. Retrieved from http://www.websocket.org/quantum.html [15 April 2024].

Lyu, S. 2021. High-Performance Web Frontend Using WebAssembly. In *Practical Rust Web Projects: Building Cloud and Web-Based Applications*, Berkeley: Apress, Berkeley, CA, 193–249. doi: 10.1007/978-1-4842-6589-5_6.

Ma, Y., Xiang, D., Zheng, S., Tian, D. & Liu, X. 2019. Moving Deep Learning into Web Browser: How Far Can We Go? In *WWW '19: The World Wide Web Conference*, New York: Association for Computing Machinery, 1234–1244. doi: 10.1145/3308558.3313639.

Maas, M., Asanović, K. & Kubiatowicz, J. 2017. Full-System Simulation of Java Workloads with RISC-V and the Jikes Research Virtual Machine. In *1st Workshop on Computer Architecture Research with RISC-V*. Association for Computing Machinery: 1–7.

Mäkitalo, N., Bankowski, V., Daubaris, P., Mikkola, R., Beletski, O. & Mikkonen, T. 2021. Bringing WebAssembly up to Speed with Dynamic Linking. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. SAC '21. New York: Association for Computing Machinery: 1727–1735. doi: 10.1145/3412841.3442045.

Malle, B., Giuliani, N., Kieseberg, P. & Holzinger, A. 2018. The Need for Speed of AI Applications: Performance Comparison of Native vs. Browser-based Algorithm Implementations. *arXiv*, abs/1802.03707: 1–21.

Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C. & Huici, F. 2017. My VM is Lighter (and Safer) than your Container. In *SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles*, New York: Association for Computing Machinery, 218–233. doi: 10.1145/3132747.3132763.

March, S.T. & Smith, G.F. 1995. Design and Natural Science Research on Information Technology. *Decision Support Systems*, 15(4): 251–266. doi: 10.1016/0167-9236(94)00041-2.

Marion, C. & Jomier, J. 2012. Real-time collaborative scientific WebGL visualization with WebSocket. *Web3D '12: Proceedings of the 17th International Conference on 3D Web Technology*: 47–50. doi: 10.1145/2338714.2338721.

Matsakis, N.D., Herman, D. & Lomov, D. 2014. Typed Objects in JavaScript. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*. DLS '14. New York: Association for Computing Machinery: 125–134. doi: 10.1145/2661088.2661095.

Mazaheri, M.E., Bayat Sarmadi, S. & Taheri Ardakani, F. 2022. A Study of Timing Side-Channel Attacks and Countermeasures on JavaScript and WebAssembly. *ISC International Journal of Information Security*, 14(1): 27–46. doi: 10.22042/isecure.2021.263565.599.

McAnlis, C., Lubbers, P., Jones, B., Tebbs, D., Manzur, A., Bennett, S., D'Erfurth, F., Garcia, B., Lin, S., Popelyshev, I., Gauci, J., Howard, J., Ballantyne, I., Freeman, J., Kihira, T., Smith, T., Olmstead, D., McCutchan, J., Austin, C. & Pagella, A. 2014a. High-Performance JavaScript. In *HTML5 Game Development Insights*, Berkeley: Apress, Berkeley, CA, 43–57. doi: 10.1007/978-1-4302-6698-3_3.

McAnlis, C., Lubbers, P., Jones, B., Tebbs, D., Manzur, A., Bennett, S., D'Erfurth, F., Garcia, B., Lin, S., Popelyshev, I., Gauci, J., Howard, J., Ballantyne, I., Freeman, J., Kihira, T., Smith, T., Olmstead, D., McCutchan, J., Austin, C. & Pagella, A. 2014b. HTML5 Games in C++ with Emscripten. In *HTML5 Game Development Insights*, Berkeley: Apress, Berkeley, CA, 283–298. doi: 10.1007/978-1-4302-6698-3_18.

McAnlis, C., Lubbers, P., Jones, B., Tebbs, D., Manzur, A., Bennett, S., D'Erfurth, F., Garcia, B., Lin, S., Popelyshev, I., Gauci, J., Howard, J., Ballantyne, I., Freeman, J., Kihira, T., Smith, T., Olmstead, D., McCutchan, J., Austin, C. & Pagella, A. 2014c. JavaScript Is Not the Language You Think It Is. In *HTML5 Game Development Insights*, Berkeley: Apress, Berkeley, CA, 1–13. doi: 10.1007/978-1-4302-6698-3_18.

McManus, P. 2018. Bootstrapping WebSockets with HTTP/2. Retrieved from https://rfc-editor.org/rfc/rfc8441 [15 April 2024].

Mikkonen, T., Pautasso, C., Systä, K. & Taivalsaari, A. 2019. On the Web Platform Cornucopia. In *Web Engineering*, vol. 11496, Cham: Springer International Publishing, 347–355. doi: 10.1007/978-3-030-19274-7_25.

Miller, R.B. 1968. Response time in man-computer conversational transactions. In *AFIPS '68 (Fall, part I): Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, New York: Association for Computing Machinery, 267–277. doi: 10.1145/1476589.1476628.

Millhouse, T. 2018. Virtual Machines and Real Implementations. *Minds and machines*, 28(3): 465–489. doi: 10.1007/s11023-018-9472-7.

Minichiello, V., Aroni, R., Timewell, E. & Alexander, L. 1990. *In-depth Interviewing: Researching People*. Hong Kong: Longman Cheshire.

Møller, A. 2018. Technical Perspective: WebAssembly: A Quiet Revolution of the Web. *Communications of the ACM*, 61(12): 106. doi: 10.1145/3282508.

Mouton, J. 2011. *How to succeed in your master's and doctoral Studies: A South African guide* *and resource book*. 2nd ed. Hatfield: Van Shaik Publishers.

Musch, M., Wressnegger, C., Johns, M. & Rieck, K. 2019a. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 11543, Cham: Springer, 23–42. doi: 10.1007/978-3-030-22038-9_2.

Musch, M., Wressnegger, C., Johns, M. & Rieck, K. 2019b. Thieves in the Browser: Web-Based Cryptojacking in the Wild. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*. ARES '19. New York: Association for Computing Machinery: 1–10. doi: 10.1145/3339252.3339261.

Myers, M.D. & Venable, J.R. 2014. A set of ethical principles for design science research in information systems. *Information & Management*, 51(6): 801–809. doi: 10.1016/j.im.2014.01.002.

Ménétrey, J., Pasin, M., Felber, P. & Schiavoni, V. 2021. TWINE: An Embedded Trusted Runtime for WebAssembly. *2021 IEEE 37th International Conference on Data Engineering (ICDE)*: 205–216. doi: 10.1109/ICDE51399.2021.00025.

Na, Y., Kim, S.W. & Han, Y. 2016. JavaScript Parallelizing Compiler for Exploiting Parallelism from Data-Parallel HTML5 Applications. *ACM Transactions on Architecture and Code Optimization*, 12(4): 1–25. doi: 10.1145/2846098.

Nicula, S. & Zota, R.D. 2022. An Analysis of Different Browser Attacks and Exploitation Techniques. In *Education, Research and Business Technologies*, Singapore: Springer, 31–41. doi: 10.1007/978-981-16-8866-9_3.

Nießen, T., Dawson, M., Patros, P. & Kent, K.B. 2020. Insights into WebAssembly: compilation performance and shared code caching in Node.js. *CASCON 2020: Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*: 163–172.

Odell, D. 2014. Boosting JavaScript Performance. In *Pro JavaScript Development: Coding, Capabilities, and Tooling*, Berkeley: Apress, Berkeley, CA, 91–118. doi: 10.1007/978-1-4302-6269-5_4.

Ortiz, A. 2022. Using WebAssembly to Teach Code Generation in a Compiler Design Course. In *SIGCSE 2022: Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2*, New York: Association for Computing Machinery, 1167. doi: 10.1145/3478432.3499119.

Pace, D.S. 2021. Probability and Non-Probability Sampling - An Entry Point for Undergraduate Researchers. *International Journal of Quantitative and Qualitative Research Methods*, 9(2): 1–15.

Page, M.J., McKenzie, J.E., Bossuyt, P.M., Boutron, I., Hoffmann, T.C., Mulrow, C.D., Shamseer, L., Tetzlaff, J.M., Akl, E.A., Brennan, S.E., Chou, R., Glanville, J., Grimshaw, J.M., Hróbjartsson, A., Lalu, M.M., Li, T., Loder, E.W., Mayo-Wilson, E., McDonald, S., McGuinness, L.A., Stewart, L.A., Thomas, J., Tricco, A.C., Welch, V.A., Whiting, P. & Moher, D. 2021. The PRISMA 2020 statement: an updated guideline for reporting systematic reviews. *Systematic Reviews*, 10(1): 1–11. doi: 10.1186/s13643-021-01626-4.

Park, H., Cha, M. & Moon, S.M. 2016. Concurrent JavaScript Parsing for Faster Loading of Web Apps. *ACM Transactions on Architecture and Code Optimization*, 13(4): 1–24. doi: 10.1145/3004281.

Park, H., Kim, S. & Moon, S.M. 2017. Advanced Ahead-of-Time Compilation for Javascript Engine: Work-in-Progress. In *Proceedings of the 2017 International Conference on Compilers, Architectures and Synthesis for Embedded Systems Companion*. CASES '17. New York: Association for Computing Machinery: 1–2. doi: 10.1145/3125501.3125512.

Park, H., Kim, S., Park, J.G. & Moon, S.M. 2018. Reusing the Optimized Code for JavaScript Ahead-of-Time Compilation. *ACM Transactions on Architecture and Code Optimization*, 15(4): 1–20. doi: 10.1145/3291056.

Peffers, K., Rothenberger, M., Tuunanen, T. & Vaezi, R. 2012. Design Science Research Evaluation. In *Proceedings of the 7th International Conference on Design Science Research in Information Systems: Advances in Theory and Practice*. DESRIST 2012. Las Vegas: Springer-Verlag: 398–410. doi: 10.1007/978-3-642-29863-9_29.

Peffers, K., Tuunanen, T., Rothenberger, M.A. & Chatterjee, S. 2007. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3): 45–77. doi: 10.2753/MIS0742-1222240302.

