WArdian: Securing WebAssembly Applications on Untrusted Mobile Operating Systems

José Francisco Oliveira Pinto Malafaia Canana

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor(s): Prof. Nuno Miguel Carvalho dos Santos

Examination Committee

Chairperson: Prof. Mário Jorge Costa Gaspar da Silva
Supervisor: Prof. Nuno Miguel Carvalho dos Santos
Member of the Committee: Prof. Bernardo Luís da Silva Ferreira

November 2020

Acknowledgments

First of all, I'm very grateful to my professor advisor Nuno Santos for all the patience, guidance and support during this past year and for the overall availability to aid in all my difficulties and setbacks during the development of *WArdian*.

Also, a big tank you to David Cerdeira, and the entire team at Universidade do Minho, for providing us access to the Bao hypervisor and for the countless hours of help, debug and integration of their project into ours.

I'm also very grateful to my friends and colleagues at Instituto Superior Técnico, specially Luís Tonicha, Leonardo Vieira and Rafael Moráis dos Santos, for all the fruitful discussions, comments and brainstorms that considerably improved my final project.

To my parents and grandparents, I am very grateful for all the unconditional support and obviously for all the financial funds which without would have made my academic path an impossibility.

And last, but definitely not least, a big thank you to my girlfriend Margarida Récio for all the added motivation and positive pressure needed to finish this thesis.

Resumo

Os dispositivos móveis são extremamente populares e usados para correr várias aplicações que trabalham com dados sensíveis ao utilizador. Graças ao desenvolvimento de WebAssembly, uma tecnologia emergente com suporte para a maioria dos browsers modernos, vastas aplicações deixam de necessitar uma instalação prévia no dispositivo. Desta nova forma, necessitam apenas de ser transferidas, em tempo real, para o browser utilizado e conseguem correr localmente no dispositivo. Isto é possível visto que WebAssembly traz aos web developers a possibilidade de correr código nativo C/C++ compilado directamente para a web page, o que corre muito mais rapidamente do que típico JavaScript. Utilizando esta nova técnica, os antigos mecanismos de encapsulamento dos browsers modernos não serão suficientes para tolerar ataques provenientes de um adversário que consiga comprometer o estado do sistema operativo do dispositivo móvel. Por exemplo, caso um dispositivo Android seja atacado por um rootkit, o adversário consegue comprometer a integridade e confidencialidade da informação trabalhada pela aplicação de WebAssembly, mesmo que devidamente encapsulada pelo browser. Este trabalho apresenta a concepção, implementação e avaliação de *WArdian*, um ambiente de execução seguro para aplicações de WebAssembly em sistemas operativos móveis não confiáveis. Aproveitando um hipervisor existente, pretendemos desconstruir o Chrome browser e correr WebAssembly em memória protegida – intitulada *Web Cage* – onde o sistema operativo não apresenta privilégios.

Palavras-chave: WebAssembly, Chrome browser, Android, Encapsulamento, Hipervisor, Virtualização

Abstract

Mobile devices are extremely popular and are used for running many data sensitive applications. Thanks to the development of WebAssembly, an emerging browser technology, applications no longer need to be installed on the device. Instead, they can be download on-the-fly and executed directly on the local browser. This is possible because WebAssembly enables web developers to run native C/C++ code on a web page, which runs much faster than typical JavaScript. Using this new approach, the previously implemented sandboxing mechanisms of modern web browsers will not be enough to prevent attacks from an adversary that can compromise the mobile operating system. For instance, if the android OS is attacked by a rootkit, the adversary could compromise the integrity and confidentiality of data manipulated by the sandboxed WebAssembly application. This work presents the design, implementation, and evaluation of *Wardian*, a secure execution environment to run WebAssembly applications on untrusted mobile operating systems. Leveraging on an existent hypervisor, we intend to deconstruct the Chrome browser and run WebAssembly code inside a protected memory space – named *web cage* – where the operating system holds no privileges.

Keywords: WebAssembly, Chrome browser, Android OS, Sandboxing, Hypervisor, Virtualization

Contents

Acknowledgments . . . iii
Resumo . . . v
Abstract . . . vii
List of Tables . . . xiii
List of Figures . . . xv

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objectives . . . 2
1.3 Contributions . . . 2
1.4 Thesis Outline . . . 3

2 Background **5**
2.1 WebAssembly . . . 5
2.2 Chrome Browser Architecture . . . 6
2.2.1 Process models . . . 6
2.2.2 Sandboxing the renderer . . . 7
2.2.3 Inter-process Communication . . . 9
2.2.4 V8 . . . 9
2.3 The *Bao* hypervisor . . . 10

3 Related Work **13**
3.1 Protection from Untrusted Applications . . . 13
3.1.1 Taint tracking of JavaScript and WebAssembly code . . . 13
3.1.2 Same Origin Policy and Site Isolation . . . 14
3.1.3 Secure UI . . . 15
3.1.4 Browser hardening . . . 15
3.1.5 Dynamic monitoring of browser extensions . . . 16
3.2 Protection from Untrusted Environments . . . 17
3.2.1 Memory-safe compilers . . . 17
3.2.2 Hardware enclaves . . . 18
3.2.3 Rollback protection . . . 20

3.2.4 Obfuscated execution . . . 20
3.2.5 TrustZone-assisted TEEs . . . 21

4 WArdian Design **23**
4.1 Motivational Example and Threat Model . . . 23
4.2 System Overview . . . 24
4.3 WArdian API . . . 27
4.4 Extending the Browser . . . 29
4.5 Securing Wasm Execution . . . 30
4.6 Communication Channels . . . 31
4.6.1 Native Messaging . . . 32
4.6.2 Shared Memory . . . 33
4.7 Secure Bootstrap . . . 33

5 Implementation **35**
5.1 WArdian Prototypes . . . 35
5.1.1 QEMU Prototype . . . 35
5.1.2 Rock960 Prototype . . . 36
5.2 Chrome Modifications . . . 37
5.2.1 Web Page Content Access . . . 38
5.2.2 Changing JavaScript Behavior . . . 39
5.2.3 Wasm Data Structures . . . 40
5.3 Enclave Environment . . . 40
5.3.1 Embedding WAMR in Zephyr . . . 40
5.3.2 Communication with the Extension . . . 41
5.3.3 Dynamic Enclave Execution . . . 42
5.4 Bao Integration . . . 43
5.4.1 Hardware Setup . . . 43
5.4.2 Communication Protocols . . . 45
5.4.3 Prototyping Problems . . . 47

6 Evaluation **49**
6.1 Evaluation Goals . . . 49
6.2 Performance . . . 50
6.2.1 Global Performance . . . 50
6.2.2 Micro-benchmarks . . . 52
6.3 Memory Footprint and Code Size . . . 54
6.4 Usability . . . 54
6.5 Security Analysis . . . 55

7 Conclusions **57**
7.1 Achievements 57
7.2 Future Work 58

Bibliography **61**

List of Tables

5.1 QEMU prototypes performance setback. 42

6.1 *WArdian* QEMU Micro-benchmarking. 52
6.2 *WArdian* Simulated Bao Micro-benchmarking. 53
6.3 *WArdian*'s source lines of code (SLoC). 54

List of Figures

2.1 WebAssembly compilation process. 5
2.2 Chrome multi-process architecture. 7
2.3 Renderer process isolation 8
2.4 *Bao* hypervisor architecture. 10

3.1 Site Isolation. 14
3.2 Fidelius architecture. 19
3.3 ARM TrustZone Architecture for Cortex-A. 21

4.1 Bitcoin Wallet Example. 24
4.2 *WArdian* architecture 25
4.3 Simple *wasm* web page. 26
4.4 Chrome extensions page. 27
4.5 *WArdian* extension architecture. 29
4.6 *WArdian* communication channels. 32

5.1 QEMU prototype architecture. 36
5.2 *Wasm* testing web page. 36
5.3 *WArdian* prototyping boards. 37
5.4 Rock960 prototype architecture. 37
5.5 Socket communication improvement. 43
5.6 Debug wiring diagram. 44
5.7 Android OS capture. 44
5.8 UART debug configuration. 45

6.1 *WArdian* performance testing web pages. 50
6.2 *WArdian* global performance. 51

Chapter 1

Introduction

Nowadays, mobile devices have a strong influence in our personal connections, and therefore, these devices hold several applications that manage sensitive data, such as banking applications, private social messengers and so on. With the development of WebAssembly [1], an emerging browser technology, developers can now run native C/C++ code directly on the browser and make use of its faster performance when comparing it to typical JavaScript. This technology makes possible the creation of applications that run in the local browser, and therefore do not need to be physically installed on the device. Consequently, browser security is extremely important to prevent adversaries from gaining access to web pages' state and steal users' private information, for instance, prevent access to the user's private keys of a BitCoin wallet software implemented as a downloadable WebAssembly module. This thesis focuses on securing browser WebAssembly executions, specifically on the Chrome mobile version, in an environment completely isolated from the browser and the underlying untrusted operating system.

1.1 Motivation

Currently, Chrome uses a multi-process sandboxing environment to protect the user against attackers that can manipulate web pages and exploit bugs in the renderer process [2]. Due to the nature of web attacks that intend to escape the sandbox environment, Chrome's isolation techniques leverage on the existent operating system (OS) security measures to create and maintain this secured environment. In Linux specifically, the kernel sandboxing environments play an important role in the overall Chrome's isolation and security. In the remaining of this project, we will focus on the Chrome browser and how to protect it against attacks on the Android OS.

Unfortunately, modern browsers do not have the necessary security measures to protect the execution state of browser-hosted application from an adversary that can compromise the integrity of the operating system of the mobile device [3, 4]. Since the dependency on the operating system security measures is so ingrained in Chrome's sandboxing procedures, an attacker that can successfully compromise the operating system can utilize the browser for more complex attacks on the device. For instance, if the Android OS is attacked by a rootkit, a malicious agent will be able to extract secrets and other

sensitive data manipulated by the WebAssembly application running inside a browser's sandbox.

Considering that this class of attacks to the operating system is far from uncommon [5–8] and that the sophistication of WebAssembly applications will naturally lead to the processing of larger amounts of sensitive data, providing a strong line of defense mechanisms is fundamental.

1.2 Objectives

In this work, our goal is to address browser security by isolating WebAssembly executions without degrading the overall performance of the Chrome browser nor the underlying operating system's. As browsers manage private user information, and mobile devices are extremely populated by such data, WebAssembly code execution needs to be isolated to ensure data confidentiality and integrity. Taking it a step further, a compromised operating system can take advantage of the modern browser architecture to have unlimited access to such data.

Our approach is to combine TrustZone-technology with partitioned execution of the Chrome browser. In particular, by using a TrustZone-assisted hypervisor and creating isolated memory zones for securing WebAssembly programs execution inside the Chrome browser, we aim to ensure that an untrusted Android OS cannot tamper with the data processed by such programs. As for partitioned execution, the idea is to separate Chrome's renderer process into two parts: one runs normally on the Android OS, while the other partition provides for the instantiation of *web cages*, i.e., trusted execution environments that load and run WebAssembly on a secure memory zone created by the hypervisor. A set of drivers and API's must be created to allow for the communication between both zones while preserving the integrity of the secured space.

1.3 Contributions

This thesis presents *WArdian*, a system that provides a trusted execution environment within the browser for loading, interpreting, and executing WebAssembly code completely isolated from the host operating system. Our solution leverages an existing TrustZone-assisted hypervisor that can create protected memory spaces where the operating system holds no privileges. By partitioning the Chrome browser and learning from their existent sandboxing mechanisms we have created *web cages*, sandboxing environments for running WebAssembly code inside secure memory zones provided by the used hypervisor.

In *WArdian*, the deployed WebAssembly code is able to interact with the remaining components of the web page located in the browser and renderer's process outside the secure memory zone (i.e., DOM and JavaScript code), and have access to security services that will prevent specialized attacks potentially issued by an untrusted operating system, e.g. tampering with the system clock, file system, or random number generation.

In summary, this thesis makes the following contributions:

- A new browser architecture featuring separation of privileges between the Android OS and trusted

execution environments for hosting WebAssembly code;

- Implementation of this architecture for the Chrome browser's version targeting mobile devices;
- Improvement on mobile browser security with modest performance degradation;
- An extensive evaluation with real use case web applications.

1.4 Thesis Outline

The rest of this document is organized as follows. Chapters 2 and 3 present some background and related work, respectively, where Chrome and WebAssembly internals are introduced and the current state of vulnerabilities and possible solutions are analyzed. Chapter 4 describes the proposed architecture of *WArdian*, and Chapter 5 describes the steps taken in order to implement it. Chapter 6 presents the evaluating process of *WArdian* and the consequent results obtained. Finally, Chapter 7 concludes this report and describes possible future work implementations to extend *WArdian* functionalities.

Chapter 2

Background

This chapter provides some necessary background on our work. We begin by introducing WebAssembly and discussing its potential for speeding up the Web. Then, we present the internals of a modern browser – Chrome – focusing in particular on its security mechanisms. Lastly, we present a brief introduction on *Bao*, our used hypervisor, and its security mechanisms.

2.1 WebAssembly

WebAssembly, usually shortened to *wasm*, is a low level binary format that can be executed in most of modern browsers. Its small-sized footprint, due to being optimized and precompiled, makes it faster to load and execute than typical JavaScript code (20x-40x faster) [9] which makes it more desirable to use for compute-intensive workloads and all sorts of browser applications [1, 10].

In browsers today, JavaScript is executed inside a virtual machine which optimizes the code to improve performance. JavaScript is one of the fastest dynamic languages but cannot compete with native C/C++ raw code. WebAssembly runs in the same virtual machine but its performance is much better.

To take advantage of this new technology, web developers can compile their native code into a *wasm module* as described in Figure 2.1. This module will then be glued to the HTML page using intermediary JavaScript code so that finally the application can be presented on the browser. In order to compile the C code into *wasm code*, a precompiled toolchain is used, the most popular being Emscripten [11, 12].

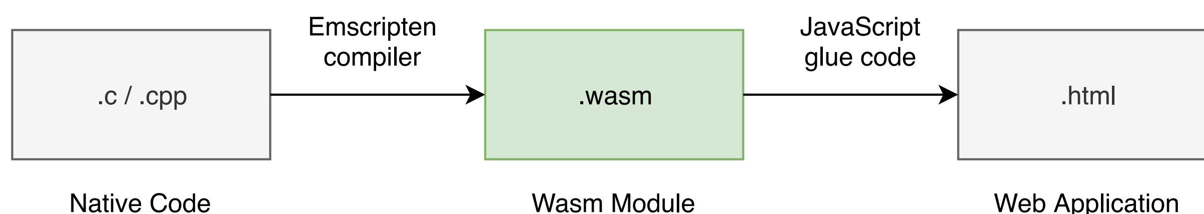

Figure 2.1: WebAssembly compilation process.

Although WebAssembly has many advantages in terms of speed and execution, JavaScript is still necessary to create the connection between the *wasm module* and the browser. The two technologies can communicate freely which lets us get the best out of both worlds, JavaScript has a vast library ecosystem and friendly syntax while WebAssembly delivers near native performance.

Due to its faster performance, WebAssembly is commonly leveraged to perform compute intensive browser workloads that would otherwise be impossible to accomplish. Since browser and cloud featured applications are on the rise, several native applications are making their transition into the browser [10] to benefit from its mobility and user convenience. Recent use case applications go from simple media decoders to more complex gaming experiences and even 3D modelling applications like AutoCad.

2.2 Chrome Browser Architecture

Modern browsers, like operating systems, are robust and place each application (renderers, plugins, extensions, etc.) in a separate process that is walled off the concurrent processes. This architecture ensures that a crash in one component will not disturb the others nor the integrity of the underlying system. It also ensures that each user's access to other user's data is restricted. Essentially, this internal structure brings into the browser the benefits that memory protection and access control have brought to operating systems [13].

To accomplish the desired architecture, Chrome is mainly separated into two different kinds of processes: a privileged *browser process* that is in charge of running the user interface and managing all the other processes, and non-privileged *renderer processes* that are responsible for actually interpreting and laying out the HTML pages. The full architecture diagram is as shown in Figure 2.2. To communicate with each other, browser and renderer processes use Chrome's Inter-Process Communication system (IPC) [14], which in Linux and Android is mainly implemented using asynchronous socket pairs. On the browser process side, communication with the renderers is done via a separate I/O thread (input/output) to ensure that the main thread is never blocked. This is important when making resource requests, for example, this way the main thread can continue to run while waiting for the request to succeed. On the renderer process side, communication is done in the main thread while the remaining work, like decoding media and interpreting JavaScript, is done in a separate thread.

2.2.1 Process models

To understand how renderer processes are created, we first need to understand how browser tabs are managed between different renderers. To this affect, Chrome supports four different models that dictate how the browser allocates web pages into renderer processes [15]. With Figure 2.2 as reference, we can see that the first renderer has two web pages – RenderViews – and that the second renderer has only one. By default, Chrome uses a separate render process for each instance of web site visited. However, users can specify which model to use at startup.