Pierce, B.C. 2002. *Types and Programming Languages*. Cambridge: MIT Press.

Pinckney, D., Guha, A. & Brun, Y. 2020. Wasm/k: delimited continuations for WebAssembly. In *DLS 2020: Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, New York: Association for Computing Machinery, 16–28. doi: 10.1145/3426422.3426978.

Popek, G.J. & Goldberg, R.P. 1974. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7): 412–421. doi: 10.1145/361011.361073.

Powers, B., Vilk, J. & Berger, E.D. 2017. Browsix: Bridging the Gap Between Unix and the Browser. *ASPLOS 2017: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 52(4): 253–266. doi: 10.1145/3093336.3037727.

Puder, A., Woeltjen, V. & Zakai, A. 2013. Cross-compiling Java to JavaScript via tool-chaining. *PPPJ 2013: Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*: 25–34. doi: 10.1145/2500828.2500831.

Radhakrishnan, J. 2015. Hardware dependency and performance of JavaScript engines used in popular browsers. In *2015 International Conference on Control Communication & Computing India (ICCC)*. Trivandrum: Institute of Electrical and Electronics Engineers: 681–684. doi: 10.1109/ICCC.2015.7432981.

Rahimi, N. 2021. A Study of the Landscape of Security Issues, Vulnerabilities, and Defense Mechanisms in Web Based Applications. In *2021 International Conference on Computational Science and Computational Intelligence (CSCI)*. Las Vegas: Institute of Electrical and Electronics Engineers: 806–811. doi: 10.1109/CSCI54926.2021.00194.

Randal, A. 2020. The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers. *ACM Computing Surveys*, 53(1): 1–31. doi: 10.1145/3365199.

Reiser, M. & Bläser, L. 2017. Accelerate JavaScript Applications by Cross-Compiling to WebAssembly. *VMIL 2017: Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*: 10–17. doi: 10.1145/3141871.3141873.

Rempel, G. 2015. Defining Standards for Web Page Performance in Business Applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. New York: Association for Computing Machinery: 245–252. doi: 10.1145/2668930.2688056.

Rigger, M., Grimmer, M. & Mössenböck, H. 2016. Sulong - Execution of LLVM-Based Languages on the JVM: Position Paper. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS '16. Rome: Association for Computing Machinery: 1–4. doi: 10.1145/3012408.3012416.

Romano, A. & Wang, W. 2020. WASim: Understanding WebAssembly Applications through Classification. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. New York: Association for Computing Machinery: 1321–1325. doi: 10.1145/3324884.3415293.

Romano, A., Zheng, Y. & Wang, W. 2020. MinerRay: Semantics-Aware Analysis for Ever-Evolving Cryptojacking Detection. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. New York: Association for Computing Machinery: 1129–1140. doi: 10.1145/3324884.3416580.

Rose, S., Borchert, O., Mitchell, S. & Connelly, S. 2020. Zero Trust Architecture. *NIST Special Publication 800-207*: 1–59. doi: 10.6028/NIST.SP.800-207.

Rossberg, A. (ed.) 2022. *WebAssembly Specification Release 2.0 - Draft 2022-06-01*. Cambridge: World Wide Web Consortium.

Rossberg, A., Titzer, B.L., Haas, A., Schuff, D.L., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F. & Holman, M. 2018. Bringing the web up to speed with WebAssembly. *Communications of the ACM*, 61(12): 107–115. doi: 10.1145/3282510.

Salim, S.S., Nisbet, A. & Luján, M. 2020. TruffleWasm: A WebAssembly Interpreter on GraalVM. In *1st Workshop on Computer Architecture Research with RISC-V*. VEE '20. Lausanne: Association for Computing Machinery: 88–100. doi: 10.1145/3381052.3381325.

Sanders, I., Pilkington, C. & Pretorius, L. 2022. Making Research Methodologies in Theoretical Computing Explicit. *South African Computer Journal*, 34(1): 192–216. doi: 10.18489/sacj.v34i1.881.

Saunders, M.N.K., Lewis, P. & Thornhill, A. 2019. *Research Methods for Business Students*. 8th ed. London: Pearson.

Selakovic, M. & Pradel, M. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. New York: Association for Computing Machinery: 61–72. doi: 10.1145/2884781.2884829.

Serrano, M. 2018. JavaScript AOT Compilation. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*. DLS 2018. New York: Association for Computing Machinery: 50–63. doi: 10.1145/3276945.3276950.

Serrano, M. 2021. Of JavaScript AOT Compilation Performance. *Jan*, 5(70): 1–30. doi: 10.1145/3473575.

Sharrock, R., Angrave, L. & Hamonic, E. 2018. WebLinux: a scalable in-browser and client-side Linux and IDE. *L@S 2018: Proceedings of the Fifth Annual ACM Conference on Learning at Scale*: 1–2. doi: 10.1145/3231644.3231703.

Shepherd, C. & Markantonakis, K. 2024. Operating System Controls. In *Trusted Execution Environments*, Cham: Springer, 33–53. doi: 10.1007/978-3-031-55561-9_3.

Sipser, M. 2012. *Introduction to the Theory of Computation*. 3rd ed. Boston: Cengage Learning.

Skiena, S.S. 2008. *The Algorithm Design Manual*. 2nd ed. London: Springer-Verlag.

Smith, J. & Nair, R. 2005. The Architecture of Virtual Machines. *Computer*, 38(5): 32–38. doi: 10.1109/MC.2005.173.

Song, J., Ahn, M., Lee, G., Seo, E. & Jeong, J. 2021. A Performance-Stable NUMA Management Scheme for Linux-Based HPC Systems. *IEEE Access*, 9: 52987–53002. doi: 10.1109/ACCESS.2021.3069991.

South Africa. 2013. Protection of Personal Information Act, No. 4 of 2013. *Government Gazette*, 581(37067): 1–76.

Southern, G. & Renau, J. 2016. Overhead of Deoptimization Checks in the V8 JavaScript Engine. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. Providence: Institute of Electrical and Electronics Engineers: 1–10. doi:

10.1109/IISWC.2016.7581268.

Spies, B. & Mock, M. 2021. An Evaluation of WebAssembly in Non-Web Environments. In *2021 XLVII Latin American Computing Conference (CLEI)*. Cartago: Institute of Electrical and Electronics Engineers: 1–10. doi: 10.1109/CLEI53233.2021.9640153.

Stiévenart, Q., De Roover, C. & Ghafari, M. 2022. Security Risks of Porting C Programs to Webassembly. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. SAC '22. Virtual: Association for Computing Machinery: 1713–1722. doi: 10.1145/3477314.3507308.

Sun, J., Cao, D., Liu, X., Zhao, Z., Wang, W., Gong, X. & Zhang, J. 2019. SELWasm: A Code Protection Mechanism for WebAssembly. In *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. Xiamen: Institute of Electrical and Electronics Engineers: 1099–1106. doi: 10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00157.

Sun, K. & Ryu, S. 2018. Analysis of JavaScript Programs: Challenges and Research Trends. *ACM Computing Surveys*, 50(4): 1–34. doi: 10.1145/3106741.

Sun, Y. 2019. Server-Side Rendering. In *Practical Application Development with AppRun: Building Reliable, High-Performance Web Apps Using Elm-Inspired Architecture, Event Pub-Sub, and Components*, Berkeley: Apress, 191–217. doi: 10.1007/978-1-4842-4069-4_9.

Szabó, M. & Nehéz, K. 2019. C/C++ Applications on the Web. *Production Systems and Information Engineering*, 8: 69–87. doi: 10.32968/psaie.2019.005.

Szewczyk, R., Stonehouse, K., Barbalace, A. & Spink, T. 2022. Leaps and bounds: Analyzing WebAssembly's performance with a focus on bounds checking. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. Austin: Institute of Electrical and Electronics Engineers: 256–268. doi: 10.1109/IISWC55918.2022.00030.

Taft, D.K. 2003. Clash over Flash is heating up. *eWeek*, 20(46): 9–10.

Taivalsaari, A. & Mikkonen, T. 2017. The Web as a Software Platform: Ten Years Later. In *Proceedings of the 13th International Conference on Web Information Systems and Technologies*. WEBIST 2017. Porto: SciTePress: 41–50. doi: 10.5220/0006234800410050.

Taivalsaari, A. & Mikkonen, T. 2018. Return of the Great Spaghetti Monster: Learnings from a Twelve-Year Adventure in Web Software Development. In *Web Information Systems and Technologies*, vol. 322, Cham: Springer International Publishing, 21–44. doi: 10.1007/978-3-319-93527-0_2.

Taivalsaari, A., Mikkonen, T., Pautasso, C. & Systä, K. 2018. Client-Side Cornucopia: Comparing the Built-In Application Architecture Models in the Web Browser. In *Web Information Systems and Technologies*, vol. 372, Cham: Springer International Publishing, 1–24. doi:

10.1007/978-3-030-35330-8_1.

The Linux Foundation. 2005. SELinux Command Line Documentation. Retrieved from https://man7.org/linux/man-pages/man8/SELinux.8.html [15 April 2024].

The Linux Foundation. 2022. Linux Miscellaneous Information Manual - UTS Namespaces. Retrieved from https://man7.org/linux/man-pages/man7/uts_namespaces.7.html [15 April 2024].

The Linux Foundation. 2023a. Linux Miscellaneous Information Manual - Cgroup Namespaces. Retrieved from https://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html [15 April 2024].

The Linux Foundation. 2023b. Linux Miscellaneous Information Manual - Cgroups. Retrieved from https://man7.org/linux/man-pages/man7/cgroups.7.html [15 April 2024].

The Linux Foundation. 2023c. Linux Miscellaneous Information Manual - IPC Namespaces. Retrieved from https://man7.org/linux/man-pages/man7/ipc_namespaces.7.html [15 April 2024].

The Linux Foundation. 2023d. Linux Miscellaneous Information Manual - Mount Namespaces. Retrieved from https://man7.org/linux/man-pages/man7/mount_namespaces.7.html [15 April 2024].

The Linux Foundation. 2023e. Linux Miscellaneous Information Manual - Namespaces. Retrieved from https://man7.org/linux/man-pages/man7/namespaces.7.html [15 April 2024].

The Linux Foundation. 2023f. Linux Miscellaneous Information Manual - Network Namespaces. Retrieved from https://man7.org/linux/man-pages/man7/network_namespaces.7.html [15 April 2024].

The Linux Foundation. 2023g. Linux Miscellaneous Information Manual - PID Namespaces. Retrieved from https://man7.org/linux/man-pages/man7/pid_namespaces.7.html [15 April 2024].

The Linux Foundation. 2023h. Linux Miscellaneous Information Manual - Time Namespaces. Retrieved from https://man7.org/linux/man-pages/man7/time_namespaces.7.html [15 April 2024].

The Linux Foundation. 2023i. Linux Miscellaneous Information Manual - User Namespaces. Retrieved from https://man7.org/linux/man-pages/man7/user_namespaces.7.html [15 April 2024].

The Linux Foundation. 2023j. Linux System Calls Manual - Seccomp. Retrieved from https://man7.org/linux/man-pages/man2/seccomp.2.html [15 April 2024].

The Linux Foundation. 2024. Linux User Commands - Chroot. Retrieved from https://man7.org/

linux/man-pages/man1/chroot.1.html [15 April 2024].

Titzer, B.L. 2022. A fast in-place interpreter for WebAssembly. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2): 646–672. doi: 10.1145/3563311.

Topic, D. 2016. *Migrating from Java Applets to plugin-free Java technologies*. Oracle Corporation, Redwood Shores.

Tushar. & Mohan, B.R. 2022. Comparative Analysis Of JavaScript And WebAssembly In The Browser Environment. In *2022 IEEE 10th Region 10 Humanitarian Technology Conference (R10-HTC)*. Hyderabad: Institute of Electrical and Electronics Engineers: 232–237. doi: 10.1109/R10-HTC54060.2022.9929829.

Ueda, Y. & Ohara, M. 2017. Performance competitiveness of a statically compiled language for server-side Web applications. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Santa Rosa: Institute of Electrical and Electronics Engineers: 13–22. doi: 10.1109/ISPASS.2017.7975266.

Van Es, N., Stievenart, Q., Nicolay, J., D'Hondt, T. & De Roover, C. 2017. Implementing a performant scheme interpreter for the web in asm.js. *Computer Languages, Systems & Structures*, 49: 62–81. doi: 10.1016/j.cl.2017.02.002.

van Hasselt, M., Huijzendveld, K., Noort, N., de Ruijter, S., Islam, T. & Malavolta, I. 2022. Comparing the Energy Efficiency of WebAssembly and JavaScript in Web Applications on Android Mobile Devices. In *EASE 2022: The International Conference on Evaluation and Assessment in Software Engineering 2022*, New York: Association for Computing Machinery, 140–149. doi: 10.1145/3530019.3530034.

Venable, J., Pries-Heje, J. & Baskerville, R. 2012. A Comprehensive Framework for Evaluation in Design Science Research. In *Proceedings of the 7th International Conference on Design Science Research in Information Systems: Advances in Theory and Practice*. DESRIST 2012. Las Vegas: Springer-Verlag: 423–438. doi: 10.1007/978-3-642-29863-9_31.

Verdú, J. & Pajuelo, A. 2016. Performance Scalability Analysis of JavaScript Applications with Web Workers. *IEEE Computer Architecture Letters*, 15(2): 105–108. doi: 10.1109/LCA.2015.2494585.

Vilk, J. & Berger, E.D. 2014. Doppio: Breaking the Browser Language Barrier. *ACM SIGPLAN Notices*, 49(6): 508–518. doi: 10.1145/2666356.2594293.

Šipek, M., Muharemagić, D., Mihaljević, B. & Radovan, A. 2021. Next-generation Web Applications with WebAssembly and TruffleWasm. In *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*. Opatija: Institute of Electrical and Electronics Engineers: 1695–1700. doi: 10.23919/MIPRO52101.2021.9596883.

Wagner, L. 2017. Turbocharging the Web. *IEEE Spectrum*, 54(12): 48–53. doi: 10.1109/MSPEC.2017.8118483.

Wang, S., Ye, G., Li, M., Yuan, L., Tang, Z., Wang, H., Wang, W., Wang, F., Ren, J., Fang, D. & Wang, Z. 2019. Leveraging WebAssembly for Numerical JavaScript Code Virtualization. *IEEE Access*, 7: 182711–182724. doi: 10.1109/ACCESS.2019.2953511.

Wang, W. 2021. Empowering Web Applications with WebAssembly: Are We There Yet? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Melbourne: Institute of Electrical and Electronics Engineers: 1301–1305. doi: 10.1109/ASE51524.2021.9678831.

Wang, W. 2022. How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. Austin: Institute of Electrical and Electronics Engineers: 228–241. doi: 10.1109/IISWC55918.2022.00028.

Waterman, A. & Asanović, K. (eds.) 2019. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. Berkeley: RISC-V Foundation.

Watt, C. 2018. Mechanising and verifying the WebAssembly specification. *CPP 2018: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*: 53–65. doi: 10.1145/3167082.

Watt, C., Pulte, C., Podkopaev, A., Barbier, G., Dolan, S., Flur, S., Pichon-Pharabod, J. & Guo, S.y. 2020. Repairing and Mechanising the JavaScript Relaxed Memory Model. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York: Association for Computing Machinery: 346–361. doi: 10.1145/3385412.3385973.

Watt, C., Rossberg, A. & Pichon-Pharabod, J. 2019. Weakening WebAssembly. *Proceedings of the ACM on Programming Languages*, 3: 1–28. doi: 10.1145/3360559.

Wen, E., Warren, J. & Weber, G. 2020. BrowserVM: Running Unmodified Operating Systems and Applications in Browsers. In *2020 IEEE International Conference on Web Services (ICWS)*. Beijing: Institute of Electrical and Electronics Engineers: 473–480. doi: 10.1109/ICWS49710.2020.00070.

Wen, Y. & Wang, H. 2007. A Secure Virtual Execution Environment for Untrusted Code. In *Proceedings of the 10th international conference on Information security and cryptology*. ICISC 2007. Berlin: Springer-Verlag: 156–167. doi: 10.1007/978-3-540-76788-6_13.

Wieringa, R.J. 2014. *Design Science Methodology for Information Systems and Software Engineering*. Berlin: Springer-Verlag.

Wirfs-Brock, A. & Eich, B. 2020. JavaScript: The First 20 Years. *Proceedings of the ACM on Programming Languages*, 4(77): 1–189. doi: 10.1145/3386327.

Yan, Y., Tu, T., Zhao, L., Zhou, Y. & Wang, W. 2021. Understanding the performance of webassembly applications. In *IMC '21: Proceedings of the 21st ACM Internet*

Measurement Conference, New York: Association for Computing Machinery, 533–549. doi: 10.1145/3487552.3487827.

Yeaman, J. & Dawson, V. 1996. *Macromedia Shockwave for Director*. Indianapolis: Hayden Books.

Yin, J., Tan, G., Bai, X. & Hu, S. 2015. WebC: toward a portable framework for deploying legacy code in web browsers. *Science China Information Sciences*, 58(7): 1–15. doi: 10.1007/s11432-015-5285-y.

Yu, G., Yang, G., Li, T., Han, X., Guan, S., Zhang, J. & Gu, G. 2020. MinerGate: A Novel Generic and Accurate Defense Solution Against Web Based Cryptocurrency Mining Attacks. In *Cyber Security*, vol. 1299, Singapore: Springer, 50–70. doi: 10.1007/978-981-33-4922-3_5.

Yuki, T. 2014. Understanding PolyBench/C 3.2 Kernels. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*. IMPACT. Vienna: Association for Computing Machinery: 1–5.

Yuki, T. & Pouchet, L.N. 2016. PolyBench 4.2.1 (pre-release). Retrieved from https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1/blob/master/polybench.pdf [15 April 2024].

Zakai, A. 2011. Emscripten: an LLVM-to-JavaScript compiler. *OOPSLA 2011: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*: 301–312. doi: 10.1145/2048147.2048224.

Zakai, A. 2017. Why WebAssembly is Faster Than asm.js. Retrieved from https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/ [15 April 2024].

Zakai, A. 2018. Fast Physics on the Web Using C++, JavaScript, and Emscripten. *Computing in Science & Engineering*, 20(1): 11–19. doi: 10.1109/MCSE.2018.110150345.

Zhao, T., Berger, A. & Li, Y. 2019. Concurrency Control of JavaScript with Arrows. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS 2019. New York: Association for Computing Machinery: 1–10. doi: 10.1145/3358503.3361273.

Zhuykov, R. & Sharygin, E. 2017. Ahead-of-time compilation of JavaScript programs. *Programming and Computer Software*, 43(1): 51–59. doi: 10.1134/S036176881701008X.

Zhuykov, R., Vardanyan, V., Melnik, D., Buchatskiy, R. & Sharygin, E. 2015. Augmenting JavaScript JIT with ahead-of-time compilation. In *2015 Computer Science and Information Technologies (CSIT)*. Yerevan: Institute of Electrical and Electronics Engineers: 116–120. doi: 10.1109/CSITechnol.2015.7358262.

Šipek, M., Mihaljević, B. & Radovan, A. 2019. Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. Opatija: Institute of

Electrical and Electronics Engineers: 1671–1676. doi: 10.23919/MIPRO.2019.8756917.

APPENDICES

APPENDIX A

BENCHMARKING SUITES

This section details all of the benchmarking suites discovered as part of the SLR.

Table A.1: Benchmarking Suites Discovered through the Quality Assessments

Name	**ACME Air[1] (ACME)**
Details	
A fictitious implementation of an airline called ACME Air.	

Name	**Algorithms Custom Set 1 (AL1)**
Algorithms	**Details**
1. Mandelbrot	Computes a Mandelbrot set with 10,000 iterations.
2. Matrix Multiplier	Multiplies 3,000 by 3,000 matrices.
3. Sort	Sorts an unknown amount of integer vectors.