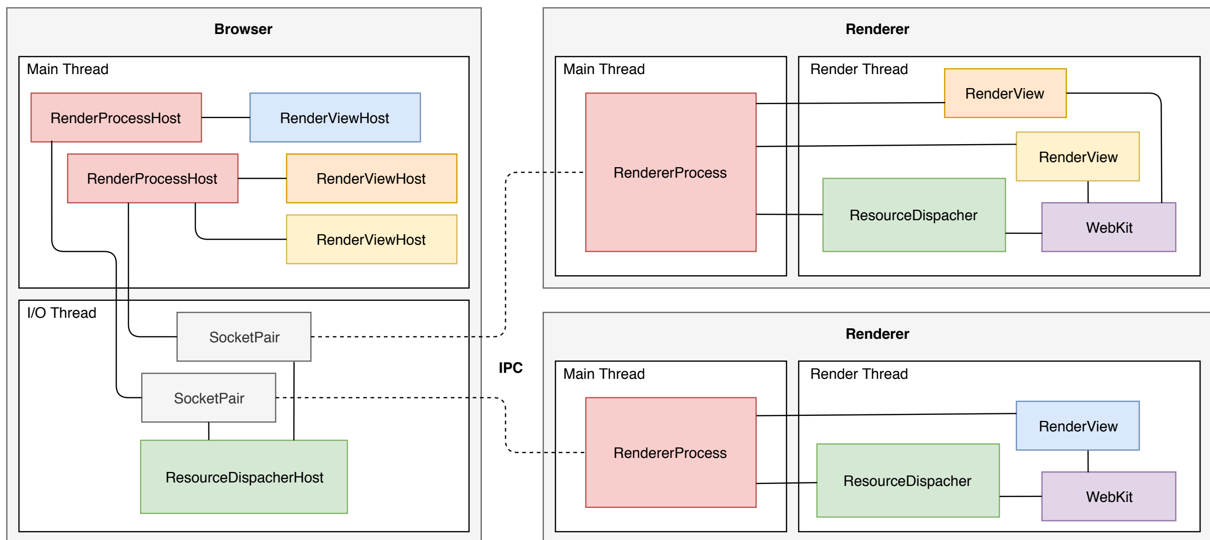

Figure 2.2: Chrome multi-process architecture.

Process per site instance: The default model ensures that pages from different sites are rendered inside separate processes, and separate visits to the same site are also walled off from each other. A site instance is a collection of connected pages from the same site. Two pages are considered connected if they can obtain references to each other, e.g. if we login at ist-fenix and open several course pages they will all share the same renderer, but if we manually open a new tab and start a new fenix session a new process will be created. This model has the advantages of isolating content from different sites while also isolating independent tabs from showing the same web site. On the other hand, it produces more memory overhead, due to the fact that it will generate several renderer processes.

Process per site: This model is similar to the previous one but will group all instances of the same site in the same renderer process. This will result in less memory overhead due to less processes being created, but will generate very large renderer processes, which can be a burden if one tab crashes.

Process per tab: As the name suggests, in this case one renderer process is created for each different browser tab. While being simple to understand, and therefore used in most of public educational presentations, it leads to undesirable context and information sharing between web pages.

Single Process: Finally, and only for the purpose of testing and development, a single process is also supported. In this model, both the browser and renderer engine share the same and only process. It is safe to assume that this is not a secure nor reliable environment for regular usage.

2.2.2 Sandboxing the renderer

To provide a more robust layer of security and to make use of Chrome's multi-process architecture, each renderer process is enclosed in a separate sandbox environment. This approach is essential to the

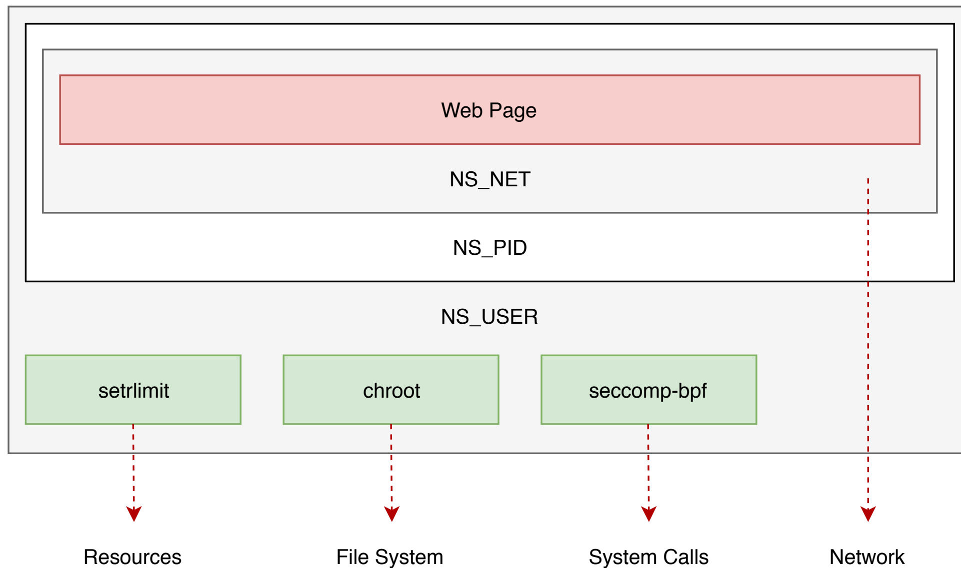

Figure 2.3: Renderer process isolation

security of the browser since the renderers process untrusted input and therefore face the risk of being compromised.

To fully isolate the process, the sandboxing environment makes sure to restrict its access to system resources, to the network and to the file system using the host OS built in permissions and a layered approach focused on semantics and attack surface reduction [4]. In this approach, a semantics layer is in charge of preventing access to most of the resources from a process where it is engaged using *user namespaces*. An attack surface reduction layer restricts access from a process to the attack surface of the kernel, making use of a native *seccomp-bpf* sandbox that both the kernels of Linux and Android offer. This layered approach is necessary due to the fact that it is hard to understand semantics when filtering at the system calls interface. There needs to be guarantees that different renderer processes can be distinguished and are unable to affect the integrity of each other while running under different *seccomp-bpf* sandboxes. If this separation was not made, sandbox escapes would be easy to accomplish.

Digging a bit further, in the semantics layer, three main namespaces are used to isolate the renderer process (see Figure 2.3). The first namespace – *NS_NET* – isolates the process from the network, although the renderer process has to display web pages, all resource requests are made through the parent Browser process, which is the only privileged process, and consequently the only one with internet access. The second namespace – *NS_PID* – isolates the process from the knowledge of existence of other processes by giving it *pid=1*. This makes the process think that it is alone on the machine. The third, and last, namespace – *NS_USER* – makes the process think it is root, in fact, the process is root inside the isolated sandbox environment. Inside this last namespace, three other tools are used to isolate the process further more. *Setrlimit* and *Chroot*, two Linux security functions, are used to limit the resources available to the process and to restrict file system access respectively, and finally the second layer is introduced.

The attack surface reduction layer, as stated before, leverages on an existent Linux kernel sandbox environment, *seccomp-bpf*, to manage and restrict the system calls issued by the renderer process.

Seccomp-bpf was designed to shelter the kernel from malicious code executed in userland. A BPF compiler will compile a process-specific program to filter system calls and send it to the kernel which will interpret this program for each syscall made and allow or disallow the call.

2.2.3 Inter-process Communication

Currently, Chrome uses Mojo and Mojo services in order to handle all the inter-process communications. Given Chrome's multi-process architecture, as portrayed in Figure 2.2, mojo creates message pipes between two endpoints, e.g. the renderer's Main thread and the browser's I/O thread. Each endpoint has a queue of incoming messages and can add new messages to the other endpoint's queue, which makes it a bidirectional message pipe.

Mojo interfaces are described by *mojom* files, which hold a collection of accepted message types that are available to circulate through specific Chrome processes. In order to establish communication, one endpoint must take initial action and become the *Remote* end of the communication which makes it able to send interface messages. The other endpoint, the *Receiver*, is able to receive said messages. The communication remains bidirectional due to the fact that *receivers* can reply to incoming messages. *Mojom* gives the ability to exchange simple messages through Chrome processes but, in order to create more advanced actions, *services* can be used.

A *service* is a self-contained library of code which implements a more complex action or behavior on the receiving end and its interaction with Chrome processes or outside code is done exclusively through mojo interface connections.

For example, when a renderer process needs to know the battery level of the device (information not directly accessible by the process) the renderer needs to communicate with the parent browser process via mojo message pipe. On the browser side, a mojom service can be used to get the information from the underlying operating system.

2.2.4 V8

We have talked about Chrome's architecture and how each component can communicate with each other but we have yet to explain where WebAssembly is actually loaded and executed.

The V8 engine is the current Chrome's open-source engine for high-performance JavaScript and WebAssembly execution. Inside each renderer process, a new and isolated instantiation of V8 is allocated in order to manage, load and execute all the JavaScript and WebAssembly code local to the renderer. V8 is a standalone engine with the ability to instantiate several tiny virtual machines were web page scripts are interpreted and executed safely and isolated from other renderer components, e.g. scripts from other web pages.

In figure 2.2, V8 is represented by the *webkit* component in each renderer process and, as we can see, all web pages from the same renderer share the same V8 instance.

When new JavaScript code is called by the browser, V8 instantiates a new virtual machine to handle the request. After the V8 environment initialization, a new *isolate* is created and made the current one.

Isolates are small memory zones that work as containers to separate and isolate code when several scripts are running inside the same V8 virtual machine. Each *isolate* has its own stack and context where JavaScript and WebAssembly code is compiled and executed. After the conclusion of the page's script the *isolate* is dismissed and, if no more isolates currently exist, the renderer's V8 virtual machine is teared down.

2.3 The *Bao* hypervisor

Currently being developed by our colleagues at Universidade do Minho, the *Bao* hypervisor, a lightweight static partitioning tool, is a key component of our proposed architecture, being in charge of managing and creating secure memory zones outside the privileged scope of the Android OS [16]. This lightweight component, will leverage on virtual memory and a TrustZone based approach to create and manage enclaves, in our case, guest zones to allocate *web cages* where the main OS holds no privileges. *Bao* controls the booting process of the Android OS and constantly monitors and manages its behavior, e.g. system calls performed and physical resources used.

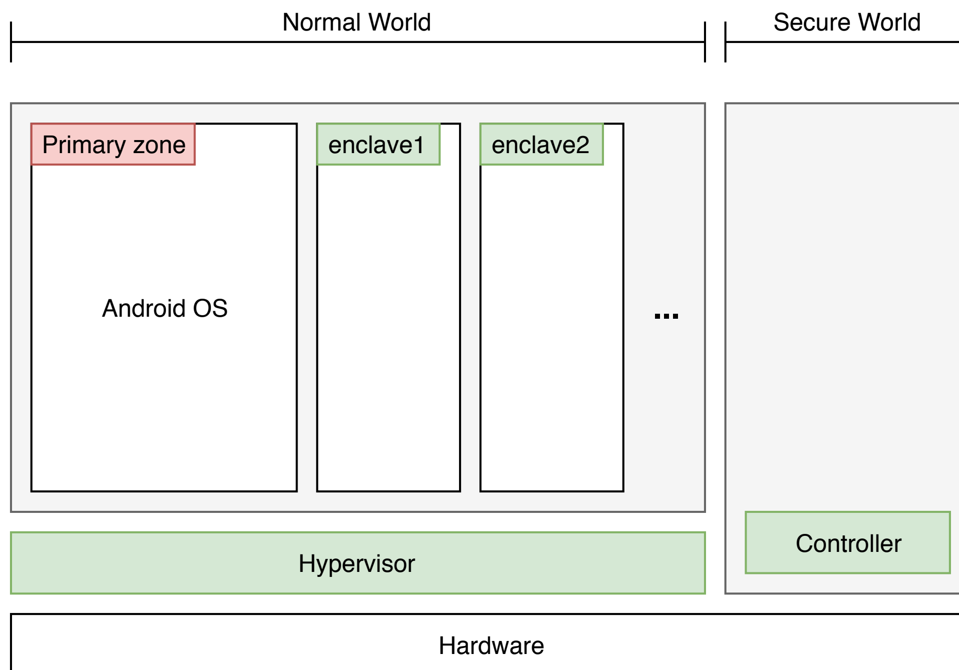

Figure 2.4: *Bao* hypervisor architecture.

The base architecture of *Bao* is shown in Figure 2.4. The primary zone allocates the Android OS and has the limited ability to instantiate other enclaves. In the *secure world*, a *controller* module is also used to execute a *trusted boot* and to calculate and store the hash of all the new enclaves instantiated in order to be able to verify the integrity of the same if later requested, i.e. external parties (e.g. web servers) can remotely attest to the running conditions of the enclaves. Since the enclaves are stateless, and therefore do not have persistence storage capabilities, the *controller* is also used to securely *seal* all the necessary information in order to be able to store it at the primary zone. The data is sealed and

configured to the enclave software and hardware combination so that only enclaves are able to access that information.

Bao is also in charge of creating and managing a shared memory zone, where both the Android OS and the enclaves can access, to provide a communication channel for both ends to exchange the necessary information and data needed.

Summary

In this chapter we introduced the main technologies and some necessary background knowledge on our work. We looked into WebAssembly, the *Bao* hypervisor and dissected the Chrome browser to understand how they are structured and work together. In the next chapter we'll be reviewing some of the related work studied.

Chapter 3

Related Work

Our goal is to build a security mechanism for the Chrome browser that can protect the execution state of WebAssembly code from a potentially compromised operating system. This chapter presents the related work, focusing first on existing mechanisms that provide protection against untrusted applications, and then on known mechanisms that can be used for protecting trusted applications from the environment. This second line of work is mostly related to ours.

3.1 Protection from Untrusted Applications

Unfortunately, when a new technology is introduced and supported by all major modern browsers, malicious actors will try (and have tried) to find ways to use it to their advantage. WebAssembly-based attacks were first introduced in 2017 involving in-browser coin mining [17]. The attackers would take control over the victim's computer by making them visit a malicious web site that would use *wasm code* to leverage on the local machine CPU to mine e-coins. Nowadays, other exploits have surfaced that rely on WebAssembly to compromise the victim's device [18–21]. Lonkar and Chandrayan [22] performed an in-depth study on several real use cases where WebAssembly was used maliciously, including tech support scams, browser exploits and script-based keyloggers. Next, we describe some relevant approaches proposed in the literature to secure the system against untrusted Javascript / WebAssembly code executed in the browser.

3.1.1 Taint tracking of JavaScript and WebAssembly code

Although security is one of the major requirements for local code execution, sensitive information flow is still at a young development age regarding WebAssembly. To help solve this problem, Szanto, Tamm and Pagnoni [23] developed the first JavaScript based virtual machine for *wasm* execution. Taint Tracking is a technique that marks all program variables that have contact with sensitive user information, e.g. banking card information, user personal identification, and hardware system properties, and follows them though the entire execution of the program in order to detect and prevent illegal access, modification, and possible leakage of said data. The developed JavaScript based virtual machine [23] loads

individual *wasm modules* and follows their execution while performing a taint tracking analysis. While *wasm* execution analysis is not one of *WArdian* goals, *wasm* secure execution is. In order to create our isolated *web cages*, the work of Szanto et la. can serve as a valuable foundation to build our desired sandboxing environments.

3.1.2 Same Origin Policy and Site Isolation

All modern web browsers, including Chrome, implement *same origin policy* as a security measure to protect websites information from being accessed by different websites. This approach allows scripts in the web page to access or request data from other web pages but only if both share the same origin. An origin can be defined as a combination of URI scheme, host name and port used. With this policy, a malicious script in one web page will not be able to access sensitive data from another web page through it's Document Object Model (DOM).

Oftentimes, skilled adversaries can find vulnerabilities in the code that enforces same origin policy and try to bypass it in order to steal from other websites. *Site isolation*, a recent security feature in Chrome, makes it harder for malicious websites to have access to information outside of their scope [24]. This new approach fully isolates each website, based on its source, in their own renderer process, i.e. in their own sandbox. We have studied how Chrome allocates renderer processes for different web site instances in Section 2.2.1. Site isolation adds an additional layer of security to that previous implementation. Without site isolation, a malicious web page with an iframe with login buttons for Facebook would have access to user credentials inside the same renderer, this is, the same process would be managing a vulnerable website and user Facebook credentials. With site isolation, a new renderer process is instantiated for each new website involved in the web page, while showing the same visual information to the user. This is done by creating a *dummy* iframe in each renderer process that points to a different one. Figure 3.1 demonstrates the previous example. Currently, Chrome 79 has site isolation [25] enable by default in all desktop environments and isolates all web sites. On Android, only websites that request user credentials are isolated due to performance restrictions of mobile devices.

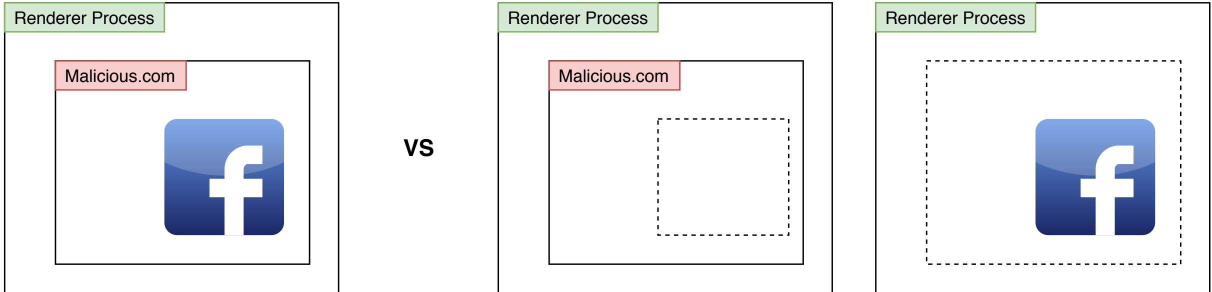

Figure 3.1: Before and after site isolation.

3.1.3 Secure UI

Regarding user interactions with web pages, several security mechanisms can be placed to protect user private input while browsing though vulnerable / malicious web pages. ShadowCrypt [26], a local solution that encrypts / decrypts browser textual data, was for many years the state-of-the-art approach for secure I/O within the browser. This solution was implemented as a browser extension and made use of *Shadow DOM* [27] to isolate and encrypt DOM trees. The extension sits between the user and the browser and only provides encrypted private data to the web application. When encrypted information is retrieved from the web page, the data is locally decrypted and shown to the user.

Freyberger et al. [28], explored the limitations of secure I/O systems in the browser, and propose several attacks that successfully bypass *ShadowCrypt*. The attack vectors expose the privacy weaknesses of *Shadow DOM*, the key browser component of *ShadowCrypt*. Via client-side JavaScript code the authors could trick the *ShadowCrypt* extension into not encrypting private data while displaying encrypted data to the user.

Both *ShadowCrypt* and the techniques proposed to evade it are very interesting to study while developing our security approach. Adding the additional untrusted operating system to these case studies makes for a very similar threat model when comparing it to *WArdian*. In Section 3.2.2 we will discuss in extent another security implementation that takes into consideration user input in web pages and implements several mitigations to similar and current attacks.

3.1.4 Browser hardening

As mentioned in Section 2.2, the Chrome browser – as well as all other modern browsers – implements sandboxing mechanisms for confining untrusted web application code within the boundaries of an unprivileged execution domain. In general, there are several ways to isolate programs and processes from the host operating system, e.g. virtual machines, hardware/software enclaves and so on. A sandbox environment is an isolated space where processes, programs, systems can be executed without disrupting the host machine operating system, i.e. without having full access (or none at all) to the file system, resources and kernel calls. This is possible by virtualizing all the resources that the isolated process needs and make it believe it's running at the real host machine. Shanmugapriya and Geetha [29] performed an in-depth study on sandboxing environments and how to use them to create lightweight secure environments in large scale systems. This techniques are very useful when running untested / untrustworthy code while making sure it cant have any impact on the underlying operating system and even other sandboxing environments.

However, implementing robust sandboxes is a difficult feat. For instance, in the Chrome browser, several techniques have been attempted to escape the sandbox environments (although different levels of expertise are needed for the adversary to be able to bypass the security measures implemented):

- *Forged IPC messages:* The renderer process, properly sandboxed, has connection with the privileged browser process through an IPC system. An attacker that can find and exploit vulnerabilities in the renderer process may be able to bypass some security checks and eventually run arbitrary

code in the sandboxed process. An adversary with this capabilities has access to any data in the renderer process while also able to control the IPC messages sent to the browser. Forged IPC messages can be used to lie to the privileged process and steal information from other renderers [24].

- *Memory disclosure attacks:* This kind of attacker cannot run arbitrary code neither lie to the browser process, but has the ability to read arbitrary data within the renderer process memory address space [24]. To achieve this goal, the adversary relies on speculative execution attacks such as *Spectre* and *Meltdown* [5]. All modern processors perform speculative execution to some extent. This is a way to optimize and improve the performance of some intensive or recurring executions. For example, assuming that a certain condition will be met, the processor executes instructions accordingly and when the condition can be determined true or false the executed instructions will have effect or be discarded respectively. However, while discarded speculative executions do not alter the state of a program, they do make some changes to the lowest level of architectural features of the processors. For example, if by result of a speculative execution a resource is loaded into cache, it may not be removed after the execution is discarded.

To defend against such attacks, most existing approaches involve hardening techniques which consist in the development of customised fixes for the browser [30].

3.1.5 Dynamic monitoring of browser extensions

In Chrome, plugins and browser extensions are also maintained by separate processes, and much like renderer processes, they are fully isolated and sandboxed from the parent browser process, i.e. they have limited privileges. However, since they execute third party code and have several interactions with user data, several web attacks try to exploit vulnerabilities in these processes, such as private information gathering, browsing history retrieval and password theft. Sanchez-Rola et al. [31] presented two main attacks that attempt to bypass browser security measures related to browser extensions, i.e. timing side-channel attacks on access control settings and attacks that take advantage of poor programming practices. Access control settings are security measures implemented by most browsers to mitigate enumeration and exploitation of the installed extensions by a malicious user.

This class of attacks was very relevant when extension processes had full access to the browser process, and therefore had access to system files and resources. Nowadays, with process sandboxing, this types of attacks are less frequent but there still exist some timing side-channel attacks that can be used to exploit them, mainly due to poor programming practices, e.g. a malicious user can make several requests to extensions known to be installed and compare the response times with extensions that are known to not be installed. With this approach, the adversary can eventually enumerate the installed extensions and look for individual vulnerabilities. Combining the information that an adversary can gather with poor programming practices, a skilled attacker can retrieve sensible user information and use it for malicious purposes, e.g. targeted phishing attacks using compromised websites.

Although many possible deviant behaviours by malicious extensions can be mitigated by the latest isolation and sandboxing environments, user tracking at the network level can still be a problem. M. Weissbacher et al. [32] focused on solving this problem and presented a dynamic technique, *Ex-Ray*, for identifying user privacy violations on Chrome's extensions based purely on monitoring network traffic patterns. By presenting unique URLs to several extensions and with the help of a pre-configured honeypot to detect access to said URLs the authors were able to find which extensions were leaking data and tracking user activity. *Ex-Ray* automates this process while creating a useful dataset to train classifiers and autonomously detect leaking and tracking extensions. Similarly to this project, we will be monitoring incoming browser connections to detect WebAssembly modules autonomously.

Chen and Kapravelos [33], also focused on browser extensions related to user privacy, developed a taint analysis framework to perform a large scale study of Chrome's extensions and their privacy practices. Even though *Ex-Ray* [32] used machine learning to identify and analyse network traffic, they remain vulnerable to attackers that possess the ability to mask their network traffic with noise. To solve this issue, the authors present *Mystique* [33], an extension analysis framework that implements dynamic taint tracking for the Chrome browser. In particular, they augmented the V8 JavaScript engine [34] with taint tracking capabilities and automatically load extensions to a monitored environment while analysing them. This work, even with a main objective different then ours, propose several implementations very similar to our project, mainly in regards to monitored secure environments and the modification of Chrome's V8 engine.

3.2 Protection from Untrusted Environments

Above, we focused on browser's security mechanisms that can protect the environment against untrusted applications. In this section, we switch roles and discuss existing techniques that can protect the execution state of trusted web application code from untrusted environments, e.g., attacks launched by an external party sending crafted inputs through an insecure interface, or by an adversary that controls the browser or (worse) the entire operating system which means he has full access to the application's runtime state.

3.2.1 Memory-safe compilers

One class of problems involves the existence of potential vulnerabilities in web applications. Given that C and C++ are potentially problematic when memory related issues are poorly developed, the consequent *wasm module* can suffer from the same problems when compiled from a poorly written piece of native code. Vulnerabilities such as buffer overflows and use-after-free can still disturb the *wasm code* execution. Even though WebAssembly is executed in a sandboxed virtual machine, these classic techniques can still be used to corrupt the memory within the sandbox and mount several attacks, e.g. remote code execution and cross site scripting [35], that may even be able to successfully escape the sandboxed environment.

To prevent these attacks, Vassena et al. [36] developed a study of secure compilers to eliminate memory safety violations based on hypersafety properties, i.e. properties defined over multiple runs of the *wasm module*, that extend the native *wasm* security mechanisms. The authors affirm that, with a secure memory-safety-preserving compiler, memory vulnerabilities are less frequent and therefore sandbox escapes, related with memory issues, are harder to exploit.

3.2.2 Hardware enclaves

A second broad class of problems occurs when the adversary can control the operating system, and from there can naturally access the execution state of web applications in the browser. Hardware enclaves can be very successful against compromised browsers and operating systems. Simply put, they provide user-level execution environments for running sensitive code in isolation from the OS. In commodity hardware, hardware enclaves are supported by Intel's software guard extensions (SGX) [37] technology.

Distributed hardware enclaves: In regards to complex distributed platforms, such as modern data-processing online services, users are forced to trust their private data to the service providers due to their lack of control over the service and the computational platform. Hunt et al. [38] developed a distributed sandbox environment, leveraging hardware enclaves to protect user secret data while it is being processed by untrustworthy services.

Ryoan, the system proposed by the authors, is a remotely attestable sandbox environment that runs on the service machines with an individual instance for each user. With this implementation, users can securely insert sensitive information while verifying that the sandbox environment is running as supposed, which means that they don't have to place trust in the service, developers nor administrators. Each sandbox has a data-processing module that interprets data only once and do not save any state or log of the input to prevent any information leakage (intentional or not) and leverages on trusted hardware enclaves, e.g. based on Intel SGX [37], to provide remote attestation and bypass compromised operating systems.

Our project has similar threat models and trust computing primitives but relies on sandboxing environments to secure local executions instead of remote services executions. Nevertheless, the sandboxing environment is very relatable with our own in terms of privilege separation, trust on the local OS and sandbox environments communication.

Hardware enclaves for browsers: When dealing with untrusted clients, web servers cannot rely on the confidentiality and integrity of client-side JavaScript code and data operated on. For example, a local browser JavaScript credit card validation must be made first locally, to warn the user in case of errors, and validated again when it reaches the web server, since it cannot trust the client. This sort of necessary validations adds time to the operations and waste server resources.

TrustJS [39] explores the execution of client-side JavaScript inside a hardware enclave, e.g. Intel SGX, in order to improve user experience and conserve server resources. The developed framework

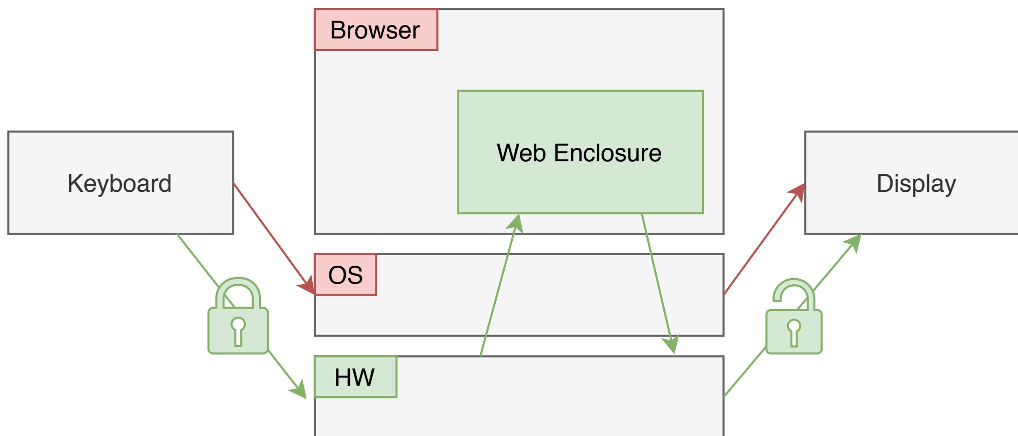

Figure 3.2: Fidelius architecture.

enables trustworthy execution of local JavaScript code that can be attested at the server, instead of validated again.

Fidelius [40], an architecture also based on hardware enclaves, provides user data protection during web browsing sessions. The enclaves are integrated into the browser and enable protection even if the underlying browser and OS are fully controlled by a malicious attacker, a threat model that is very similar to ours. The isolated hardware enclave functions as a small trusted running environment for managing and executing all the web page JavaScript input forms related with user credentials and private actions, e.g. banking transactions. By having included a browser isolated component, which they refer to as *Web Enclosure*, they mitigate the possibility of a compromised browser / OS to interfere with the execution environment.

The main focus of this project is to create secure channels for the user to input data on web pages and also to verify that the visual output shown to the user is valid and correct. To accomplish this tasks, and to protect the user against keyloggers and end-user malware, the authors introduce new protocols for interactions between the enclave, the keyboard and the display by creating secure I/O paths with the help of external raspberry pi's. These external devices will encrypt all keyboard input, only when inserted into a *Web Enclosure*, in a way that only the enclave has the ability to interpret that data. For the user to visualise the input given, the second raspberry pi, connected between the machine and the display, will decrypt and forward the data directly to the display. Figure 3.2 illustrates the system architecture and the secure I/O paths.

This project takes a very interesting approach to protect user private data while browsing the web. Similarly to our work, this project depends on remote attestation and sealed data to verify the code version being run on the hardware enclave and to be able to store persistent data on the untrusted OS respectively. On the other hand, we are focused on controlling WebAssembly executions while maintaining browser performance, which could be severely influenced when encrypting / decrypting large portions of *wasm code* (and the data it manipulates) running on a separate hardware enclave.

3.2.3 Rollback protection

Hardware enclaves, like said SGX [37] enclaves, bring several security advantages, e.g. handle untrusted OS, support attestation and information sealing, but fall short when confronted with side channels and rollback attacks [41]. For instance, consider a banking service and an enclave that stores sealed data on the untrusted OS every time a bank action is performed. In case of crash or reboot, the enclave loses the current state and asks the untrusted OS for the last bank action. Although the untrusted OS can't compromise the integrity of the sealed data, it can send an old commit to the enclave which will make it disregard the previous actions.

To protect against these sort of attacks, Matetic et al. [41] developed *Rote*, a distributed system to provide freshness and integrity to enclave contents and secure storage. All enclave participants send and receive each other sealed data and reply with confirmation of success with a commit id. The authors start with the premise that one single platform is not enough to prevent rollback attacks and finish demonstrating that the only way to break *Rote* is to reset all the participating platforms to their initial state. Similarly to this implementation, *WArdian* also stores sealed data on the untrusted OS, e.g. persistent cookies, but only for performance enhancement which makes rollback attacks not being a critical system requirement. Also, this implementation makes use of a distributed system which steers away from our localized system.

3.2.4 Obfuscated execution

When countering untrusted operating systems, hardware enclaves [42] and systems that offer similar defenses based on hardware-virtualization techniques [43, 44] manage secure memory zones protected from compromised operating systems. In particular, they control the memory pages the application needs to access while remaining protected from the operating system.

However, Xu, Cui and Peinado [6] introduced a controlled side channel attack to take advantage of page access patterns of legacy applications protected with shielding systems. The attacks proposed were successful against famous systems like Overshadow [43], InkTag [44] and Haven [42]. The attack uses page faults as a side channel, it starts by monitoring the addresses of each binary used by the legacy application and revoking access to particular code or data pages. When the application tries to access said pages, page faults will occur which will give opportunity for the malicious operating system to record the accessed pages and analyse their contents afterward. Using this approach, the authors were able to retrieve large amounts of private data from shielding systems and their secured applications. The authors note that several applications could be rewritten to avoid access patterns that depend on application's secret data but that could severely impact performance.

As a response to side channel attacks, obfuscation has been proposed to confuse attackers and therefore prevent memory and data leakage. Rane, Lin and Tiware[45] developed *Raccon*, a set of mechanisms to obfuscate applications at the source code level and create several different and inconsequential paths, which the authors call *decoy paths*, to provide confidentiality to private application data while minimizing execution overhead. This implementation focus only on digital side channels, which the

authors describe as side channels that carry information over discrete bits, e.g. address spaces, cache usage and data size. The authors were able to mitigate several types of side channel attacks, in particular the ones we described above. This implementation, combined with secure data page encryption, can be a viable security mechanism to integrate into *WArdian* to prevent side channel attacks potentially issued by the malicious OS.

3.2.5 TrustZone-assisted TEEs

Trusted Execution Environments (TEEs) are secure integrity-protected environments that provide processing, memory, and secure storage capabilities for the processes and applications running inside. These environments tend to be isolated from the protection environment where the OS runs, which is designated as the *rich execution environment* (REE). Although SGX-based enclaves can be considered an enabling technology for TEE, in mobile computing devices, TEEs are mostly supported by a technology specific to Arm hardware (which is prevalent on mobile platforms). This technology is named ARM TrustZone [46], and it is widely adopted in Android devices as the main isolated zone to run and store sensitive applications and data, e.g. cryptographic keys and certificates [47].

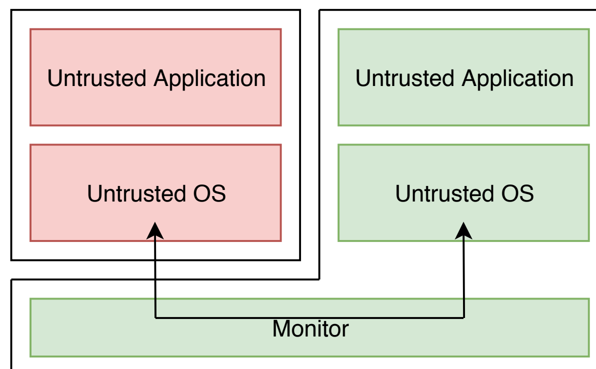

Figure 3.3: ARM TrustZone Architecture for Cortex-A.

The typical architecture of a TrustZone-assisted TEE can be shown in Figure 3.3. TrustZone relies on *secure* (green) and *normal* worlds (red), hardware separated, to securely isolate programs and data. To accomplish this task, it follows a System-on-Chip (SoC) and CPU system-wide approach to security that relies on trusted software developed to run inside the secure world and manage trusted boot, secure world switch monitor, a small trusted OS and all the trusted applications. The combination of all the trusted software creates a *trusted execution environment* that can be expanded for future usage [48]. Unfortunately, due to the widely adoption of the secure world to run and store applications and data, this secure zone is starting to get bloated and security issues might rise from buggy (or even malicious) code / data stored inside [49]. As an attempt to increase user data security, sandboxing environments that do not rely on the secure world to be isolated from the host OS are worth exploring.