Name	**Algorithms Custom Set 2 (AL2)**
Algorithms	**Details**
1. Factorial	Factorial function that executes 10,000! and iterates 10,000 times.
2. Fibonacci	Calculates the 45th Fibonacci number.
3. Matrix Multiplication	Multiplies 3 by 3 matrices for 100,000 iterations.

Name	**Algorithms Custom Set 3 (AL3)**
Algorithms	**Details**
1. NFib	Naive Fibonacci function that calculates nfib of 38.
2. Primes	Sieve program that calculates the 5,000th prime number.
3. Queens	A number of placements of 11 Queens on a 11 by 11 chess board.
4. Twice	Higher order function Twice, repeated 400 times.
5. Knights	Finds a Knights tour on a 5 by 5 chess board.
6. Match	Pattern matching (5 levels deep) repeated 3,000,000 times.
7. SPrimes	Sieve using Peano numbers, calculating the 280th prime number.
8. Eval	Sapl interpreter, with a sieve that calculates the 100th prime number.
9. Hamming	List taking the 1,000th Hamming number, repeated 4,000 times.
10. Parser	Prolog Parser Combinators parsing a 35,000 lines Prolog program.
11. Prolog	Prolog interpreter calculating a four generation family tree.
12. Sort	Insertion - 6,000 items, quick - 10,000 items and merge sort - 100,000 items.

Name	**Algorithms Custom Set 4 (AL4)**
Algorithms	**Details**
1. Hough Transform	Simplified standard variant of Hough transform (SHT).

(continued on next page)

[1] https://github.com/wasperf/acmeair

Table A.1: Benchmarking Suites Discovered through the Quality Assessments (continued)

Name	Algorithms Custom Set 5[2] (AL5)
Algorithms	**Details**
1. C-Ray	Ray tracing compute-intensive algorithm.

Name	Algorithms Custom Set 6 (AL6)
Algorithms	**Details**
1. Linpack[3]	A function that uses a matrix of 1,500 by 1,500.
2. JBox2D	A function that uses its own Piston benchmark.

Name	Algorithms Custom Set 7 (AL7)
Algorithms	**Details**
1. ArrayReverse	Computes the reverse of an array with 10,000 elements 999 times.
2. Fib	Computes the Fibonacci number of 40.
3. IsPrime	Tests if the prime number $2^{31} - 1$ is prime.
4. MergeSort	Sorts an array of 10,000 double elements using the merge sort algorithm.
5. Nsieve	Sieve of Eratosthenes that counts prime numbers between 2 and 39,999.
6. Simjs[4]	Generates one thousand random numbers using the SIM.js library.
7. TspDouble[5]	Nearest neighbor traveling salesman solver for the test data of Tanzania.
8. TspInt	The same as TspDouble but uses integer instead of double coordinates.

Name	Algorithms Custom Set 8 (AL8)
Algorithms	**Details**
1. almabench.js	Calculates the daily ephemeris for the 21th century.
2. bague.js[6]	Scheme benchmark that solves the Baguenaudier game.
3. basic.js	Tiny Basic interpreter.
4. boyer-scm.js	Boyer.js belongs to the Octane test suite.
5. earley-scm.js	Earley.js belong to the Octane test suite.
6. jpeg.js	JPEG encoder/decoder used to encode and decode a 16 by 16 image.
7. js-of-ocaml.js	OCaml game of life.
8. leval.js	Scheme interpreter.
9. marked.js	Markdown parser.
10. maze.js	Jigsaw game originally implemented in Scheme by O. Shivers.
11. minimatch.js	Popular npm package for file globing.
12. minimist.js	Most popular npm package for command line parsing.
13. moment.js	Time manipulation library.
14. qrcode.js	Builds QRCodes.
15. richards+.js	Objects wrapped with JavaScript proxies.
16. rho.js	Implementation of the Racket contract system.
17. uuid.js	Is another very popular npm package.
18. z80.js	A Z80 emulator.

Name	Algorithms Custom Set 9 (AL9)
Algorithms	**Details**
1. Fibonacci	Calculates the Fibonacci sequence for term n.
2. Collision Detection	Performs collision detection of simple 2D shapes.
3. MultiplyIntVec	Function that multiplies two integer vectors.
4. QuicksortInt	The popular divide-and-conquer sort algorithm.
5. ImageThreshold	Algorithm to classify pixels as ”dark” or ”light”.
6. VideoConvolute	Algorithm that applies a convolution matrix to a portion of a video.

Name	Algorithms Custom Set 10 (AL10)
Algorithms	**Details**
1. Fibonacci	Calculates the Fibonacci sequence for term 1 through 51.
2. IsPrime	Checks if a number between 1,009 and 2,124,749,677 is prime.

(continued on next page)

Table A.1: Benchmarking Suites Discovered through the Quality Assessments (continued)

Name	**Algorithms Custom Set 11 (AL11)**
Algorithms	**Details**
1. Fannkuch	From Computer Languages Benchmarks Game.
2. Fasta	From Computer Languages Benchmarks Game.
3. Primes	Tiny loop that calculates prime numbers.
4. Raytrace	Real-world code, from the sphereflake ray-tracer.
5. DLmalloc	Doug Lea's malloc that tests memory access and integer calculations.

Name	**Algorithms Custom Set 12 (AL12)**
Algorithms	**Details**
1. Skinny	Vertex skinning algorithm common in 3D graphics.
2. Box2D	Popular physics engine for games.
3. zlib	Popular compression library.

Name	**CHStone[7] (CHS)**
Algorithms	**Details**
1. DFAdd	Double-precision floating-point addition.
2. DFDiv	Double-precision floating-point division.
3. DFMul	Double-precision floating-point multiplication.
4. DFSin	Sine function for double-precision floating-point numbers.
5. MIPS	Simplified MIPS processor.
6. ADPCM	Adaptive differential pulse code modulation decoder and encoder.
7. GSM	Global System for Mobile communications linear predictive coding analysis.
8. JPEG	JPEG image decompression.
9. Motion	Motion vector decoding of the MPEG-2.
10. AES	Advanced encryption standard.
11. Blowfish	Data encryption standard.
12. SHA	Secure hash algorithm.

Name	**Computer Languages Benchmarks Game[8] (CLG)**
Algorithms	**Details**
1. n-body	Performs an N-body simulation of the Jovian planets.
2. fannkuch-redux	Repeatedly access a tiny integer-sequence.
3. fasta	Generate and write random DNA sequences.
4. spectral-norm	Calculate an eigenvalue using the power method.
5. reverse-complement	Read DNA sequences and write their reverse-complement.
6. mandelbrot	Generate a Mandelbrot set and write a portable bitmap.
7. k-nucleotide	Repeatedly update hashtables and k-nucleotide strings.
8. binary-trees	Allocate and deallocate many binary trees.

Name	**Computer Languages Benchmarks Game[9] Partial 1 (CL1)**
Algorithms	**Details**
1. fannkuch-redux	Indexed access to tiny integer sequence.
2. fasta	Generate and write random DNA sequences.

(continued on next page)

[2]https://github.com/vkoskiv/c-ray
[3]http://www.netlib.org/benchmark/linpackjava/
[4]http://www.simjs.com/ offline, can be accessed through the internet archive.
[5]https://goo.gl/D6VULu
[6]https://en.wikipedia.org/wiki/Baguenaudier

Table A.1: Benchmarking Suites Discovered through the Quality Assessments (continued)

Name	**JetStream**[10] **(JET)**
Algorithms	**Details**
1. 3d-cube-SP	3D cube rotation that tests arrays and floating-point math.
2. 3d-raytrace-SP	Simple raytracer that tests arrays and floating-point math.
3. acorn-wtb	JS-based string manipulation and regular expression performance tester.
4. ai-astar	A JS implementation of the A* search algorithm.
5. Air	A test that runs allocateStack on hot function bodies.
6. async-fs	Mock filesystem that stresses the performance of async iteration.
7. Babylon	Non-trivial JS string processing that creates non-trivial object graphs.
8. babylon-wtb	Computes the Abstract Syntax Tree of an input JS program.
9. base64-SP	Base64 encoder/decoder written in JS
10. Basic	Stresses performance of generator functions and finds prime numbers.
11. bomb-workers	Runs subtests of the SunSpider benchmark in parallel using Web Workers.
12. box2d	JS Box2D physics engine that tests floating point math and data structures.
13. cdjs	Measures the performance of over 200 CDx collision detection runs.
14. chai-wtb	An assertion library that is commonly used to write unit and integration tests.
15. coffeescript-wtb	CoffeeScript compiler testing string manipulation and regular expressions.
16. crypto	RSA cypher implemented in JS that tests integer math and arrays.
17. crypto-aes-SP	An AES implementation in JS that tests integer math.
18. crypto-md5-SP	An MD5 implementation in JS that tests interesting integer math idioms.
19. crypto-sha1-SP	An SHA-1 implementation in JS that tests interesting integer math idioms.
20. date-format-tofte-SP	JS library functions that test date and time formatting.
21. date-format-xparb-SP	Sophisticated date formatting and parsing library test.
22. delta-blue	Devirtualisation of JS code that uses idiomatic class hierarchy.
23. earley-boyer	Chart parser algorithm with variadic functions and object construction.
24. espree-wtb	JS parser testing string manipulation and regular expressions.
25. first-inspector-code-load	Measures the first-time parsing of a modern JS code base.
26. FlightPlanner	Aircraft flight plan parser that and computes distance, courses, and more.
27. float-mm.c	Floating point matrix multiplier.
28. gaussian-blur	Gaussian blur that tests numeric analysis speed and uses typed arrays.
29. gbemu	Gameboy emulator that tests typed array and property access performance.
30. gcc-loops-wasm	Loops used to tune the GCC and LLVM vectorisers, compiled to WASM.
31. hash-map	Apache Harmony-based HashMap that performs hash table functions.
32. HashSet-wasm	A WASM test that replays a set of hash table operations.
33. jshint-wtb	Static analysis tool that warns about errors and problems in JS code.
34. json-parse-inspector	Tests a set of objects that WebKit's Web Inspector parses.
35. json-stringify-inspector	Tests a set of objects that WebKit's Web Inspector stringifies.
36. lebab-wtb	Transcompiler that converts ES5 code into ES6/ES7 code.
37. mandreel	Tests the Bullet physics engine which is compiled to JS with Mandreel.
38. ML	Feedforward neural network trained with different activation functions.
39. multi-inspector-code-load	Measures the repeated parsing of a modern JS code base.
40. navier-stokes	Solar system simulation that tests math and object access performance.
41. n-body-SP	Fluid simulation that emphasises floating point array performance.

(continued on next page)

[7]http://www.ertl.jp/chstone/ offline, can be accessed through the internet archive.
[8]http://benchmarksgame.alioth.debian.org/ offline, can be accessed through the internet archive.
[9]https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html
[10]http://browserbench.org/JetStream/

Table A.1: Benchmarking Suites Discovered through the Quality Assessments (continued)

42. octane-code-load	Test of code load speed of the jQuery and Closure libraries.
43. octane-zlib	A test based on compiling zlib to JS using Emscripten.
44. OfflineAssembler	A lexer, parser, and AST layer of the offline assembler for JavaScriptCore.
45. pdfjs	JS-based PDF reader that tests array manipulation and bit operations.
46. prepack-wtb	A compile time JS source code optimiser.
47. quicksort-wasm	A sorting test compiled to WASM with Emscripten.
48. raytrace	JS Ray tracer that tests object construction and floating point math.
49. regex-dna-SP	Regular-expression-based solution to DNA manipulation.
50. regexp	A collection of regular expressions found by the V8 team in 2010.
51. richards	Martin Richard's system language ported to JS.
52. richards-wasm	Martin Richard's system language ported to a hybrid of WASM and JS.
53. segmentation	Parallel computation of a time series segmentation algorithm.
54. splay	Tests the manipulation of splay trees represented using plain JS objects.
55. stanford-crypto-aes	The AES hashing algorithm using the Stanford JS Crypto Library.
56. stanford-crypto-pbkdf2	The PBKDF2 hashing algorithm using the Stanford JS Crypto Library.
57. stanford-crypto-sha256	The SHA256 hashing algorithm using the Stanford JS Crypto Library.
58. string-unpack-code-SP	Unpacks various minified JS libraries.
59. tagcloud-SP	Parses JSON and generates markup for a tag cloud view of the data.
60. tsf-wasm	A Typed Stream Format implementation in WASM.
61. typescript	Tests how quickly Microsoft's TypeScript compiler can compile itself.
62. uglify-js-wtb	A JS parser, minifier, compressor, and beautifier.
63. UniPoker	5 card stud poker simulation using Unicode playing card code points.
64. WSL	An implementation of a GPU shading language written in JS.

Name	**JetStream Partial 1 (JE1)**
Algorithms	**Details**
1. cdjs	Measures the performance of over 200 CDx collision detection runs.

Name	**JSBench[11] (JSB)**
Algorithms	**Details**
1. Amazon	Real-world stress test of JS found within the Amazon website.
2. Facebook	Real-world stress test of JS found within the Facebook website.
3. Google	Real-world stress test of JS found within the Google website.
4. Twitter	Real-world stress test of JS found within the Twitter website.
5. Yahoo	Real-world stress test of JS found within the Yahoo website.

Name	**Kraken[12] (KRA)**
Algorithms	**Details**
1. ai-astar	A path-finding program that uses A* search.
2. audio-beat-detection	Identical to audio-fft, with an insignificant amount of extra stuff at the end.
3. audio-dft	A kernel that computes a Discrete Fourier Transform.
4. audio-fft	A kernel that computes a Fast Fourier Transform.
5. audio-oscillator	A kernel that does an unknown audio oscillation test.
6. imaging-gaussian-blur	A kernel that tests gaussion blur on a desaturated image.
7. imaging-darkroom	A kernel that tests image negative.
8. imaging-desaturate	A kernel that test image desaturation.

(continued on next page)

[11] http://jsbench.cs.purdue.edu/ offline, can be accessed through the internet archive.
[12] https://wiki.mozilla.org/Kraken

Table A.1: Benchmarking Suites Discovered through the Quality Assessments (continued)

9. json-parse-financial	JSON parser test on financial data.
10. json-stringify-tinderbox	Stringify an object 1,000 times that uses 450,000+ chars to express.
11. stanford-crypto-aes	An AES crypto test.
12. stanford-crypto-ccm	A CCM crypto test.
13. stanford-crypto-pbkdf2	A PBKDF2 test.
14. stanford-crypto-sha256	A SHA256 crypto test.

Name	**Larceny R7RS[13] (LAR)**
Algorithms	**Details**
1. tower-fib	Dual metacircular interpreters running a recursive Fibonacci of 16.
2. nqueens	Backtracking algorithm to solve the n-queens puzzle where n 1⁄4 11.
3. qsort	Uses the quicksort algorithm to sort 500,000 numbers.
4. hanoi	The classical Hanoi puzzle with problem size 25.
5. tak	Calculates the Takeuchi function (tak 35 30 20) using a recursive definition.
6. cpstak	Calculates the same tak function using a continuation-passing style.
7. ctak	Calculates the same cpstak function capturing the continuation with call/cc.
8. destruct	Test of destructive list operations (set-car! and set-cdr!).
9. array1	Test a lot of allocation/initialisation and copying of large 1D arrays.
10. mbrot	Generates a Mandelbrot set. Mainly a test of floating-point arithmetic.
11. primes	Computes primes less than 50,000 with a list-based sieve of Eratosthenes.

Name	**Octane[14] (OCT)**
Algorithms	**Details**
1. Richards	OS Kernel test that tests property load/store and function/method calls.
2. Deltablue	One-way constraint solver that tests polymorphism.
3. Raytrace	A Raytracer test.
4. Regexp	Tests based on regular expression operations from 50 popular web pages.
5. NavierStokes	2D NavierStokes equations solver that manipulates double precision arrays.
6. Crypto	Encryption and decryption that tests bit operations.
7. Splay	Splay trees that exercise the automatic memory management subsystem.
8. SplayLatency	Splay test that stresses the Garbage Collection subsystem of a VM.
9. EarleyBoyer	Classic Scheme that tests fast object creation and destruction.
10. pdf.js	Mozilla's JS-based PDF reader that tests arrays and typed arrays.
11. Mandreel	The 3D Bullet Physics Engine ported from C++ to JavaScript via Mandreel.
12. MandreelLatency	Mandreel with frequent time measurement checkpoints.
13. GB Emulator	Emulates the portable console's architecture while running a 3D simulation.
14. Code Loading	Measures a JS engine startup time after loading a large JS program.
15. Box2DWeb	The popular 2D physics engine that tests floating point math.
16. zlib	The zlib asm.js/Emscripten test running with workload 1.
17. Typescript	Typescript compiler that measures how long it takes to compile itself.

Name	**Octane Partial 1 (OC1)**
Algorithms	**Details**
1. Richards	OS Kernel test that tests property load/store and function/method calls.
2. Deltablue	One-way constraint solver that tests polymorphism.
3. Raytrace	A Raytracer test.

(continued on next page)

[13] http://www.larcenists.org/benchmarksAboutR7.html

[14] https://developers.google.com/octane offline, can be accessed through the internet archive.

Table A.1: Benchmarking Suites Discovered through the Quality Assessments (continued)

4. NavierStokes	2D NavierStokes equations solver that manipulates double precision arrays.
5. Crypto	Encryption and decryption that tests bit operations.
6. Splay	Splay trees that exercise the automatic memory management subsystem.

Name	**PolyBench/C**[15] **(PBC)**
Algorithms	**Details**
1. 2mm	2 Matrix Multiplications (alpha * A * B * C + beta * D).
2. 3mm	3 Matrix Multiplications ((A*B)*(C*D)).
3. adi	Alternating Direction Implicit solver.
4. atax	Matrix Transpose and Vector Multiplication.
5. bicg	BiCG Sub Kernel of BiCGStab Linear Solver.
6. cholesky	Cholesky Decomposition.
7. correlation	Correlation Computation.
8. covariance	Covariance Computation.
9. deriche	Edge detection filter.
10. doitgen	Multi-resolution analysis kernel (MADNESS).
11. durbin	Toeplitz system solver.
12. fdtd-2d	2-D Finite Different Time Domain Kernel.
13. gemm	Matrix-multiply C=alpha.A.B+beta.C.
14. gemver	Vector Multiplication and Matrix Addition.
15. gesummv	Scalar, Vector and Matrix Multiplication.
16. gramschmidt	Gram-Schmidt decomposition.
17. head-3d	Heat equation over 3D data domain.
18. jacobi-1D	1-D Jacobi stencil computation.
19. jacobi-2D	2-D Jacobi stencil computation.
20. lu	LU decomposition.
21. ludcmp	LU decomposition followed by Forward Substitution.
22. mvt	Matrix Vector Product and Transpose.
23. nussinov	Dynamic programming algorithm for sequence alignment.
24. seidel	2-D Seidel stencil computation.
25. symm	Symmetric matrix-multiply.
26. syr2k	Symmetric rank-2k update.
27. syrk	Symmetric rank-k update.
28. trisolv	Triangular solver.
29. trmm	Triangular matrix-multiply.

Name	**SPEC CPU 2006**[16] **(SP16)**
Algorithms	**Details**
1. 410.bwaves	Floating Point - Computes 3D transonic transient laminar viscous flow.
2. 416.gamess	Floating Point - Wide range Quantum chemical computations.
3. 433.milc	Floating Point - Gauge field generator for lattice gauge theory programs.
4. 434.zeusmp	Floating Point - Computational fluid dynamics of astrophysical phenomena.
5. 435.gromacs	Floating Point - Newtonian equations of motion for up to millions of particles.
6. 436.cactusADM	Floating Point - Einstein evolution equation solver.
7. 437.leslie3d	Floating Point - Computational Fluid Dynamics with 3D Linear-Eddy Model.