To make TrustZone-assisted TEEs more robust against side channels, Costan et al. [50] introduced *Sanctum*. This system provides defenses against known side-channel attacks, such as cache timing

attacks and passive address translation attacks by monitoring the memory access patterns of the enclaves while hiding them from the host OS. To accomplish this task, the authors implemented a software based secure monitor, to create and manage enclaves, with minimal invasive hardware modifications. Since the implementation is not based in a specific hardware piece, like the widely used Intel SGX, the system can be slight modified to correctly function in several different hardware architectures. This sort of implementations is very useful to our own case, *WArdian*, since we intend to create isolated zones without creating noticeable performance drawbacks to the user.

Summary

In this chapter, we reviewed several articles and projects related to WArdian. We started by looking into protections against untrusted applications and followed our study with research about untrusted environments. From several studies, we gathered important information that was kept in mind while developing WArdian. In the next chapter, we present our solution and dive deep into its architecture.

Chapter 4

WArdian Design

This chapter presents *WArdian*, our proposed system for securing WebAssembly code within the Chrome browser against attackers with the ability to control the Android OS. Our solution is targeted to run on Arm platforms featuring ARM TrustZone technology. After providing a motivating example and describing our threat model, we present an overview of the system's components and discuss each one in detail.

4.1 Motivational Example and Threat Model

Before the technical specifications of our solution, we begin with a demonstrative example showing the need to protect the current state of *wasm* web applications. As portrayed in Figure 4.1, consider a bitcoin wallet service that allows its users to manage their account and transactions on a mobile device through the local browser. The local browser, running on the Android OS, has a sandboxed partition where all the JavaScript / WebAssembly code is loaded and executed. When a *wasm module* is requested by the browser, a new sandboxed environment – V8 instance – is created and the application's web server proceeds to send the module requested. In the current state of affairs, an adversary with the ability to control the browser or the OS (e.g., by installing a rootkit), would be able to retrieve sensitive application data, which in this case includes the private key associated with the user's BitcoinWallet. By retrieving this key, it is straightforward for the adversary to transfer all money from the user's account.

This problem can be further generalized to other WebAssembly applications that manipulate data items that must be preserved absolutely private to the user (e.g., account details, user credentials, user information and health records). Protection of this information is a priority for the correct execution of the application. Unfortunately, like for said example, an attacker that can compromise the integrity of the operating system, can potentially have access to the sandboxed information of all these applications. Our work aims to create a separate memory zone to load and execute *wasm modules* while preventing the compromised Android OS from tampering with it. We call these secured WebAssembly packages by the name *cagelets*, and the secure environments where they execute as *web cages*.

We want to protect against an attacker that has full control over the operating system of the mobile device, e.g., the attacker can obtain this level of control by installing a rootkit. The adversary can also

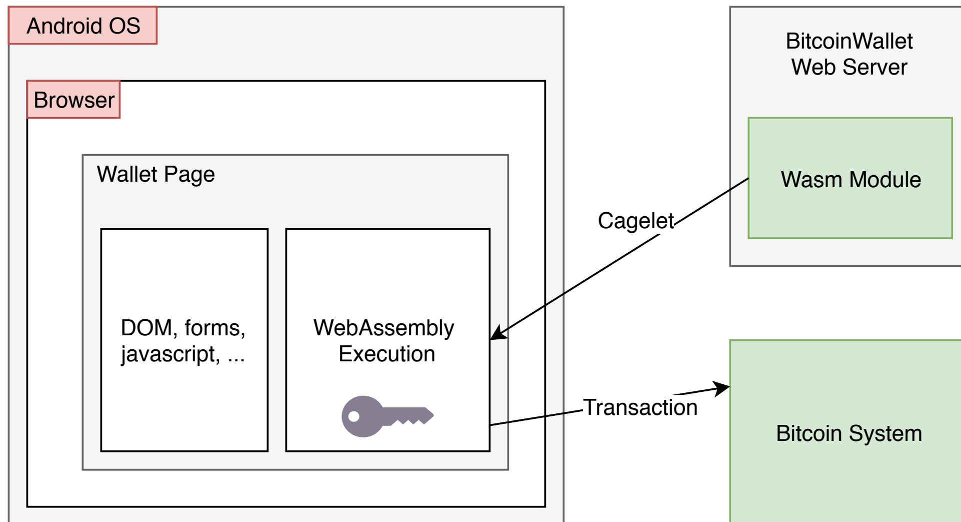

Figure 4.1: Bitcoin Wallet Example.

control the browser, e.g., by exploiting a vulnerability in the browser that allows him to access the execution environment of JavaScript / WebAssembly sandboxes. We assume that our attacker can examine and modify unprotected memory where Web pages are allocated, and that he/she can examine the communication exchanged between the code executed by the Web page and the remote site. We assume, however, that the attacker cannot inspect the contents of secure memory regions allocated to hardware enclaves or to trusted execution environments. It is also worth noticing that the secure memory regions we envision for securing the WebAssembly code are instantiated by the browser process, so we do not consider any type of DOS attacks.

4.2 System Overview

Our solution tackles the previous problem by isolating the WebAssembly execution from the browser. *WArdian* prevents Chrome from loading and running *wasm modules* and instead leverages on the *Bao* hypervisor, and its enclave environment, to safely execute *wasm code*. *Bao* has the ability to create separate memory spaces inside the same physical device – enclaves – where the Android OS has no privileges, and therefore, cannot access the isolated WebAssembly application's data. The native Chrome's sandboxing mechanisms remain fully functional, and continue to operate for JavaScript code, but have no indication that WebAssembly is being called by the web pages running inside it's renderer processes. This is accomplished by the introduction of a browser extension that is in charge of tricking web pages into communicating with *WArdian* instead of Chrome's V8 virtualization environment. Running at the enclave, *WArdian* maintains a fully functional WebAssembly standalone runtime that is able to securely communicate with our browser extension, creating the abstraction needed to extract and inject *wasm code*, and the corresponding outputs, without V8's knowledge.

Figure 4.2 presents the architecture of *WArdian*, our proposed solution for securing WebAssembly code on Android platforms. In order to provide this level of protection from a compromised Android OS,

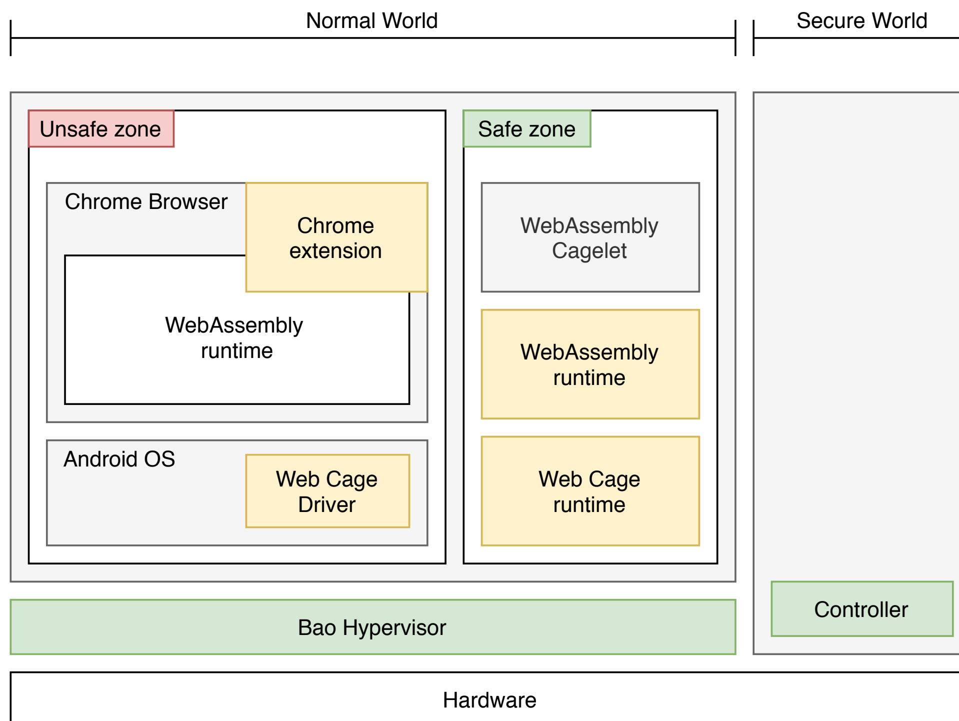

Figure 4.2: *WArdian* architecture: its specific components are colored in yellow.

we leverage on an existing hypervisor – named *Bao* – that is in charge of creating a secure memory zone where the operating system holds no privileges. In the normal world, the Android OS runs as normally expected. The Chrome browser, running on the Android OS, will also have the same behaviour as before, with the exception of renderers that have a *wasm module* loaded. When an IPC message reaches the browser process with a request to download a *wasm module*, a new *web cage* is created inside a new and secure environment where the received *cagelet* is loaded and executed. The underlying mechanisms offered by *Bao* provide memory isolation procedures that prevent an attacker from inspecting the content of a web cage and access the execution state of guest cagelet code.

To illustrate how *WArdian* works in practice, consider a simple web page as portrayed in Figure 4.3. This simple web page application presents a new dice face each time the user clicks on it, i.e. the faces 1 through 6 are randomized and one is shown on screen. A web developer can take advantage of WebAssembly's near native speed and program the randomizer function in C/C++ instead of relying on the browser's JavaScript. Analogous to the previous example, in this mock up application, our goal is to secure the integrity and confidentiality of the randomizer function and respective execution state.

With the *WArdian* extension enabled, the WebAssembly function call is detected and overridden in order to disable Chrome's V8 participation on the request. Instead, the *wasm module* information – cagelet – is sent to the web cage environment, via web cage driver and the *Bao* communication mechanisms, and the code is isolated from the browser at the moment of execution. Finally, the *wasm* output is redirected through the *WArdian* extension and consequently injected back into the browser.

With this new WebAssembly life-cycle provided by *WArdian*, *wasm code* is executed outside the privileged bounds of the Android OS which prevents a malicious operating system from tampering with

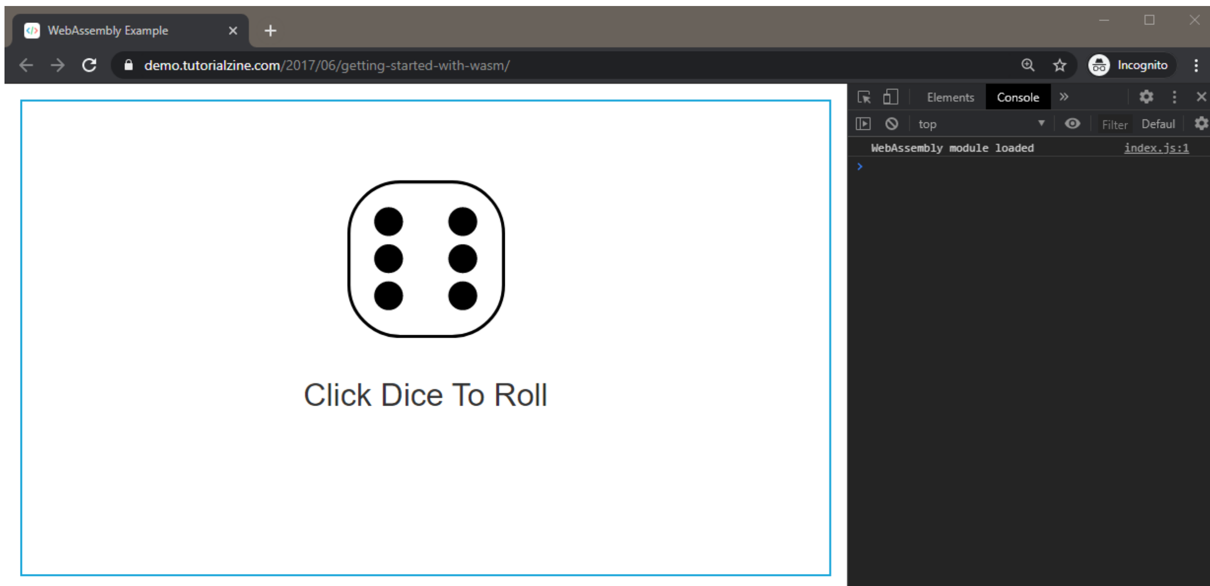

Figure 4.3: Simple *wasm* web page.

the information managed by the sandboxed *wasm* application, e.g. by tampering with the *wasm module* in order to create a weighted dice (one that always shows the same face).

In contrast to renderer processes that interpret and layout HTML pages, the *web cages* only load and execute *cagelets*, which means that most of the renderer process is still executed in the untrusted environment. The creation of these cages is very similar to the previously discussed creation of renderer processes. Using site isolation [24], one *web cage* will be created for each different source, and no communication is allowed between different cages. Regarding the allocation of *Web Cages*, two scenarios can be explored:

- New *web cages* are instantiated inside the same and only secure zone; This implementation facilitates the creation of Cages and utilize the same LibOS, making it have a shorter impact on memory used. However, several *Web Cages* in the same zone may deteriorate the overall security pretended.

- New *web cages* are instantiated in a new and separate secure zone; On the other hand, this implementation improves security but may have performance problems when considering a user with several *cagelets* requested from different sources.

We ended up going with the first approach, since the *Bao* mechanisms work in a similar fashion out of the box, and made our lives easier when developing and iterating through our proof of concept prototypes. However, the second approach can also be implemented, in the future, with some tweaks at the internals of *Bao*.

In regards to usability, users will always have the knowledge of *WArdian* since the extension needs to be installed and operational in the local user's browser. In the current state of *WArdian*, since the extension isn't available at the Google store, users' need to browse to *chrome://extensions*, enable *Developer mode* and load the unpacked .zip *WArdian* extension, as shown in Figure 4.4. Nevertheless,

these are the only necessary steps for *WArdian* to work on the device and, after this initial setup, *WArdian* execution is completely invisible to the user. In the future, we intend to add a user options page to the *WArdian* extension so that users can also chose which *wasm* operations they intend to be executed inside the enclave.

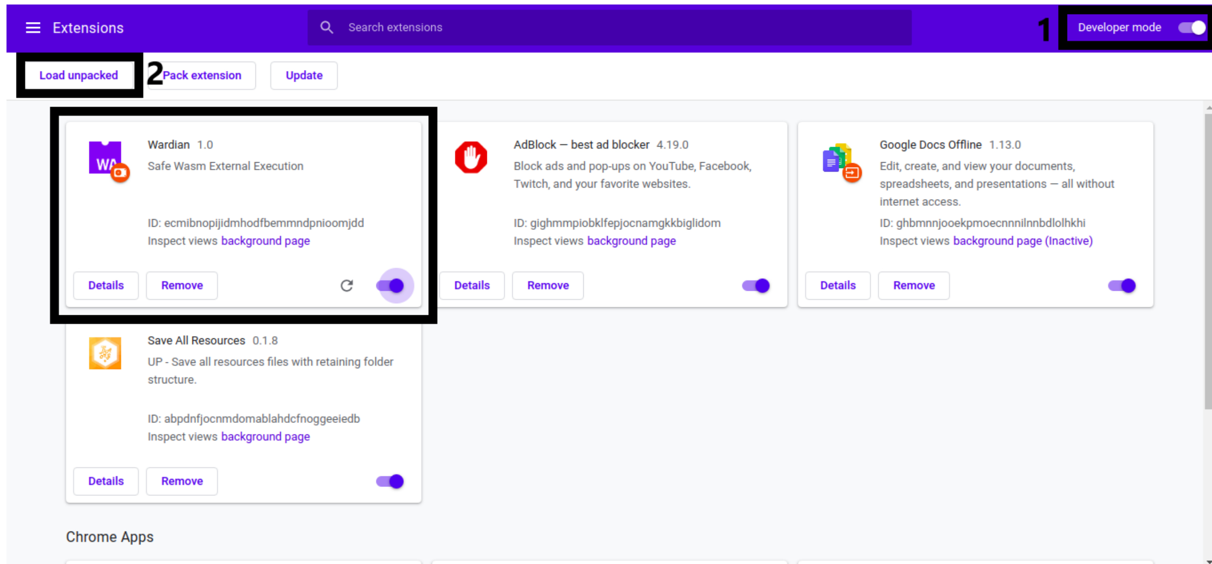

Figure 4.4: Chrome extensions page.

4.3 WArdian API

WArdian is meant for user local data protection by improving the ability of web developers to securely insert *wasm* operations in their web pages. In order for *WArdian* to work correctly, it needs to be able to detect and intercept *wasm* calls made by the current web pages loaded in the browser. To accomplish this task, some HTML and JavaScript syntax restrictions are expected to be followed by web developers. These restrictions demand that the web developer uses a specific WArdian API comprising a few HTML tags and JavaScript function calls.

Specifically, whenever a web developer intends to execute *wasm code* on the browser, and isolate it via *WArdian*, he/she must name the parent script function with a default name, and all the necessary arguments, for *WArdian* to correctly detect the call. This is implemented this way, due to the fact that differently compiled *wasm modules* are called using different methods by the JavaScript code on each page. For example, the same piece of C code can be compiled to *wasm* using different compilers, e.g. emscripten or WASI. Both compilers will originate the same *wasm module* but with different internal structures that rely on different JavaScript instantiation functions. Since more compilers are constantly appearing, with *wasm* security and performance improvements [36], we choose to introduce these syntax restrictions in order to focus on the development of *WArdian* instead of overriding each new JavaScript fetching form.