(continued on next page)

[15]https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1
[16]https://www.spec.org/cpu2006/

Table A.1: Benchmarking Suites Discovered through the Quality Assessments (continued)

8. 444.namd	Floating Point - Biomolecular systems with 92,224 apolipoprotein A-I atoms.
9. 447.dealII	Floating Point - Helmholtz-type equation solver using coefficients.
10. 450.soplex	Floating Point - Linear solver with a simplex algorithm and sparse algebra.
11. 453.povray	Floating Point - Rendering a 1280 by 1024 anti-aliased landscape image.
12. 454.calculix	Floating Point - Finite element for linear/nonlinear 3D structural applications.
13. 459.GemsFDTD	Floating Point - Solves the Maxwell equations in 3D.
14. 465.tonto	Floating Point - Molecular Hartree-Fock wavefunction calculation.
15. 470.lbm	Floating Point - Simulate incompressible fluids in 3D.
16. 481.wrf	Floating Point - Weather modeling from a 30km area over 2 days.
17. 482.sphinx3	Floating Point - Widely-known speech recognition system.
18. 400.perlbench	Integer - Workloads based on the Perl programming language.
19. 401.bzip2	Integer - In memory bzip2 compression/decompression algorithm.
20. 403.gcc	Integer - GCC code generator for Opeteron.
21. 429.mcf	Integer - Public transport scheduler using network simplex algorithm.
22. 445.gobmk	Integer - Simulation that plays a game of Go.
23. 456.hmmer	Integer - Protein sequence analysis using profile hidden Markov models.
24. 458.sjeng	Integer - Highly-ranked chess program that also plays several variants.
25. 462.libquantum	Integer - Quantum simulation of Shor's polynomial-time factorisation.
26. 464.h264ref	Integer - Implementation of the H.264/AVC videostream encoder.
27. 471.omnetpp	Integer - Event simulator to model a large Ethernet campus network.
28. 473.astar	Integer - Pathfinding library for 2D maps using the A* algorithm.
29. 483.xalancbmk	Integer - Transforms XML documents to other document types.

Name	**SPEC CPU 2017**[17] **(SP17)**
Algorithms	**Details**
1. (5/6)00.perlbench_(r/s)	Integer - Perl interpreter.
2. (5/6)02.gcc_(r/s)	Integer - GNU C compiler.
3. (5/6)05.mcf_(r/s)	Integer - Route planning.
4. (5/6)20.omnetpp_(r/s)	Integer - Discrete Event simulation - computer network.
5. (5/6)23.xalancbmk_(r/s)	Integer - XML to HTML conversion via XSLT.
6. (5/6)25.x264_(r/s)	Integer - Video compression.
7. (5/6)31.deepsjeng_(r/s)	Integer - Artificial Intelligence: alpha-beta tree search (Chess).
8. (5/6)41.leela_(r/s)	Integer - Artificial Intelligence: Monte Carlo tree search (Go).
9. (5/6)48.exchange2_(r/s)	Integer - Artificial Intelligence: recursive solution generator (Sudoku).
10. (5/6)57.xz_(r/s)	Integer - General data compression.
11. (5/6)03.bwaves_(r/s)	Floating Point - Explosion modeling.
12. (5/6)07.cactuBSSN_(r/s)	Floating Point - Physics: relativity.
13. 508.namd_r	Floating Point - Molecular dynamics.
14. 510.parest_r	Floating Point - Biomedical imaging: optical tomography with finite elements.
15. 511.povray_r	Floating Point - Ray tracing.
16. (5/6)19.lbm_(r/s)	Floating Point - Fluid dynamics.
17. (5/6)21.wrf_(r/s)	Floating Point - Weather forecasting.
18. 526.blender_r	Floating Point - 3D rendering and animation.
19. (5/6)27.cam4_(r/s)	Floating Point - Atmosphere modeling.
20. 628.pop2_s	Floating Point - Wide-scale ocean modeling (climate level).
21. (5/6)38.imagick_(r/s)	Floating Point - Image manipulation.

(continued on next page)

[17] https://www.spec.org/cpu2017/

Table A.1: Benchmarking Suites Discovered through the Quality Assessments (continued)

22. (5/6)44.nab_(r/s)	Floating Point - Molecular dynamics.
23. (5/6)49.fotonik3d_(r/s)	Floating Point - Computational Electromagnetics.
24. (5/6)54.roms_(r/s)	Floating Point - Regional ocean modeling.

Name	**SunSpider[18] (SUN)**
Algorithms	**Details**
1. 3d-cube	Graphical 3D rotating cube algorithm using DHTML.
2. 3d-morph	Graphical 3D morphing algorithm.
3. 3d-raytrace	Graphical 3D raytracing algorithm.
4. access-binary-trees	Binary tree allocation, traversal and deallocation algorithm.
5. access-fannkuch	Pancake flipping algorithm that calculates permutations of flips.
6. access-nbody	The N-body problem that models the orbits of Jovian planets.
7. access-nsieve	A prime number sieve using the sieve of Eratosthenes.
8. bitops-3bit-bits-in-byte	Bitwise operation for 3-bit per byte lookup.
9. bitops-bits-in-byte	Bitwise operation for bit per byte lookup.
10. bitops-bitwise-and	Bitwise AND operation algorithm.
11. bitops-nsieve-bits	Bitwise operation for prime number sieve.
12. controlflow-recursive	Recursion and looping operations using Fibonacci .
13. crypto-aes	Cryptography calculations for the AES hashing algorithm.
14. crypto-md5	Cryptography calculations for the MD5 hashing algorithm.
15. crypto-sha1	Cryptography calculations for the SHA1 hashing algorithm.
16. date-format-tofte	Various date formatting operations using JS date objects.
17. date-format-xparb	Various date formatting operations using JS date objects.
18. math-cordic	Coordinate rotation digital computer mathematical calculations.
19. math-partial-sums	Mathematical calculations for partial sums.
20. math-spectral-norm	Mathematical calculations for spectral norm of a matrix.
21. regexp-dna	Regular expression operations for DNA data.
22. string-base64	String operations on Base64 data.
23. string-fasta	String operations on random DNA sequences.
24. string-tagcloud	String operations to manipulate a string tag cloud.
25. string-unpack-code	String operations to encode/decode data.
26. string-validate-input	Input string validation operations.

Name	**Rosetta[19] (ROS)**
Details	
Various undisclosed algorithms taken from a corpus containing over 1,200 algorithm implementations.	

Name	**Rosetta Partial 1[20] (RS1)**
Algorithms	**Details**
1. Banker's Algorithm	Resource allocation and deadlock avoidance algorithm.
2. Addition Chains	Mathematical addition chain and star addition chain algorithms.
3. Aliquot Sequence Classifier	Mathematical Aliquot Sequence of a positive integer.
4. Babbage Problem	Find the smallest positive integer whose square ends in the digits 269,696.
5. Bitwise IO	Read and write bit sequences with the most significant bit first.
6. Eban Numbers	Find numbers with no letter ”e” in it when the number is spelled in English.

(continued on next page)

[18] https://webkit.org/perf/sunspider/sunspider.html offline, can be accessed through the internet archive.
[19] https://rosettacode.org/wiki/Rosetta_Code
[20] https://github.com/KTH/slumps/tree/master/utils/pipeline/benchmark4pipeline_c offline, no longer accessible.

Table A.1: **Benchmarking Suites Discovered through the Quality Assessments (continued)**

7. Flipping Bits Game	Convert an N-by-N square array of zeroes or ones into a target state.
8. Paraffins	An organic chemistry tree enumeration algorithm without repetitions.
9. Pascal Matrix Generation	A 2-D square matrix containing numbers from Pascal's triangle.
10. Resistor Mesh	Resistance calculator across a grid or 10 by 10 nodes.
11. Runlength Encoding	Implementation of a run length text compression encoder/decoder.
12. Zebra Puzzle	Tries to answer Einstein's Riddle of who owns the Zebra.

Name	**Rosetta Partial 2**[21] **(RS2)**
Algorithms	**Details**
1. Bead sorting	Sort an array of positive integers using the Bead Sort Algorithm.
2. Circle sorting	Sort an array of integers into ascending order using Circlesort.
3. Identifier sorting	Sort a list of OIDs, in their natural sort order.
4. Lexicographic sorting	Given an integer n, return n in lexicographical order.
5. Merge sorting	The merge sort is a recursive sort of order n*log(n).
6. Natural sorting	Sort a list of strings, in their natural sort order.
7. Quick sorting	Sort an array of elements using the quicksort algorithm.
8. Remove duplicates and sort	Remove all duplicates of a given array and sort.

Name	**WABench**[22] **(WAB)**
Algorithms	**Details**
1. gcc-loops	Loops used to tune GCC vectoriser.
2. hashset	Hash table operations of web page loading.
3. quicksort	Quick sort algorithm implementation.
4. tsf	Implementation of a typed stream format.
5. basicmath	Basic mathematical computations.
6. bitcount	Bit manipulations.
7. jpeg	JPEG image compression/decompression.
8. stringsearch	Searching given words in phrases.
9. blowfish	Symmetric block cipher.
10. rijndael	Block cipher with variable length keys.
11. sha	Secure hash algorithm.
12. adpcm	Adaptive differential pulse code modulation.
13. crc32	32-bit Cyclic Redundancy Check.
14. correlation	Correlation computation.
15. covariance	Covariance computation.
16. gemm	Matrix multiplication.
17. gemver	Vector multiplication and matrix addition.
18. gesummv	Scalar, vector and matrix multiplication.
19. symm	Symmetric matrix multiplication.
20. syr2k	Symmetric rank-2k operations.
21. syrk	Symmetric rank-k operations.
22. trmm	Triangular matrix multiplication.
23. 2mm	Two matrix multiplications.
24. 3mm	Three matrix multiplications.
25. atax	Matrix transpose and vector multiplication.

(continued on next page)

[21] https://github.com/greensoftwarelab/WasmBenchmarks
[22] https://github.com/wabench/wabench

Table A.1: Benchmarking Suites Discovered through the Quality Assessments (continued)

26. bicg	BiCG sub kernel of BiCGStab linear solver.
27. doitgen	Multiresolution analysis kernel.
28. mvt	Matrix vector product and transpose.
29. cholesky	Cholesky decomposition.
30. durbin	Toeplitz system solver.
31. gramschmidt	Gram-Schmidt.
32. lu	LU decomposition.
33. ludcmp	LU decomposition.
34. trisolv	Triangular solver.
35. deriche	Edge detection filter.
36. floyd-warshall	Computing shortest paths in a graph.
37. nussinov	Sequence alignment.
38. adi	Alternating direction implicit solver.
39. fdtd-2d	2-D finite-difference time-domain kernel.
40. heat-3d	Heat equation over 3D data domain.
41. jacobi-1d	1-D jacobi stencil computation.
42. jacobi-2d	2-D jacobi stencil computation.
43. seidel-2d	2-D seidel stencil computation.
44. bzip2	File compression/decompression.
45. espeak	Text-to-Speech synthesiser.
46. facedetection	Detecting human faces in images.
47. gnuchess	Chess-playing game.
48. mnist	A neural network for digit recognition.
49. snappy	Data compression/decompression library.
50. whitedb	Lightweight NoSQL database.

APPENDIX B

SOURCE CODE

This section details portions of the source code that makes up the SYS23 enclave and the bulk testing harnesses.

B.1. System23 secure computing

First, we have a Seccomp implementation which restricts all types of *system calls*. When used to test the benchmarks, it will cause the benchmarks to fail, as expected.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/seccomp.h>
#include <sys/prctl.h>

int main(int argc, char *argv[])
{
    printf("sys23-scmp-strict\n");

    if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT) != 0) {
        printf("Setting strict mode failed, terminating!\n");

        exit(1);
    }

    printf("Seccomp strict mode active.\n\n");

    execvp(argv[1], &argv[1]);

    exit(0);
}
```

Listing B.1: System23 Secure Computing Strict Test Filter in C

Next, we have a Seccomp implementation based on the SYS23 prototype. This implementation allows certain *system calls* to pass through and execute as expected. When used to test the benchmarks, it will allow the benchmarks to execute successfully.

```c
#include <seccomp.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/prctl.h>

scmp_filter_ctx ctx;

void _exit(int status)
{
    seccomp_release(ctx);

    exit(status);
}

void whitelist_syscall(int syscall)
{
    if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, syscall, 0) != 0) {
        printf("Adding syscall filter failed, terminating!\n");

        _exit(1);
    }

    return;
}

int main(int argc, char *argv[])
{
    printf("sys23-scmp-bpf\n");

    if ((ctx = seccomp_init(SCMP_ACT_KILL)) == NULL) {
        printf("Setting filter mode failed, terminating!\n");

        _exit(1);
    }

    // Needed by scmp-bpf-sys23 and the PolyBench/C benchmarks
    whitelist_syscall(SCMP_SYS(access));
    whitelist_syscall(SCMP_SYS(arch_prctl));
    whitelist_syscall(SCMP_SYS(brk));
    whitelist_syscall(SCMP_SYS(close));
    whitelist_syscall(SCMP_SYS(execve));
    whitelist_syscall(SCMP_SYS(exit_group));
    whitelist_syscall(SCMP_SYS(getrandom));
    whitelist_syscall(SCMP_SYS(mmap));
    whitelist_syscall(SCMP_SYS(mprotect));
    whitelist_syscall(SCMP_SYS(munmap));
    whitelist_syscall(SCMP_SYS(newfstatat));
    whitelist_syscall(SCMP_SYS(openat));
    whitelist_syscall(SCMP_SYS(prctl));
    whitelist_syscall(SCMP_SYS(pread64));
    whitelist_syscall(SCMP_SYS(prlimit64));
    whitelist_syscall(SCMP_SYS(read));
    whitelist_syscall(SCMP_SYS(rseq));
    whitelist_syscall(SCMP_SYS(seccomp));
    whitelist_syscall(SCMP_SYS(set_robust_list));
    whitelist_syscall(SCMP_SYS(set_tid_address));
    whitelist_syscall(SCMP_SYS(write));

    // Needed by time_benchmark and Bash shell commands
    whitelist_syscall(SCMP_SYS(clone));
    whitelist_syscall(SCMP_SYS(dup2));
    whitelist_syscall(SCMP_SYS(fadvise64));          // grep, sort
    whitelist_syscall(SCMP_SYS(fcntl));
    whitelist_syscall(SCMP_SYS(futex));
    whitelist_syscall(SCMP_SYS(getegid));
    whitelist_syscall(SCMP_SYS(geteuid));
    whitelist_syscall(SCMP_SYS(getgid));
    whitelist_syscall(SCMP_SYS(getgroups));          // awk
    whitelist_syscall(SCMP_SYS(getpgrp));
    whitelist_syscall(SCMP_SYS(getpid));
    whitelist_syscall(SCMP_SYS(getppid));
    whitelist_syscall(SCMP_SYS(getuid));
    whitelist_syscall(SCMP_SYS(ioctl));
    whitelist_syscall(SCMP_SYS(lseek));
    whitelist_syscall(SCMP_SYS(pipe2));
    whitelist_syscall(SCMP_SYS(pselect6));           // bc
    whitelist_syscall(SCMP_SYS(rt_sigaction));
    whitelist_syscall(SCMP_SYS(rt_sigprocmask));
    whitelist_syscall(SCMP_SYS(rt_sigreturn));
    whitelist_syscall(SCMP_SYS(sched_getaffinity)); // grep, sort
    whitelist_syscall(SCMP_SYS(setfsgid));           // bc
    whitelist_syscall(SCMP_SYS(setfsuid));           // bc
    whitelist_syscall(SCMP_SYS(sigaltstack));        // grep
    whitelist_syscall(SCMP_SYS(sysinfo));
    whitelist_syscall(SCMP_SYS(uname));
    whitelist_syscall(SCMP_SYS(unlinkat));           // rm
    whitelist_syscall(SCMP_SYS(wait4));

    if (seccomp_load(ctx) != 0) {
        printf("Loading context failed, terminating!\n");

        _exit(1);
    }

    printf("Seccomp filter mode active.\n\n");

    execvp(argv[1], &argv[1]);

    _exit(0);
}
```

Listing B.2: System23 Secure Computing Pass-Through Test Filter in C

Next, we have three components which when executed together configures and then boots the SYS23 enclave. First, we configure the SYS23 enclave by creating the required Cgroups. In this instance it will enforce that the SYS23 enclave is pinned to CPU 4, is allowed to use 100%

of the CPU capacity and is allowed to use a maximum of 137MB of memory.

```sh
#!/bin/sh

USE_CPU=3
CPU_MAX=100
MEM_MAX=137
CGRP_CAT="cpu,cpuset,memory"
CGRP_ROOT_PATH="/system23"
CGRP_PATH1="${CGRP_ROOT_PATH}/enclave-1"
CGRP_PATH2="${CGRP_ROOT_PATH}/enclave-2"

cgcreate -g ${CGRP_CAT}:${CGRP_ROOT_PATH}
cgcreate -g ${CGRP_CAT}:${CGRP_PATH1}
cgcreate -g ${CGRP_CAT}:${CGRP_PATH2}

cgset -r cpuset.cpus="${USE_CPU}" ${CGRP_PATH1}
cgset -r cpu.max="${CPU_MAX}000 100000" ${CGRP_PATH1}
cgset -r memory.max="${MEM_MAX}M" ${CGRP_PATH1}
```

Listing B.3: System23 Create Control Groups Script

```sh
#!/bin/sh

NAMESPACES="--cgroup --ipc --mount --net --pid --time --user --uts"
PID_OPTS="--fork --mount-proc"
USER_OPTS="--map-root-user --map-current-user"

unshare ${NAMESPACES} ${PID_OPTS} ${USER_OPTS} /bin/bash
```

Listing B.4: System23 Boot Enclave Script

```sh
#!/bin/sh

CGRP_CAT="cpu,cpuset,memory"
CGRP_PATH="/system23/enclave-1"

cgclassify -g ${CGRP_CAT}:${CGRP_PATH} $(pgrep --parent $(pidof unshare)
    bash)
```

Listing B.5: System23 Bind Control Groups to Enclave Script

B.2. Benchmarking harnesses

These benchmarking harnesses tie into the base PolyBench/C benchmarks. They execute them in bulk, whereby all 30 benchmarks will be run as per the desired rounds, while also ensuring we pin the benchmarks to the CPU that has been isolated. The first script shown in Listing B.6 executes the benchmarks in a native environment, this serves as our baseline performance against which all other benchmarks are to be compared.

```
#!/bin/sh

USE_CPU=3
ROUNDS=10
SLEEP_TIME=30

declare -a PBC=("2mm" "3mm" "adi" "atax" "bicg" "cholesky"
"correlation" "covariance" "deriche" "doitgen" "durbin" "fdtd-2d"
"floyd-warshall" "gemm" "gemver" "gesummv" "gramschmidt" "heat-3d"
"jacobi-1d" "jacobi-2d" "lu" "ludcmp" "mvt" "nussinov" "seidel-2d"
"symm" "syr2k" "syrk" "trisolv" "trmm")

for BENCHMARK in "${PBC[@]}"
do
    PIN="taskset --cpu-list ${USE_CPU}"
    CMD="${PIN} ./time_benchmark ./pbc_${BENCHMARK}"

    for ((I = 0 ; I < ${ROUNDS} ; I++ ))
    do
        echo "Round ${I}: ${CMD}"
        eval ${CMD}

        sleep ${SLEEP_TIME}
        printf "\n"
    done

    printf "%0.s=" {1..80}
    printf "\n\n"
done
```

Listing B.6: Bulk Native Benchmarking Script

The next script depicted in Listing B.7 executes the benchmarks within a WASM environment or VM. While the last script depicted in Listing B.8 executes the benchmarks within a SYS23 environment or enclave.