The previous dice example can be further explored in order to understand these restrictions. Instead of using typical JavaScript to program the dice randomizer function, WebAssembly allows the developer

to use native C/C++ code and run it directly in the browser. For instance, consider the following C file that could be used to accomplish the dice randomizer function:

```c
// DiceRoll.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char ** argv) {
    printf("WebAssembly module loaded\n");
}

int roll_dice() {
    srand ( time(NULL) );
    return rand() % 6 + 1;
}
```

After compiling it to a *wasm module*, the developer has to fetch the resulting *.wasm* file into the web page and call the roll_dice() function via JavaScript.

```html
// index.html
<script>

var module = path_to_wasm_module;

async function WardianfetchAndInstantiate() {
    const response = await fetch(module);
    const buffer = await response.arrayBuffer();
    const obj = await WebAssembly.instantiate(buffer);
    res = obj.instance.exports.roll_dice();
    return res;
}

</script>
```

The function *WardianFetchAndInstantiate* is the one being detected by the *WArdian* extension and overridden. Without *WArdian* enabled in the browser, this JavaScript function would be interpreted and executed inside Chrome's V8 environment but with *WArdian* enabled the entire function's code is overwritten to trigger the enclave environment and send over the *wasm module*. The enclave output is then injected into the return variable, i.e. the variable *res* in this case.

In the future, we intend to detect every different way to load and call *wasm code* and implement

HTML developer tags, so that *WArdian* can correctly run with or without web developer knowledge. This will enable web developers to chose, if they want, which *wasm* functions should run in the enclave and which ones can be executed by the browser, which can be accomplished by the insertion of a custom tag before each script in the web page. This implementation can be useful if, in the same web page, there are sensitive and non-sensitive *wasm* calls that the developer wants to isolate in different forms.

4.4 Extending the Browser

In the unsafe zone, where the Android OS runs, we augmented the Chrome browser with the *WArdian* extension in order to override native JavaScript calls and disable Chrome's V8 environment when *wasm* calls are made. *WArdian* is responsible for dispatching the requests for execution of *wasm modules* to be served by the WebAssembly runtime running inside a *Bao* enclave. Our lightweight Chrome extension, with the added advantage of solution portability, improves browser *wasm* secure execution without disrupting it's native sandboxing mechanisms.

The *WArdian* extension starts by detecting web requests for new *wasm modules* and overrides the JavaScript functions that fetch and instantiate said modules. After this initial setup, the extension runs idle until it listens for a *wasm* function call. When this condition is triggered, our extension disables the web page ability to run the *wasm* function and sends the respective *wasm module*, with all its necessary arguments, to the enclave. Finally, the *wasm module* is safely executed in our isolated memory space, the result is returned to the extension and consequently injected into the respective web page.

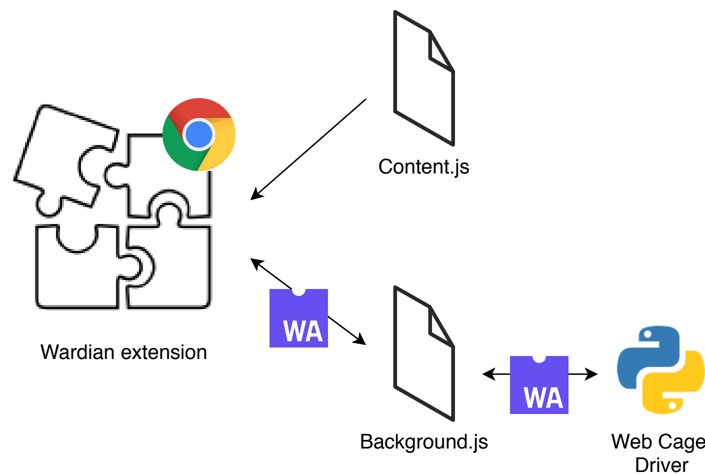

Figure 4.5: *WArdian* extension architecture.

In order to accomplish these tasks, and as portrayed in Figure 4.5, the extension relies on two main components, a background and a content script. The Background script, as the name implies, runs in the background from the moment Chrome starts and is in charge of listening to various events, such as WebAssembly fetch, instantiation and execution requests, and to establish communication with the enclave, via web cage driver.

In order to override some of the necessary JavaScript functions, a content script was introduced. Recalling Chrome multi-process architecture, extension processes are analogous to renderer processes in

which they run in a separate and isolated process, so the background script does not have the ability to directly communicate or alter the web page content. A content script, however, allows the extension to inject JavaScript code in the current web page which will allow it to run in the same context of the renderer process, making it possible to override native JavaScript functions and establish a communication channel between the web page and the background script.

Revisiting the dice example from Section 4.2, we can now understand how the extension internals work. The content script, when the web page begins to load, and since it runs in the same context as the current renderer, has the ability to parse the web page contents for *wasm* calls and override the JavaScript calls that implement them. At the same time, the background script is already running and listening for *wasm* events. When the user clicks on the dice, the *wasm* function doesn't run locally since the content script already overridden the necessary JavaScript functions. Instead, the background script sends the *wasm module* over to the web cage environment, at the enclave, where the module is executed and the corresponding output sent back to the background script. Finally, and after being notified by the background script with the *wasm* response, the content script can then inject the new dice value into the return variable of the initial web page's JavaScript function that initiated the entire process.

4.5 Securing Wasm Execution

With the *WArdian* components at the Android OS explained, we now introduce the enclave environment where the *wasm modules* are loaded and executed isolated from the privileged scope of the device's operating system inside an enclave-like abstraction that we refer to as Web Cage environment. To create our desired enclave architecture, two main components are needed, a Web Cage runtime, in charge of managing all system calls at the enclave space, which we previously refereed to as LibOs for simplicity reasons, and a WebAssembly runtime where *wasm modules* can be loaded and safely executed.

For implementing the Web Cage runtime, several open source projects were researched and analyzed but since we have a very specific hardware architecture and overall functionality in mind, our choices were very filtered at the starting point already. In particular, our Web Cage runtime needs to support ARM architectures and be as lightweight as possible, in order to minimize the enclave space. A good candidate that satisfies this requirement is the Zephyr RTOS (real time operating system), which is an open source project by the Linux Foundation. Zephyr is a scalable lightweight OS optimized for resource constrained devices and built with security in mind. It presents built in support for several different hardware architectures, including ARM, which checks all *WArdian* requirements. Zephyr is in charge of managing all the systems calls at the enclave, i.e. it works as a replacement for all the necessary Android OS functions that *WArdian* needs while running at the enclave. Being that Zephyr is mainly directed at small embedded systems, it is built with a small footprint which helps us maintain a compact enclave space.

In order for our enclave to maintain the desired small size factor, the choice of WebAssembly runtime was also very constrained. Several different solutions were studied, and then again, our specific hardware architecture and small footprint were the decisive factors. We opted to adopt the WebAssembly

Micro Runtime, i.e. WAMR, a Bytecode Alliance open source project, since it presented all *WArdian* requirements and the added bonus of built in support for the Zephyr OS. WAMR is a lightweight standalone WebAssembly runtime that makes use of it's custom VM core – *iwasm* – to interpret and compile *wasm code*. *Iwasm* supports ahead of time (AoT) and just-in-time compilation (JIT), useful for obtaining near native application speed, and its minimal binary size makes it easy to embed in all sorts of environments while also providing low memory usage.

The combination of Zephyr and WAMR makes for a very small enclave environment but with all the necessary security mechanisms and basic functionality in place that enables *WArdian* to successfully accomplish it's proposed goal. By embedding WAMR in Zephyr, and running this system configuration at our enclave space, we accomplish the desired web cage environment that is able to communicate with our *WArdian* browser extension and safely sandbox *wasm code* executions.

Recalling the previous dice example, we can now understand how each of these components work together in order to complete the *WArdian* main system architecture. When a new message is received from the browser extension, including the desired *wasm module* to execute and all it's necessary parameters, the embedded WAMR application in Zephyr collects this information and executes the desired module. After allocating memory for the new module, WAMR creates a new thread with a new instantiation of a WebAssembly runtime environment that loads and executes the received dice *wasm* randomizer function. The output of this application is then sent to the browser extension and the WebAssembly runtime environment destroyed, i.e. the allocated memory is cleaned and the thread terminated. By creating a new thread and memory allocation, inside the enclave, for each *wasm module*, we can guarantee no communication between different web cages and after terminating each execution, the destruction of the WebAssembly runtime deletes all the *wasm* used data from the enclave environment.

4.6 Communication Channels

With both the components of *WArdian* running in the Android's zone and those in the enclave explained, we now focus on the communication between both zones and the mechanisms used to accomplish it. Figure 4.6 details the entire process. In order for both endpoints to communicate with each other, a shared memory zone is implemented by two underlying components: the web cage driver deployed in the Android's memory space, and the web cage runtime residing in the *Bao* enclave. The former provides an interface to the *WArdian* extension.

In the untrusted Android OS, the web cage driver manage all the in and outbound communications between both zones. The hypervisor also manages these communications making sure to preserve the integrity of the secure environment. The web cage runtime, based on a *LibOS* (a component – Zephyr – that simulates an OS kernel and emulates system calls requested by the *web cage*), provides an interface to the WebAssembly runtime and implements some necessary system functions.

As depicted in Figure 4.6, all the communication steps are numbered in ascending order. Relying on our previous dice example we can more easily understand what kind of information is being exchanged:

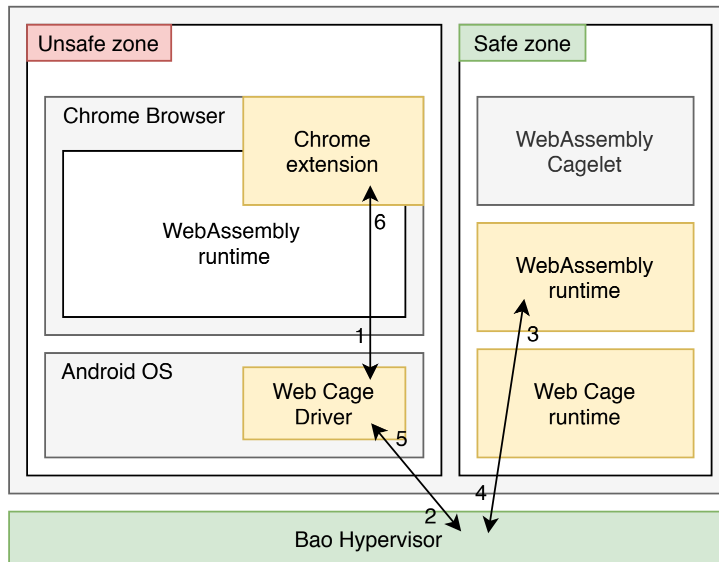

Figure 4.6: *WArdian* communication channels.

1. *WArdian* extension sends the dice *wasm module* followed by all it's necessary parameters (function to be executed, function arguments, return object type, etc) serialized using JSON format.

2. Web Cage Driver dumps the received information into the *Bao* shared memory buffer.

3. WAMR application, embedded in Zephyr, receives the *wasm* contents and is able to load and execute the dice randomizer function.

4. WebAssembly runtime output is redirected to the Web Cage Driver via the *Bao* shared memory mechanisms.

5. Web Cage Driver receives the *wasm* output, serializes it to JSON and replies the value to the *WArdian* Extension

6. *WArdian* extension receives the final result and is able to inject it into the JavaScript function that requested it.

4.6.1 Native Messaging

To establish a communication channel between the extension and the enclave, our Web Cage driver needs to be able to exchange messages with the *Wardian* extension in real time. This driver, implemented using the python programming language, for simplicity reasons, works as a communication bridge between the *WArdian* extension and the enclave environment.

Chrome allows developers to exchange messages between extensions and native applications using their Native Messaging API. Chrome starts each native messaging host, in this case our web cage driver, in a separate process and communicates with it using standard input (stdin) and standard output (stdout). The same format is used to send messages in both directions, as depicted in steps 1 and 6 of

Figure 4.6: each message is serialized using JSON, UTF-8 encoded, and preceded with 32-bit message length in native byte order, for memory allocating purposes. The maximum size of a single message from the native messaging host is 1 MB, mainly to protect Chrome from misbehaving native applications.

Our web cage driver communicates with the extension background script, via native messaging, and is in charge of receiving the *wasm module* to be executed in the enclave and their respective parameters, redirecting this information to the enclave and finally redirecting the enclave output back to the background script.

4.6.2 Shared Memory

The *WArdian* Web Cage environment, running in the enclave, is detected by *Bao* as a *bao guest* and has a *Bao* interface which allows it to communicate with the Android zone via shared memory. On the Android OS, *Bao* instantiates a high level operating system component which is an enclave application, running in the Android's memory space, that implements an interface with all the enclave available services so that they can be requested by the *WArdian* components running in the Android zone.

WArdian introduces two new services to the *Bao* communication mechanisms that can be called by our web cage driver: one to start the enclave environment, i.e. WAMR application execution, and one to send *wasm* data to the running web cage. The first one is called as soon as the Chrome browser starts (event triggered by the *WArdian* extension) and the second one is called each time a *wasm* function is requested by the browser. The communication process is based on said services. *WArdian* components at the Android zone, i.e. web cage driver, can request enclave services, that are interpreted by *Bao* and executed in the safe enclave environment. The shared memory works like a buffer between both ends.

To minimize *Bao* space occupation on the mobile device, since they can be very limited in terms of physical memory, the shared memory buffer is initialized with 8 Kb of allocated memory. This small buffer is, most of the times, enough for all the commands and parameters that need to be passed between both ends but in case of need to pass large amounts of data, as can be the case when working with big *wasm modules*, a *chunked* approach is used.

The *Bao* hypervisor implements a cpu interruption – *bao_hvc* – to notify each end when a new message is available in shared memory. When the need to exchange data is larger than the maximum shared buffer size, the message is split in chunks of 8Kb and a first message is sent with the total number of chunks of said message. Both ends can then coordinate and gather all message chunks, when a big message is sent from one end to the other, and construct the final message at the receiving end.

4.7 Secure Bootstrap

To prevent an adversary from disabling *WArdian*'s security mechanisms, it is essential to guarantee the integrity of the entire system's trusted computing base (TCB) upon boot. This TCB includes components in the secure world – the controller – and in the normal world – the *Bao* hypervisor and *WArdian*'s web cage components (see Figure 4.2). When the mobile device starts its bootstrapping process, the

controller firmware is verified (according to a typical trusted boot digital signature validation). Then, after validating the digital signatures of *WArdian*'s software images to be executed in the normal world, the controller switches to the normal world and hands over the control flow to *Bao*'s bootloader. This chain of events ensures that the integrity of *WArdian*'s TCB has not been compromised upon boot.

The remaining of the bootstrapping sequence is as follows. *Bao* starts by allocating space for three different memory regions (see Figure 4.2). One for the Android OS (unsafe zone), one for the *WArdian* enclave (safe zone), and a third memory region where both previous regions have access (managed by the hypervisor). Depending on the mobile device capability, a different processing power is allocated for the enclave environment, by default, only one processor core is allocated for the enclave. At the current state of *Bao*, one enclave needs an entire core to work correctly, which means that in order to have more than one web cage available concurrently, several of the device's CPU cores are utilized. This can create a possible bottleneck on the device's CPU and therefore *WArdian* is currently working with only one web cage at a time. Once all three memory zones are allocated, *Bao* starts the Android booting process in the unsafe zone and the enclave environment in the safe zone. The third memory region is intended for the communication process explained in the previous section.

Although all these memory regions have been successfully allocated and the enclave environment set, the *WArdian* enclave will not be ready until a new Chrome session starts, i.e. when a renderer process is initialized. When this event occurs, our Web Cage driver starts the WAMR application embedded in our enclave environment, utilizing the *Bao* service explained in the previous section, and only then the entire *WArdian* bootstrap is completed. When Chrome session ends and the main process is terminated, the *WArdian* extension notifies the Web Cage driver to tear down the WAMR application but the zephyr environment stays up to engage in future browsing sessions. We opted for this approach since we want our WAMR application ready as fast as possible for *wasm* executions while not wasting system resources when the browser is closed.

Summary

In this chapter we presented *WArdian*, our solution to isolate WebAssembly executions from untrusted mobile devices. With the help of several practical examples, we went over its utilization models, system's architecture and individually explained each component and their consequent functionality. In the next chapter we'll present our work methodology and explicitly show how we were able to implement and develop our proposed solution.

Chapter 5

Implementation

During the development of *WArdian*, several prototypes were built while trying to accomplish our desired final goal. This chapter presents all the major steps taken and our reasoning for them to happen. We also present some of the setbacks we faced and how we were able to solve them.

5.1 WArdian Prototypes

Since *WArdian* has a complex architecture where multiple components are integrated and work together in different ways, we have opted, since the beginning of the development, for a iterative and incremental approach. To follow this development decision, we have built several *WArdian* prototypes and learned from each one how to progress and optimize the next ones.

Two main prototypes were essential for the final *WArdian build*. We have started with QEMU, an open source machine emulator and virtualizer, to simulate our enclave environment and used a generic x86_64 Linux distribution – Ubuntu – as our main operating system (where Chrome runs). This first prototype does not yet run in the desired hardware architecture nor relies on the *Bao* hypervisor but implements all the functionality and communication channels desired. After the QEMU prototype, with all the components of the communication dealt with, we iterated to the second prototyping stage where we started to explore our desired hardware architecture – ARM – and introduced the *Bao* hypervisor in order to create our final *WArdian* architecture. Next, we describe both these prototypes.