```
#!/bin/sh

USE_CPU=3
ROUNDS=10
SLEEP_TIME=30

declare -a PBC=("2mm" "3mm" "adi" "atax" "bicg" "cholesky"
"correlation" "covariance" "deriche" "doitgen" "durbin" "fdtd-2d"
"floyd-warshall" "gemm" "gemver" "gesummv" "gramschmidt" "heat-3d"
"jacobi-1d" "jacobi-2d" "lu" "ludcmp" "mvt" "nussinov" "seidel-2d"
"symm" "syr2k" "syrk" "trisolv" "trmm")

for BENCHMARK in "${PBC[@]}"
do
    PIN="taskset --cpu-list ${USE_CPU}"
    CMD="${PIN} ./time_benchmark 'node pbc_${BENCHMARK}.js'"

    for ((I = 0 ; I < ${ROUNDS} ; I++ ))
    do
        echo "Round ${I}: ${CMD}"
        eval ${CMD}

        sleep ${SLEEP_TIME}
        printf "\n"
    done

    printf "%0.s=" {1..80}
    printf "\n\n"
done
```

Listing B.7: Bulk WebAssembly Benchmarking Script

```
#!/bin/sh

ROUNDS=10
SLEEP_TIME=30

declare -a PBC=("2mm" "3mm" "adi" "atax" "bicg" "cholesky"
"correlation" "covariance" "deriche" "doitgen" "durbin" "fdtd-2d"
"floyd-warshall" "gemm" "gemver" "gesummv" "gramschmidt" "heat-3d"
"jacobi-1d" "jacobi-2d" "lu" "ludcmp" "mvt" "nussinov" "seidel-2d"
"symm" "syr2k" "syrk" "trisolv" "trmm")

for BENCHMARK in "${PBC[@]}"
do
    S23="./scmp-bpf-sys23"
    CMD="${S23} ./time_benchmark ./pbc_${BENCHMARK}"

    for ((I = 0 ; I < ${ROUNDS} ; I++ ))
    do
        echo "Round ${I}: ${CMD}"
        eval ${CMD}

        sleep ${SLEEP_TIME}
        printf "\n"
    done

    printf "%0.s=" {1..80}
    printf "\n\n"
done
```

Listing B.8: Bulk System23 Benchmarking Script

APPENDIX C

BENCHMARKING RUNBOOK

This section details the benchmarking runbook as used by this study to generate the data as represented in sections 5.1., 5.2. and 5.3. Note that at this stage all of the needed benchmarks, together with the SYS23 enclave source code where required, have already been compiled.

C.1. OS noise mitigations

We start off by configuring the OS noise mitigations. It is assumed that the user performing these commands has the required access level to do so, usually *root* level access is required.

1. **Enable CPU Isolation**: Isolate the highest two CPUs namely CPUs 3 and 4.

   Shell Command

   ```
   $ grubby --update-kernel DEFAULT --args="isolcpus=2,3"
   ```

   Confirm the CPU isolation after rebooting.

   Shell Command

   ```
   $ cat /sys/devices/system/cpu/isolated

   2-3
   ```

   Remove all system threads from CPUs 3 and 4.

   Shell Command

   ```
   $ tuna isolate --cpus=2,3
   ```

   Remove all workqueues from CPUs 3 and 4.

   Shell Command

   ```
   $ find /sys/devices/virtual/workqueue -name cpumask -exec echo 3 > {} \;
   ```

Reconfirm all CPU isolation settings by reviewing all thread context switches.

Shell Command

```
$ perf stat -e 'sched:sched_switch' -a -A --timeout 30000

 Performance counter stats for 'system wide':

CPU0                 1,179      sched:sched_switch
CPU1                 1,022      sched:sched_switch
CPU2                     3      sched:sched_switch
CPU3                     3      sched:sched_switch

      30.030770397 seconds time elapsed
```

2. **Enable Timer Tick Isolation**: Reduce the number of timer ticks for the isolated CPUs 3 and 4.

Shell Command

```
$ grubby --update-kernel DEFAULT --args="nohz_full=2,3"
```

Confirm the reduction of timer ticks after rebooting.

Shell Command

```
$ perf stat -e 'irq_vectors:local_timer_entry' -a -A --timeout 30000

 Performance counter stats for 'system wide':

CPU0                30,044      irq_vectors:local_timer_entry
CPU1                   627      irq_vectors:local_timer_entry
CPU2                     2      irq_vectors:local_timer_entry
CPU3                     2      irq_vectors:local_timer_entry

      30.027916330 seconds time elapsed
```

3. **Maximise CPU Frequency Scaling**: Maximise the CPU frequency scaling for CPUs 3 and 4.

Shell Command

```
$ echo performance > /sys/devices/system/cpu/cpufreq/policy2/scaling_governor
```

Shell Command

```
$ echo performance > /sys/devices/system/cpu/cpufreq/policy3/scaling_governor
```

Confirm the frequency scaling settings for CPUs 3 and 4.

Shell Command

```
$ cat /sys/devices/system/cpu/cpufreq/policy2/scaling_governor

performance
```

Shell Command

```
$ cat /sys/devices/system/cpu/cpufreq/policy3/scaling_governor

performance
```

4. **Enable Interrupt CPU Affinity**: Remove IRQ handling from the isolated CPUs 3 and 4.

Shell Command

```
$ irqbalance --foreground --oneshot
```

Confirm that the isolated CPUs no longer handle any IRQs.

Shell Command

```
$ watch cat /proc/interrupts

Every 2.0s: cat /proc/interrupts      example.com: Wed Jul 17 13:27:44 2024

CPU0         ...        CPU3
0:          42         ...          0    IO-APIC    2-edge       timer
1:           0         ...          0    IO-APIC    1-edge       i8042
8:           1         ...          0    IO-APIC    8-edge       rtc0
9:           0         ...          0    IO-APIC    9-fasteoi    acpi
...
```

5. **Disable Swaps**: Do not allow any memory swapping to disk.

Shell Command

```
$ swapoff -a
```

Shell Command

```
$ systemctl --type swap

  UNIT          LOAD   ACTIVE SUB    DESCRIPTION
  ---------------------------------------------------------------
dev-zram0.swap loaded active active Compressed Swap on /dev/zram0
...
```

Shell Command

```
$ systemctl stop 'dev-zram0.swap'
```

Shell Command

```
$ systemctl mask 'dev-zram0.swap'
```

Confirm that swapping to disk has been disabled.

Shell Command

```
$ swapon --show
```

6. **Disable Transparent Huge Pages**: Do not allow transparent huge page elevation.

Shell Command

```
$ echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

7. **Disable NUMA Memory Balancing**: Do not allow NUMA memory balancing.

Shell Command

```
$ echo 0 > /proc/sys/kernel/numa_balancing
```

8. **Disable Mitigations for CPU Vulnerabilities**: Remove all CPU vulnerability mitigations.

Shell Command

```
$ grubby --update-kernel DEFAULT --args="mitigations=off"
```

Confirm that all CPU vulnerability mitigations are removed.

Shell Command

```
$ lscpu

...

NUMA:
  NUMA node(s):            1
  NUMA node0 CPU(s):       0-3
Vulnerabilities:
  Gather data sampling:    Vulnerable
  Itlb multihit:           KVM: Mitigation: VMX disabled
  L1tf:                    Mitigation; PTE Inversion; VMX vulnerable
  Mds:                     Vulnerable; SMT vulnerable
  Meltdown:                Vulnerable
  Mmio stale data:         Vulnerable
  Reg file data sampling: Not affected
  Retbleed:                Vulnerable
  Spec rstack overflow:    Not affected
  Spec store bypass:       Vulnerable
  Spectre v1:              Vulnerable: __user pointer sanitization and
                           usercopy barriers only; no swapgs barriers
  Spectre v2:              Vulnerable; IBPB: disabled; STIBP: disabled;
                           PBRSB-eIBRS: Not affected; BHI: Not affected
  Srbds:                   Vulnerable
  Tsx async abort:         Not affected
```

C.2. Native benchmarks

Next, we proceed with executing the native PolyBench/C benchmarks and capturing their performance results. When executing the benchmarking script, one can confirm that the benchmarks have been pinned to CPU 3.

Shell Command

```
$ ./run-all-native

Round 0: taskset --cpu-list 3 ./time_benchmark ./pbc_2mm
[INFO] Running 5 times ./pbc_2mm...
[INFO] Maximal variance authorized on 3 average runs: 5%...
[INFO] Maximal deviation from arithmetic mean of 3 average runs: 0.67900%
[INFO] Normalized time: 2.36659066
```

C.3. WebAssembly benchmarks

Next, we proceed with executing the WASM-based PolyBench/C benchmarks and capturing their performance results. To confirm that the correct benchmarks are being executed, one can confirm that the `node` command is executing the JS-based version of the benchmarks, where `node` provides the benchmarks with the required WASM-based VM. Note that the full set of OS noise mitigations are still in effect.

Shell Command

```
$ ./run-all-wasm

Round 0: taskset --cpu-list 3 ./time_benchmark 'node pbc_2mm.js'
[INFO] Running 5 times node pbc_2mm.js...
[INFO] Maximal variance authorized on 3 average runs: 5%...
[INFO] Maximal deviation from arithmetic mean of 3 average runs: 0.31400%
[INFO] Normalized time: 5.19333333
...
```

C.4. System23 benchmarks

Finally, we proceed with executing the native SYS23-based PolyBench/C benchmarks and capturing their performance results. We start by configuring and booting the prototype SYS23 enclave, after which we proceed with executing the benchmarks using the SYS23 Seccomp application. Note that we do not need to pin the benchmarks to CPU 3 using `taskset` as the SYS23 enclave has that configured as part of the Cgroups configurations.

Shell Command

```
$ ./sys23-create-cgroups
```

This step boots the SYS23 enclave using the *namespaces* required and will create its own *shell* environment.

Shell Command

```
$ ./sys23-boot-enclave
```

The next step needs to be executed outside of the SYS23 enclave that was just booted, so as to bind the Cgroup configurations to the running SYS23 enclave.

Shell Command

```
$ ./sys23-bind-cgroups-to-enclave
```

Once completed, a fully configured SYS23 enclave should be active and ready to execute the PolyBench/C benchmarks as follows. Note that as before, the full set of OS noise mitigations are still in effect.

Shell Command

```
$ ./run-all-sys23

Round 0: ./sys23-scmp-bpf ./time_benchmark ./pbc_2mm
sys23-scmp-bpf
Seccomp filter mode active.

[INFO] Running 5 times ./pbc_2mm...
[INFO] Maximal variance authorized on 3 average runs: 5%...
[INFO] Maximal deviation from arithmetic mean of 3 average runs: 0.14500%
[INFO] Normalized time: 2.36739066
...
```