5.1.1 QEMU Prototype

To simulate, as close as possible, our final architecture, we developed an initial prototype leveraging on QEMU and its built-in support to emulate Cortex-A53, the one used by most of the modern mobile devices, and used a stable version of Ubuntu, a Linux distribution, to simulate the host Android OS.

With all the previously discussed components and the addition of QEMU to this prototype, an initial architecture takes place, as shown in Figure 5.1. Inside our *qemu-cortex-a53* emulation build, the WAMR application executes the given *wasm module* with the arguments sent by the web cage driver. They both communicate via stdin / stdout, which makes it possible for the driver to get the final web cage execution

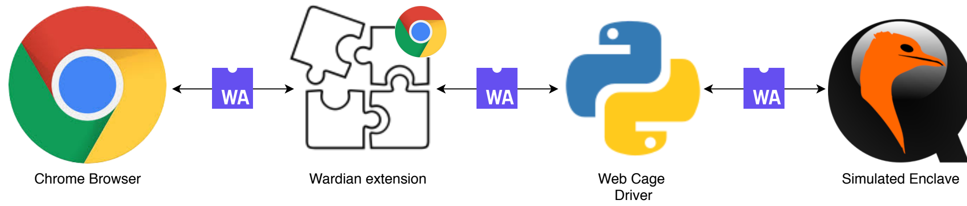

Figure 5.1: QEMU prototype architecture.

result from WAMR and consequently report it back to the extension's background process. All the remaining components work as explained in Chapter 4. To aid in the development and testing of this first prototype, a minimal web page was created, as shown is Figure 5.2.

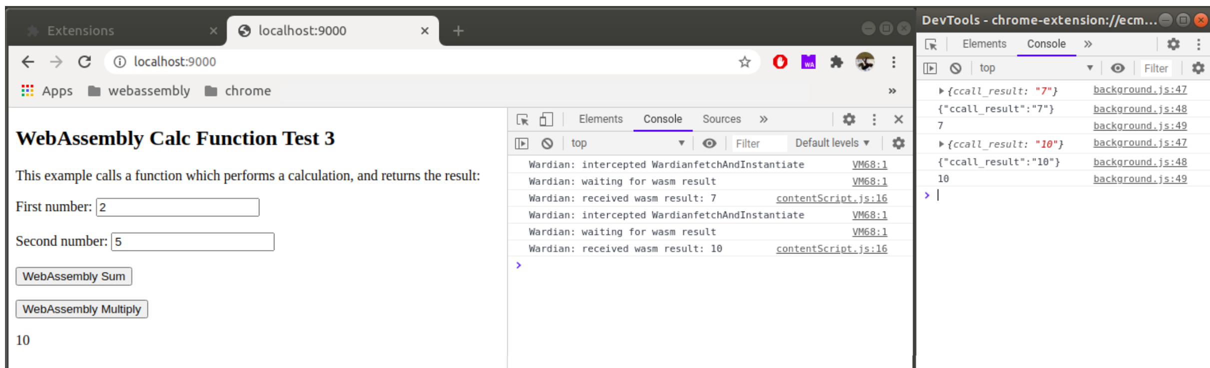

Figure 5.2: *Wasm* simple testing web page.

On the web page source folder, two *wasm modules* were introduced with one C function each (addition and multiplication basic functions). This simple testing page allows the user to insert two integers and chose a *wasm* function to execute. With the help of the native browser console and Chrome's DevTools we were able to monitor the *WArdian* extension behaviour and check if each extension stage was completed with success.

With this initial prototype we were able to achieve the desired functionality, but not yet all the security mechanisms wanted since *Bao* was not yet integrated in the system. However, some benchmark values could already be retrieved.

5.1.2 Rock960 Prototype

With the QEMU prototypes functional, it was time to introduce the *Bao* hypervisor and to run *WArdian* in the desired hardware architecture. To accomplish this task, we leveraged on Rock960, a circuit board based on RK3399 SoC – System-on-a-chip – which is a dual cortex CPU that includes Cortex-A53, the one we intend to use. The used prototype board is as shown in Figure 5.3 alongside a smaller circuit board which was wired to the first one in order to have access to a serial debug console.

Currently, the latest Chrome build for Android does not support the use of browser extensions, but since Chromium is an open source project, several builds already exist with extensions enabled. Fortunately, the Kiwi browser is one of them, and we chose it to test and debug *WArdian* since it was the one

(a) Rock960

(b) CP2102N-MINIEK

Figure 5.3: *WArdian* prototyping boards.

with less changes made to the Chrome source code. We can safely consider Kiwi to be a build of the current Chrome browser with extensions enabled on Android.

Since we already had a QEMU prototype running, the only steps missing were the introduction of the Kiwi browser and the compilation of our enclave environment to the correct hardware setup. With those changes made and the addition of the *Bao* hypervisor, to handle all the component's and device's booting and the communication protocol, we could finally build our minimal viable product prototype, as shown in Figure 5.4. The next sections explain in detail how we were able to achieve these prototypes successfully, all the major setbacks we were faced with and how we were able to overcome them.

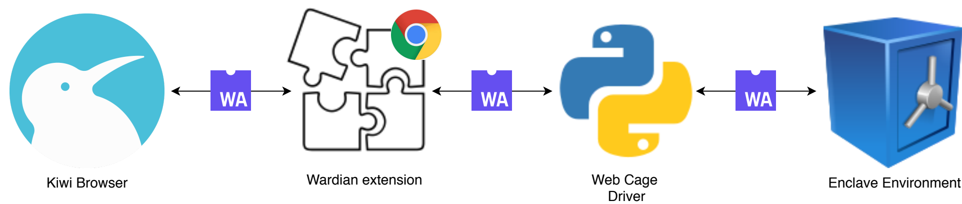

Figure 5.4: Rock960 prototype architecture.

5.2 Chrome Modifications

During the projecting phase of this thesis, our main approach was to develop a custom build of the Chrome browser with our desired isolation changes. After some in-depth search and testing, we reached to the conclusion that this would not be a good approach for our problem. Due to the nature of the pieces of code that we were trying to modify, some of Chrome's built in security mechanisms were preventing us to accomplish our isolating changes.

Since the Chrome's native WebAssembly runtime is also isolated inside the active renderer process, via the used v8 instances, communication with outside processes is not allowed. To try to overcome this issue, we dived into Chrome's IPC mechanism (mojo) in order to create a custom communication

process that would be able to bypass this security mechanism and give us the functionality that we needed to build *WArdian*. Our several attempts were mostly unsuccessful and only correctly created our communication channel when Chrome's renderer sandboxing mechanism was previously disabled.

Although this approach would solve our problems and still give *WArdian* the minimum security mechanisms needed, it would severely damage the overall Chrome's browser security, so we opted for a different solution: a Chrome extension. Running as a separate Chrome process, a Chrome extension has the ability to directly change and communicate with web page's content and consequently bypass the previous problems. In the following sections, we present the main stages, and consequent setbacks, of the *WArdian* extension development.

5.2.1 Web Page Content Access

In order to detect and intercept WebAssembly calls, we first need to have access to the web page contents, i.e. DOM and JavaScript code. Since Chrome extensions are also sandboxed, and therefore run in individual and separate processes, the *WArdian* extension needs to be granted several browser permissions, as described in its manifest file.

```
// manifest.json
[...]
"permissions": ["tabs",
               "activeTab",
               "declarativeContent",
               "nativeMessaging",
               "webRequest",
               "webRequestBlocking",
               "downloads"],
[...]
```

The described permissions enables *WArdian* to have access to all open browser tabs and to be able to distinguish them from each other. This is very important since we need to remain isolating different scripts, from different sources, to have access to each other's content. *WArdian* also has permission to engage in native messaging communications, as explained in Section 4.6.1, and several web requests and download permissions to actively monitor and detect *wasm modules* calls.

The *WArdian* extension has now the ability to track all the information flow, from and between all browser web pages, but does not yet have the capacity from accessing and actually changing the web page's source code, since separate Chrome processes run in separate contexts. To overcome this issue, and as introduced in Section 4.4, a content script is necessary.

```
// manifest.json
[...]
"content_scripts": [
   {
     "matches": ["<all_urls>"],
     "js": ["contentScript.js"]
   }],
[...]
```

A content script is an additional extension file that can run in the same context as the renderer's web page and therefore access all the page's contents. By matching it with *all_urls* we can make sure that it will share context with web pages from all sources. This field could be changed in order to restrict *WArdian* access to web pages from certain sources, similarly to a white list.

5.2.2 Changing JavaScript Behavior

With access to the web page's content, we can now modify and insert new code in the current web page. Since we want to override the JavaScript function – *WardianFetchAndInstantiate* – that initiates the entire browser WebAssembly process, we can leverage on our content script to achieve this goal.

```
// contentScript.js
[...]
WardianFetchAndInstantiate = async function(a, b, c){
    console.log('Wardian: intercepted WardianfetchAndInstantiate');
    document.dispatchEvent(new CustomEvent('wasmcall', {detail: { wasm_mod: a, function: b,
        args: c}}));
    var result = await getResult();
    return result;
}
[...]
```

Our content script is executed by the *WArdian* extension, before the web page loads, to override the desired JavaScript function and insert a new one, *getResult*. *WardianFetchAndInstantiate*, instead of actually fetching and instantiating *wasm* code, now dispatches a *WArdian* custom event, with all the *wasm* information appended, that is detected by our background script. When the enclave operations are concluded and the *wasm* result is redirected to the extension, our content script is in charge of injecting it in the web page, via the variable *result*. The previously injected function *getResult* detects that this new variable exists and notifies the *WardianFetchAndInstantiate* so that it can finally return. This new process successfully changes JavaScript behaviour without the knowledge of neither the rendering engine nor the V8 engine, which securely isolates *wasm* code execution from Chrome's scope.

5.2.3 Wasm Data Structures

After detecting and retrieving all the *wasm module*'s data, this information needs to be exchanged with our enclave environment in order for it to be externally executed. Generally, the enclave needs three pieces of data for it to successfully load and execute the desired *wasm module*: the *wasm module* binary file, the name of the *wasm* function to execute and, if necessary, the function's arguments. In order to create this communication channel, our Web Cage driver is used.

The *WArdian* extension starts this process by web requesting the desired *wasm module* and storing it in a temporary local directory. To facilitate this data transfer with the enclave, we have developed a new python tool – *WasmToBytecode.py* – that creates a simple array with the dumped contents of the original *wasm* binary file, as shown in the next code snipet:

```
unsigned char wasm_module_file[] = { 0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00, 0x01,
    0x07, 0x01, 0x60, 0x02, 0x7f, 0x7f, 0x01, [...], 0x70, 0x01, 0x01, 0x01, 0x05, 0x03, 0x01,
    0x00, 0x63, 0x32, 0x38, 0x29 };
```

The Web Cage driver is in charge of creating this array buffer, appending to it the other necessary parameters and create a single *string* to be passed in to the enclave. Initially, in the QEMU prototypes, the arguments were inserted in header files (.h) to be imported by the WAMR extension, since the *Bao* shared memory communication protocol was yet to be implemented, which would result in large *wasm* header files that could be costly to exchange. However, after the *Bao* integration, large *wasm* binary buffers could be exchanged via shared memory in a chunked approach as explained in section 4.6.2.

5.3 Enclave Environment

The environment of our *WArdian* enclave was the subject of much deliberation during the development of this thesis. Several WebAssembly runtimes, and LibOs like systems, were studied and analyzed but our specific hardware architecture, and intentions of keeping it as lightweight as possible, were the decisive factors that made us go for Zephyr RTOS and the WebAssembly Micro-Runtime (WAMR). With the added advantage of native WAMR support for Zephyr, this combination proved to accomplish all *WArdian* requirements and both their official documentation and tech support, thought their respective GitHub pages, were extremely helpful during the development of *WArdian*.

5.3.1 Embedding WAMR in Zephyr

At this point in our work, and by following our iterative approach, we were still using QEMU to simulate our enclave and to test our WAMR/Zephyr environment. Although provided with good documentation, this embedment process required much work from our part since several configuration files (both in Zephyr, WAMR and QEMU) needed to be changed in order to accept our desired hardware architecture.

In particular, by specifying the Cortex-A53 on QEMU, and AARCH64 target architecture both in QEMU and Zephyr, via their respective configuration files, these components were ready to be deployed, but a WAMR application still needed to be developed and embedded in Zephyr before the final binary compilation. WAMR already has several simple application examples publicly available. We have leveraged on this documentation to learn and create our first 'hello world' type application to embed in Zephyr and test our entire enclave environment.

Concretely, our WAMR application is in charge of creating and managing the entire WebAssembly runtime since its creation and until its teared down. The application starts by allocating executable memory, inside our Zephyr build, and by cleaning it before loading the *wasm module* it receives as an argument. The application's main – *iwasm_main* – has three main tasks:

1. **Initializing the runtime:** The WAMR application starts by creating a new thread for the received *wasm module*, it then allocates executable memory depending on the selected hardware architecture and finally initializes the runtime environment.

2. **Executing WebAssembly:** With a runtime environment created, the application can now load the *wasm* byte buffer received from the Web Cage driver and use it to instantiate and load said module. Consequently, it can now invoke the *wasm* function received as argument, execute it and reply the result back to our Web Cage driver.

3. **Destroying the runtime:** The module instance can then be destroyed, alongside with the cleanse of the allocated executable memory and the tear down of the runtime environment.

Several *cross-compile* tool chains can then be used to directly compile the WAMR application to AARCH64, we chose *aarch64-none-linux-gnu* due to its stable current version and since it is a ready-to-use open source Tollchain with all the necessary tools for the Cortex-A family. After successfully compiling our WAMR application, Zephyr was then built leveraging on it's command line tool – west – that allowed us to build the final Zephyr binary, with our WAMR application embedded, aimed at our desired architecture.

5.3.2 Communication with the Extension

One unpredictable problem in our QEMU prototype was the communication channel with the extension, via Web Cage driver. Unfortunately, QEMU does not yet support network communications for their cortex-a53 emulation, this means that we can only communicate via stdin and stdout which severely decreases the performance. Instead of being able to start our QEMU environment and then dynamically send the *wasm modules* and arguments, we have to manually build our WAMR application with the arguments given as header files (.h) and only then start the simulated enclave environment.

Although we have automated this procedure so that our Web Cage driver can start the process and successfully receive the response from the enclave, this process needs to run every time when either the *wasm module* or its arguments change, which is a very frequent event. Table 5.1 presents this faced

performance challenge with the overall time wasted in the communications between the Web Cage and the QEMU environment prototypes.

Event	time (sec)
Web Cage requests wasm execution	0.00
WAMR app build starts	+ 0.12
QEMU environment ready	+ 4.95
WAMR app runs	+ 5.07
Web Cage receives result	+ 5.23

Table 5.1: QEMU prototypes performance setback.

As we can see, the actual *wasm* execution and enclave response, in average, is rather fast and invisible to the user but the fact that the application and environment needs to start every time a new *wasm* call is made is a serious setback in performance that completely damages *WArdian*. We continued to analyse this and similar issues, and our gathered results, and consequent analysis, is extended in Chapter 6. However, this problem is only relevant when considering the initial prototypes since our final architecture does not depend on QEMU emulation.

5.3.3 Dynamic Enclave Execution

Predicting the *Bao* integration and the usage of actual hardware to develop our next batch of prototypes, we started to fix the previous communication problem, attempting at creating a dynamic enclave that does not suffer from initialization performance handicaps.

To fix this issue we built a custom socket connection between the two endpoints: the Web Cage driver and the WAMR application. The Web Cage driver now starts the enclave environment as soon as the extension background process initializes and instead of passing the *wasm module* and arguments via header files (which made necessary the rebuilding of the app), the data is now exchanged via our socket channel which makes the dynamic WAMR execution possible and therefore eliminates the previous performance issue.

Figure 5.5 portrays how the communication was improved with our custom socket protocol and how both ends can connect and use the channel. Our Web Cage driver, built in python, triggers the initialization of the WAMR application, embedded in Zephyr at our enclave environment, that then sets up its socket endpoint and binds it to a known port to the driver (we used 9090). The WAMR app, developed in C, then waits for the Web Cage connection and, after accepting it, both endpoints can freely exchange messages until the socket connection is destroyed (which only happens when the browser session is terminated).

It is worth mentioning that this optimization was only tested with a simple client-server application since we were still working with QEMU. However, the ground work was mainly implemented and after integrating *Bao* only a couple of tweaks are necessary for our socket implementation to fully work in aid of *WArdian*'s performance.

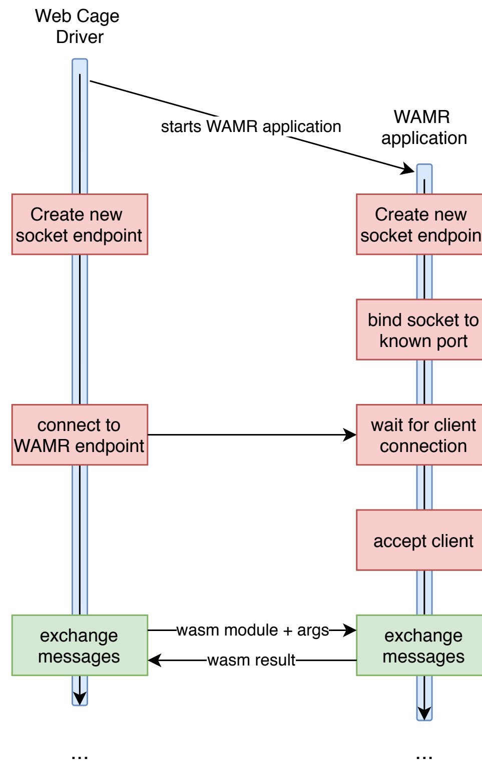

Figure 5.5: Socket communication improvement.

5.4 Bao Integration

With the previous prototypes, we were able to successfully develop and implement all of *WArdian*'s components at the Android OS zone (extension and web cage driver) and also the components at the *Bao* enclave (*wasm* runtime and web cage runtime). Now, to successfully build the final prototyping *WArdian*'s build, it was necessary to embed our zephyr binary in *Bao* and flash it to our hardware prototyping board – Rock960 – alongside with the latest Android OS image. Unfortunately, due to time restrictions, we were unable to finish this last prototype in time. This final implementation section presents all the current work in progress aiming at integrating *Bao* and *WArdian* in the same final build.

5.4.1 Hardware Setup

In order to debug our system, and consequently verify that each *WArdian* stage was being successfully implemented, we needed to have some sort of console access to the prototype environment while in execution. Also, a screen capture of the Android's environment, in real time, was extremely important to confirm the user's visual browsing experience.

To establish console access to the prototype board, an UART communication protocol was used with the help of an additional circuit board as explained in section 5.1.2. Both boards were wired together, as portrayed in Figure 5.6, in order to connect the GND(ground), RX(receiver) and TX(transmitter) pins of each board respectively. Both boards were then individually connected, via usb, to a development machine, e.g. Linux laptop, in order to have access to the wanted outputs. With this board's configura-

tion, Rock960 can be powered up and connected to an external screen via HDMI or streaming protocol, e.g. scrcpy, which we ended up using since it de-cluttered our working station and provided several automation advantages, such as drag and drop features to directly install .apk's on our Android build and the support for the native computer's keyboard and mouse to navigate the Android system.

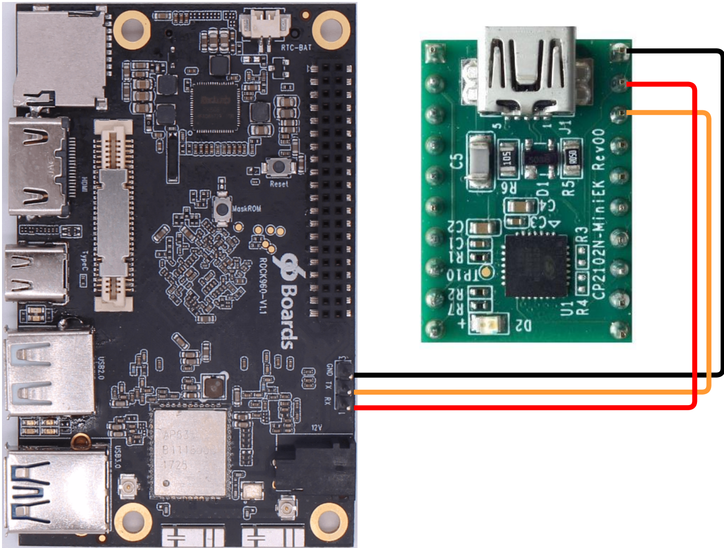

Figure 5.6: Debug wiring diagram. (black:GND, RX-TX:red, TX-RX:yellow)

With this setup, we were able to have visual output of the Android environment, via scrcpy as shown in figure 5.7, keyboard and mouse input via the development machine and serial console input/output via the UART communication with the debug board. Figure 5.8 shows the two main debug/testing terminal windows. On the left we have the UART output console, where we can check and verify several useful information, such as *Bao* booting stages and warnings, cpu interruptions, board network connectivity, etc. On the right, via adb shell, we can have access to the native Android console where we can manually run several commands, e.g. send *Bao* enclave commands and test response times.

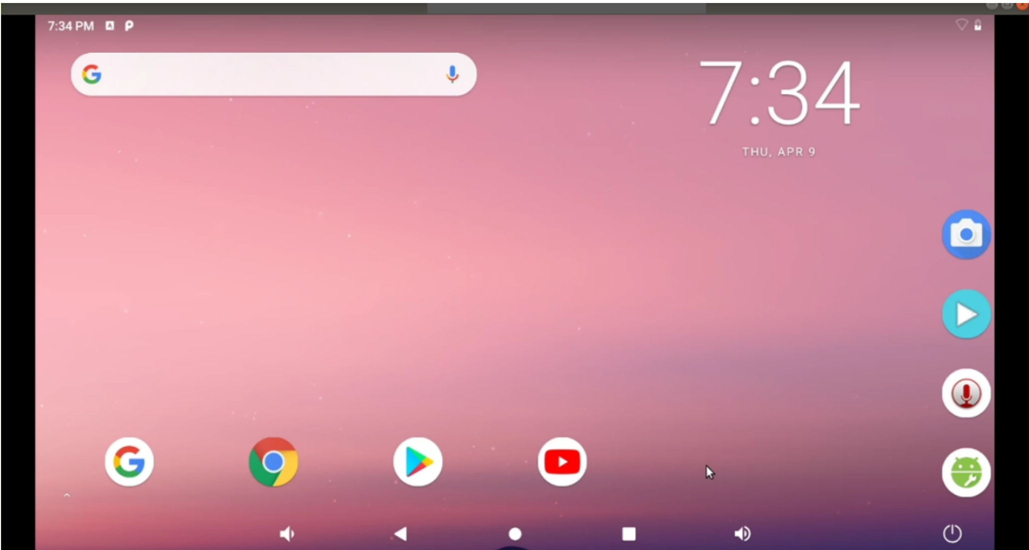

Figure 5.7: Android OS capture.

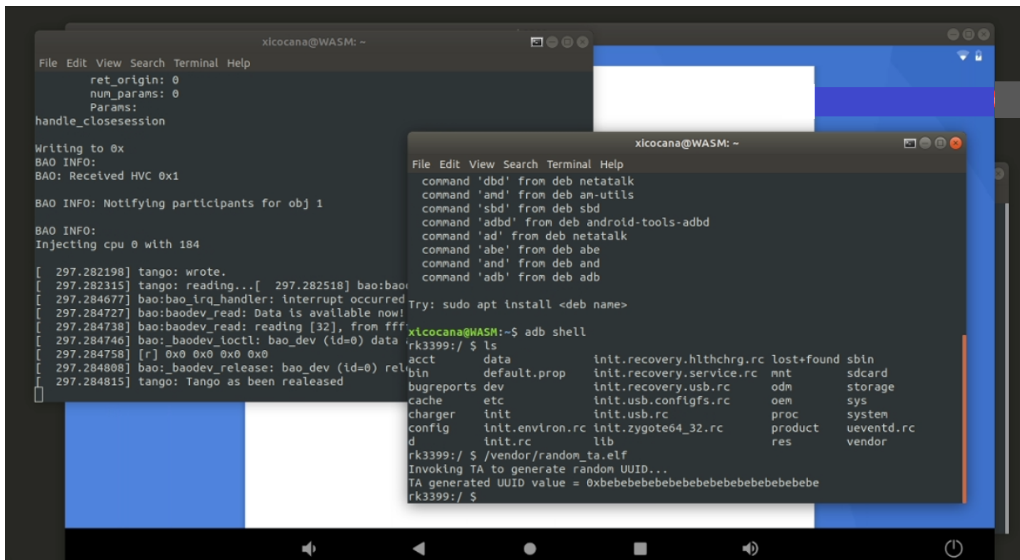

Figure 5.8: UART debug configuration.

In this particular example scenario, Figure5.8 shows a bare-metal *Bao* version responding to a simple ”hello world” type service. On the right terminal, via adb shell, we have access to the native Android console where we manually executed a simple TA (trusted application) that runs at the enclave and generates a random UUID. On the left terminal we can verify the entire *Bao* behavior while managing and executing said service.

5.4.2 Communication Protocols

Out of the box, *Bao* comes with 4 services that the *Bao* application running at the Android OS zone – HLOS – can request: start and close enclave session, invoke enclave command and cancel enclave command. Our *Bao* integration was started by adding two new commands to the custom *Bao* protocol.

As explained before, we need one command to start our WAMR application when a new browser session starts and an additional one (to be called frequently) every time *wasm* data and its arguments need to be communicated to the web cage environment. To implement these new calls, and with great help from our colleagues at Universidade do Minho, we altered *Bao*'s source code for these changes to take effect. But before looking at the changes made, we first need to understand how *Bao*'s internal messages are structured.

```
struct bao_msg_arg
{
    int cmd;
    int func;
    int session;
    int cancel_id;
    int pad;
    int ret;
```

```
    int ret_origin;
    int num-params;

    struct bao_msg_params params[0];
}
```

In order for both zones to communicate via the explained shared memory protocol, *Bao* messages need several attributes, such as return addresses, origins and internal paddings. However, to implement our *WArdian* new commands, we only need to take advantage of some of the available message fields.

Using the *cmd* field we can specify which *WArdian* command we are sending to the web cage environment: 0 for startup and 1 when requesting *wasm* execution. In case of *wasm* execution, we also leverage on the secondary *params* structure, where we insert all the necessary *wasm* data (*wasm module* binary dump, function called and its arguments), via web cage driver, that is then received and interpreted by our WAMR application. After instructing our web cage driver to create messages using the custom *Bao* structure, via HLOS, it is only a matter of interpreting this information in the web cage environment and execute our WAMR application as we are used to.

With this information in mind, our next steps to integrate *Bao* in *WArdian* were pretty straightforward. By adding some new instructions to the source code of *Bao*, specifically on the modules that run at the enclave – *bao_guest* – and interpret receiving messages from HLOS, and by embedding our zephyr final binary in the enclave we could create a similar behaviour known from our previous prototyping experiences.

```
// bao_msg.c
[...]
#define WARDIAN_START 0;
#define WARDIAN_ARGS 1;
[...]
// this function is called everytime a new msg reaches the enclave
// i.e. when a command is being invoked by HLOS
void handle_invoke(struct bao_msg_arg *msg){
    [...]
    switch(msg->cmd){
    case WARDIAN_START:
        // here we start the web cage WAMR application analogous to the QEMU prototypes
        start_wamr_app();
        break;
    case WARDIAN_ARGS:
        // here we send the wasm data to the running WAMR app
        receive_wasm_data(msg->num_params, msg->params[]);
        break;
    default:
```

```
        break;
    }
    [...]
}
[...]
```

Both functions *start_wamr_app* and *receive_wasm_data* were also added to *Bao*'s source code, however, some problems risen from these changes that we were not able to fix in time.

5.4.3 Prototyping Problems

After installing the Kiwi browser with our *WArdian* Chrome extension on our prototyping board we were ready to flash our final *Bao* and Zephyr build in order to start debugging our implementation and consequently retrieve some benchmark values once everything was in order.

Unfortunately, time constraints on the *Bao*'s development team, in combination with project synergies, delayed this final step in the integration of *WArdian* and *Bao*. Due to technical problems while flashing our zephyr binary on the *Bao* enclave, we were unable to have our web cage environment up and running in time to finish our final prototype.

Summary

In this chapter we presented the main developed prototypes alongside with detailed information about how we've achieved them, the consequent setbacks we were faced with and how we were able to solve them. We presented our entire development process, including the browser modifications made, our enclave environment setup and the integration of *Bao* into *WArdian*. In the next chapter we present the metrics used to evaluate *WArdian* and the consequent obtained benchmark values.

Chapter 6

Evaluation

In this chapter, we present the main evaluation results of our system. We start by identifying our concrete evaluation goals. Then, we present each of our main findings according to each of these goals.

6.1 Evaluation Goals

From the beginning of our work, a big emphasis was placed on device performance due to the major changes made to the WebAssembly browser runtime environment and consequently the *wasm* data life-cycle. *WArdian* main goals were always to improve mobile *wasm* execution security without disrupting the overall device's performance nor the underlying operating system's. With this in mind, we evaluate *WArdian*'s final results based on four main categories.

- **Performance**: We start by looking into performance benchmarks, where we evaluate our final solution, using not only global but also component specific metrics.

- **Resource efficiency**: We then move on to a more resource oriented metric approach, where we evaluate specific *WArdian* architecture modules and components in order to measure the overall *WArdian*'s efficiency and explore how each component is leveraged in terms of memory overhead and communication performance.

- **Usability**: *WArdian* usability is also measured. In this evaluation category we take a step back and try to fill the shoes of web developers and browser users in order to understand how *WArdian* fits into the overall developer's work-cycle and user's browsing sessions.

- **Security**: Finally, we look into the implemented *WArdian* security measures and construct a deep analysis on *WArdian*'s security, recalling not only the attacks and vulnerabilities mitigated by our final system but also the ones that are still in place.

6.2 Performance

Starting with performance evaluation, two new web pages were developed in order to facilitate results gathering. During the prototyping of *WArdian*, we leveraged on a simple web page with addition and multiplication *wasm modules* to verify the correct execution of our enclave environment. In order to run more compute extensive workloads, we felt the need to develop new *wasm* web pages.

WebAssembly Credit Card Verifier

This example validates a credit card number using the Luhn algorithm

Credit Card Number: 4716625342926335

Validate

(a) Credit card verifier

WebAssembly Char Counter

This example counts the number of chars of a given input paragraph

An example sentence with 39 characters!

WebAssembly Count

Result = 39

(b) Character counter

Figure 6.1: *WArdian* performance testing web pages.

We started by developing a more complex web page, as portrayed in Figure 6.1, that verifies if a given credit card number is valid and, if true, also detects which company it belongs to, e.g. Visa, MasterCard, etc. This new web page was implemented using the *Luhn* algorithm, that performs a series of operations on the given credit card number to verify its veracity. This is a more complex *wasm* computation that was built with the intention of measuring the efficiency of our internal *WArdian* components at the enclave, and consequently retrieve some interesting benchmark values.

We then developed another web page that receives a text input and returns a total character count. This second page was useful to evaluate several *WArdian* aspects, i.e. enclave communication speed and the underlying *Bao* shared memory communication channel, depending on the size of the given text input. Both web pages were leveraged to retrieve the benchmark values analysed in the next sections.

6.2.1 Global Performance

Starting with the overall *WArdian*'s performance, we wanted to measure the global impact of our system in browsing sessions, i.e. the total *wasm* operation's time with and without the *WArdian* extension enabled. To measure this, we changed our developed web pages to record the starting and ending time of the entire *wasm* operation, in milliseconds, and return the difference via browser console logs.

To perform this evaluation, we used a combination of all the developed web pages in order to get an average of *WArdian* execution times. To better understand how our system works, and consequently achieve more interesting conclusions, we started by evaluating our QEMU prototypes and then made our way to the *Bao* ones. Recalling the QEMU performance issues discussed in section 5.3.2, our gathered benchmark results for this set of prototypes were not surprising. The performance drawbacks of our

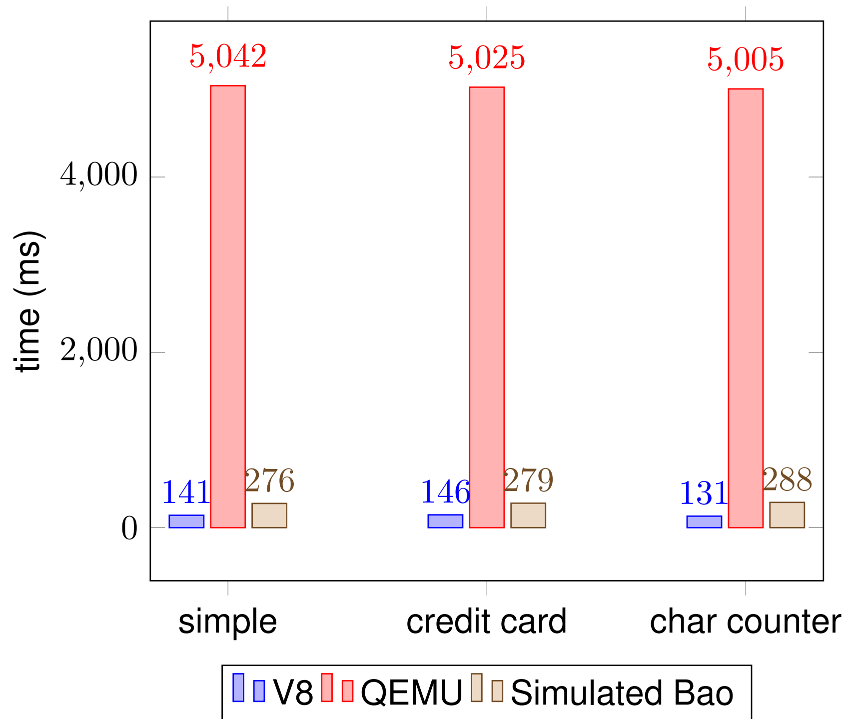

Figure 6.2: *WArdian* global performance.

QEMU prototypes severely impact the browsing user experience, due to the need to constantly rebuild our WAMR application, since QEMU does not yet support network communication on the used cortex.

At this point in time in our *WArdian* evaluation, the *Bao* hypervisor was still in development, which made us have to be creative while gathering the following benchmark values. Predicting the dynamically way in which we can run our WAMR application when using the *Bao* enclave, the need to rebuild our application vanishes, which means that this operation's time can be subtracted from our previous results. To accomplish this new batch of testing, we preemptively added our *wasm* arguments to the WAMR application before the browser *wasm* calls were made. Figure 6.2 shows the combined results of our batch of tests for the global *WArdian* performance, where we show the average operation's time of several WebAssembly browser calls, while using *WArdian* with QEMU, the simulated *Bao* environment and without the *WArdian* extension enabled, i.e. using the native browser's V8 engine.

As the gathered results show, all testing web pages show similar values, even when considering different and more intense workloads (as in the credit card samples) and even when the transferred data size increases (as in the character counter page), which praises the execution consistency of our chosen and developed enclave environments.

In terms of speed, and as predicted, the QEMU prototypes are severely hindered since the WAMR application needs to be rebuilt on each *wasm* request, and therefore, the added operation time is very noticeable. Nevertheless, in scenarios where QEMU is the only environment available option, and there is a need to securely isolate *wasm* executions and its data, this might be a trade-off in performance that it's users are wiling to accept.

Comparing now the *WArdian*'s simulated *Bao* prototypes against the native Chrome browser, the results are far more similar and almost unnoticeable to the human eye. Although there is a significant

increase in time, approximately 2 times slower, we consider these to be successful *WArdian* results since user experience is unaffected and almost unnoticeable to most users. It is also worth mentioning that, in our simulated *Bao* testing, we are still having to instantiate a QEMU virtual environment everytime a *wasm* call is made, even without rebuilding the app, which also adds to the global operation's time. We predict that the actual *Bao* final prototypes will be even faster than our obtained predictions.

6.2.2 Micro-benchmarks

After the global performance tests, we moved on to fine-grained testing. Since *WArdian* is a complex system, with several components communicating and working together in different ways, we felt the need to evaluate each component separately, and in specific, measure each communication channel.

The total *WArdian* execution time can then be separated into two parts: browser and enclave communication; and WAMR application's execution time. Knowing that the communication channel is composed of several components, we arrived at the following *total time* equation.

$$TotalTime = 2 * (t1 + t2 + t3) + t4 + t5 \tag{6.1}$$

Our complex communication channel starts at the browser. Chrome passes the *wasm* data to the *WArdian* extension (*t1*), which then needs to be communicated to our web cage driver (*t2*), and only then does it reach the enclave environment (*t3*). The WAMR execution time is represented as *t4* (with *t5* being the application build time) and the communication channel time is doubled since a response with the final *wasm* result needs to be sent back to the browser.

Using this equation, and the gathered results from the previous section, we were able to isolate each communication *bridge* and gather values to evaluate each component's performance. To achieve this, several changes were made to the *WArdian* components. Keeping in mind that the sample web pages were already recording timestamps, we added the same functionality to the extension's background process, a log file for our web cage driver to do the same and finally added a new feature to the WAMR application so it could also record it's total execution time. After these small changes were implemented, we ran the same batch of tests from the previous section and gathered the following values.

Web Page	t1	t2	t3	t4	t5	total (ms)
Simple	0.005	0.002	0.354	0.014	4.306	5.042
Credit card verifier	0.006	0.002	0.437	0.015	4.119	5.025
Char counter (small)	0.009	0.001	0.435	0.015	4.103	5.007
Char counter (medium)	0.010	0.002	0.462	0.015	4.012	4.975
Char counter (large)	0.009	0.002	0.451	0.015	4.063	5.002

Table 6.1: *WArdian* QEMU Micro-benchmarking.

As the results show, and unsurprisingly, the QEMU prototypes present very similar results on all web testing pages. Since QEMU runs in the same space as the underlying operating system, the communication between the web cage driver and the WAMR application is just as simple as two native

programs sharing information. We can also detect that the communications between the browser, the *WArdian* extension and the web cage driver (*t1* and *t2*) are very insignificant, which makes sense since these are all Chrome shared processes (the renderer, the extension and the driver), that communicate via it's inter-process mechanisms[14] and stdin/stdout respectively.

Since the communication between our *WArdian* extension and the web cage driver also happens in the same memory space, it's unsurprisingly fast, and considering that the enclave environment is the same in all *WArdian*'s prototypes, the communication between the driver and the enclave is the most impactful in the total procedure's time.

Regarding the *Bao* values, we once again had to improvise in order to obtain them, since the final prototype was not yet completed at this stage. To accomplish these simulations, we then again removed the compilation time of the WAMR application and used our socket approach to simulate the data transfer between the web cage driver and the enclave environment. In the implementation phase of our integration of *Bao* into *WArdian*, we developed a socket custom communication, based on a chunked approach, as explained in Section 5.3.3. We manually ran our testing messages through this channel to simulate the *Bao* shared memory protocol. The communication between the browser and the extension, and the execution time of the *wasm* application, are expected to remain the same between both prototypes.

Web Page	t1	t2	t3	t4	t5	total (ms)
Simple	0.005	0.002	0.124	0.014	NA	0.276
Credit card verifier	0.006	0.002	0.124	0.015	NA	0.279
Char counter (small)	0.009	0.001	0.124	0.015	NA	0.283
Char counter (medium)	0.010	0.002	0.125	0.015	NA	0.289
Char counter (large)	0.009	0.002	0.128	0.015	NA	0.293

Table 6.2: *WArdian* Simulated Bao Micro-benchmarking.

As our simulated results show, the overall *Bao* total execution time is much shorter, as expected, since the WAMR application can now run dynamically and there is no more need to instantiate a new virtualization environment each time a new *wasm* call is made.

One interesting aspect of our simulation, is that with the use of our socket testing prototype we were actually able to retrieve more realistic values in regards to the size of the *wasm* data transfered to the enclave environment. As shown in Table 6.2, we can see that time increases as the input data, and consequently the information sent to the enclave, becomes larger. With our QEMU prototypes we were not able to reach any maximum value of tolerance, but we believe that, on a real *Bao* prototype, some performance problems could surface from this issue.

To conclude this section, we emphasise that these are only simulated benchmark values and that they could be extremely different on a real functional final prototype, where we expect them to be even better since these values still have residual time damaging influences from the QEMU results.

6.3 Memory Footprint and Code Size

In terms of memory utilization, it was a *WArdian* goal, since the beginning of this project, to develop a small footprint web cage environment to occupy the least possible amount of enclave and device's memory space. To this account, our final Zephyr binary file, with the WAMR application already embedded, comprises a total of 3.9 MB which, even without a maximum enclave size to take into consideration, we feel like it's an acceptably short sized memory allocation.

Component	C	JavaScript	Python
WAMR application	239	–	–
Extension content	–	45	–
Extension background	–	61	–
Web Cage Driver	–	–	96
WasmtoBytecode.py	–	–	45
Socket optimization	45	–	33

Table 6.3: *WArdian*'s source lines of code (SLoC).

In terms of WArdian code size, the total SLoC for each component are presented in Table 6.3. This small code base reflects the overall low degree of complexity of the system. Most of the code is written in C and JavaScript, in accounts to the WAMR application and the WArdian extension respectively, although some components, such as the web cage driver at the Android OS zone, were developed in Python to simplify some functionalities and overall component's integration. Due to the small code base running at the browser and driver, *WArdian* presents a short attack surface on the Android OS zone, which considerably shortens the possibility of a malicious agent being able to compromise our system.

6.4 Usability

In regards to *WArdian* usability, both the users and the developers need to be taken into consideration. *WArdian* is intended to completely run in the background and disrupt browsing sessions and web page development as little as possible.

Regarding users, after the initial browser extension installation, the entire *WArdian* process is invisible and inconspicuous to the browsing section. In the current *WArdian* state, the extension is missing an options page where users could manage some of the extension's functionalities, such as choosing which *wasm modules* to isolate, but we consider this to be an advantage since no user interaction is needed after the initial setup. Considering that a browser user could use Chrome without knowing that *WArdian* was installed and without noticing anything strange about the browsing session, we consider our implementation to be a success regarding user experience.

On the other hand, currently web developers need to comply with restrict JavaScript syntax calls in order to successfully leverage on *WArdian*'s security mechanisms. This means that, currently, our browser extension only works properly if web developers consider it's existence and choose to obey it's restrictions. Although complying with *WArdian* syntax does not change the normal JavaScript flow

in *non-WArdian* browsers, it can pose as an additional time constraint for web developers to consider, which might make them choose not to use it. In the future, it would be interesting to tackle this problem, and make *WArdian* inconspicuous to developers as well, in order for *WArdian* to correctly work in any scenario, independently of web developer's adherence.

6.5 Security Analysis

Regarding *WArdian* security measures, we have made some efforts to increase *wasm* browsing session's isolation, and consequently security, from an untrusted mobile operating system. By combining our browser modifications with the *Bao* hypervisor, we successfully mitigated most of the possible attacks potentially issued by a malicious browser or operating system. In this section, we explore in detail the possible scenarios that *WArdian* successfully mitigates, alongside with current vulnerabilities that might influence the correctness of our final system.

- Starting with attacks potentially issued by a malicious operating system, the *Bao* hypervisor gives us the memory separation we need to combat and overcome this adversary. By statically partitioning the memory zones of both the Android OS and the enclave environment, *Bao* ensures that the OS cannot access nor override the enclave memory space, which guarantees the safe isolation of all the enclave trusted applications.

- Another potentially problematic attack surface is the hardware processor where the android OS and the enclave are running. *Bao* overcomes this issue by allocating, at boot, one single processor core to be assigned to the enclave environment, and the remaining to the Android OS, making it secure from access attempts by a malicious operating system.

- Having secured the enclave environment, the *wasm* web applications running at the local browser can still be compromised. *WArdian* deals with this problem by removing all *wasm* processes from the browser's renderer and instead executes *wasm* at the enclave. This ensures that the *wasm* application, and the data it manipulates, are loaded and executed outside the scope of the Android OS, which makes it impossible for it to read or change its contents.

- The operating system cannot then access the enclave memory space but it can still compromise the web browser in order to manipulate the communication with the *WArdian* extension, and consequently the enclave. Keeping in mind that it is the browser that is in charge of starting the enclave process, and sending the *wasm* inputs, one might argue that the isolation of *wasm* execution is pointless if the malicious operating system can still tamper with its input and control if the output is correctly delivered to the local browser. However, the malicious OS cannot access the execution information and manipulated data, which makes this a Denial-of-Service attack, that although is not a concern for *wasm* information security, can disrupt the correct functionality of *WArdian* and prevent web assembly code to be executed locally.

Summary

In this chapter we evaluated our final solution based on its performance, resource efficiency, general usability and resulting security mechanisms. In each metric we present the used batch of tests, the achieved results and our thoughts on them. In the next, and final chapter, we conclude this report with our final conclusions alongside with our major achievements and some ideas for future work.

Chapter 7

Conclusions

Although *WArdian* is not yet in a stable stage neither for users nor web developers, it already presents a strong foundation for secure WebAssembly browser execution. The current achieved mobile device's performance can leave *WArdian* as an handicap for recurring and systematic *wasm* executions but already gives the users the capability of isolating *wasm* code from malicious browsers and operating systems that can be extremely useful when dealing with data sensitive *wasm* operations. On the other hand, web developers currently have a well defined set of rules to follow in order to successfully leverage on *WArdian*'s security mechanisms, which might make it overwhelming to implement in their web pages, although sensitive *wasm* operations might justify the added work on their part and the consequent lack of performance on the user's device.

Regarding our performance issues, we should consider that WebAssembly is mostly used to perform intense and complex browser computations, e.g. media decoding, but these are not the kind of operations our system is intended to isolate. *WArdian* was developed to sandbox *wasm* browser operations that leverage and manipulate user sensitive data, like credit card verifications and money transfer operations, which execute faster and therefore will present a least noticeable performance constraint. Nevertheless, with all these factors in mind, and the overall security mechanisms that we implemented, we consider *WArdian* to be a successful project since the main sandboxing objectives were achieved.

In this final chapter we present the main *WArdian* achievements and conclude this report with some ideas for future work that would develop *WArdian* into a real world system with several real benefits for it's users and for web developers who intend to add sensitive *wasm* operations to their web pages.

7.1 Achievements

WArdian successfully achieves most of its initial proposed goals excluding a briefly noticeable lack in performance while comparing it with un-isolated *wasm* executions. Our Chrome browser extension takes control of *wasm modules*, and *wasm* operations in-browser life-cycle, making sure that the native Chrome execution environments – V8 instances – do not load, execute or even have knowledge of WebAssembly local executions.

The developed Web Cage Driver, in combination with the *Bao* hypervisor, successfully creates a secure communication channel with our enclave environment – web cages – which makes sure that no sensitive *wasm* information, or data manipulated by the *wasm module*, is leaked to a potentially malicious operating system and/or web browser, preserving the integrity and confidentiality of said data.

Finally, the used components of our web cage environment, i.e. the Zephyr RTOS and the WAMR runtime, work together to create a small footprint enclave that is able to run successfully without taking noticeable space nor memory capabilities of the mobile device.

Although, in the end, we were not able to successfully finish the integration of *WArdian* and *Bao*, the predicted combination of efforts between the *WArdian* developed components and the used hypervisor present the following measurable and successfully evaluated achievements:

- Secure isolation of WebAssembly executions from the mobile version of the Chrome browser without alerting it's native sandbox mechanisms.
- Execution of WebAssembly modules sandboxed from the underlying mobile operating system.
- Creation and maintenance of a small enclave environment, that is able to successfully perform all it's operations without consuming much of the device's resources.
- Preservation of confidentiality and integrity of the data manipulated by the WebAssembly modules.
- Low user install effort and invisible during most browsing sessions.
- Simple to integrate in existing *wasm* web pages.

7.2 Future Work

During the development of *WArdian* we were focused on creating a stable system prototype where we could prove and test our initial concept. That being said, several additional features could be implemented in order to develop *WArdian* into its full potential.

The current version of *WArdian* forces web developers to respect our JavaScript syntax restrictions which, although in a minimalist form, can increase the web application development time. In the future, it would be interesting to implement and override every single way to fetch and instantiate *wasm modules* into our *WArdian* browser extension to abstract the web page development from *WArdian* and even make web developers unaware of its existence. The introduction of HTML developer tags would also be interesting in order to enable web developers to consciously chose which *wasm modules* to isolate in *WArdian* and which modules to run in Chrome.

On the other hand, users could also benefit from an options page, where they could enable and manage the desired enclaved *wasm modules* with an intuitive visual layout to give them more control over the *WArdian* extension. This additional extension UI could give users the option to whitelist/blacklist different web site sources, enable/disable HTML developer tags, chose the amount of allocated memory to the enclave environments and so on.

Regarding our used hypervisor, some additional future work implementations could also take place in order to improve *WArdian*'s overall security and performance. The *Bao* hypervisor gives us many security mechanisms, that we successfully leverage on to create and communicate with our web cages, but we could explore it even further. In the secure world, *Bao* presents us a controller module that manages the creation and state of each running enclave environment. In the future, we could leverage on this additional component to enable the remote attestation of our enclaves from web servers that want to verify the integrity of our running system and overall isolation of the sent cagelets. The controller module can also enable the ability to seal enclave data and store it either at the Android's memory zone or in the secure world. This could be interesting to explore in order to securely save *wasm module* data to increase enclave performance, e.g. in case of increased recurring *wasm* operations during several different browsing sessions, persistent cookies, etc.

Finally, it is worth mentioning that our used WebAssembly runtime has native support for Intel's SGX, which means that, with minimal effort, our current working *WArdian* prototype could be transferred from mobile devices into desktop / laptop environments, making *WArdian* accessible in a multitude of different browsing devices.

Bibliography

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[2] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang. The "web/local" boundary is fuzzy: A security study of chrome's process-based sandboxing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2016.

[3] Google. *Chromium Sandbox Environments*, . `https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md`.

[4] Google. *Chromium Linux Sandboxing*, . `https://chromium.googlesource.com/chromium/src/+/master/docs/linux/sandboxing.md`.

[5] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. *USENIX Security Symposium*, November 2018.

[6] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[7] S. Checkoway and H. Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 03 2013.

[8] V. G. Lokhande and D. Vidyarthi. A study of hardware architecture based attacks to bypass operating system security. In *Security and Privacy*, 2019. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/spy2.81`.

[9] WebAssembly. *FAQ*, 2017. `https://webassembly.org/docs/faq/`.

[10] WebAssembly. *Application examples*, 2019. `https://webassembly.eu/`.

[11] A. Zakai. Emscripten: An llvm-to-javascript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '11)*, May 2011.

[12] A. Zakai. *Emscripten Compiler*, 2016. https://emscripten.org/.

[13] Google. *Chromium Multi-process Architecture*, . https://www.chromium.org/developers/design-documents/multi-process-architecture.

[14] Google. *Chromium Inter-process Communication*, . https://www.chromium.org/developers/design-documents/inter-process-communication.

[15] Google. *Chromium Process Models*, . https://www.chromium.org/developers/design-documents/process-models.

[16] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto. Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems. 01 2020.

[17] M. Musch, C. Wressnegger, M. Johns, and K. Rieck. New kid on the web: A study on the prevalence of webassembly in the wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2019.

[18] NVD. *CVE-2018-4121*, 2018. https://nvd.nist.gov/vuln/detail/CVE-2018-4121.

[19] NVD. *CVE-2018-4222*, 2018. https://nvd.nist.gov/vuln/detail/CVE-2018-4222.

[20] NVD. *CVE-2018-5093*, 2018. https://nvd.nist.gov/vuln/detail/CVE-2018-5093.

[21] NVD. *CVE-2018-6092*, 2018. https://nvd.nist.gov/vuln/detail/CVE-2018-6092.

[22] A. Lonkar and S. Chandrayan. *The dark side of WebAssembly*, 2018. https://www.virusbulletin.com/virusbulletin/2018/10/dark-side-webassembly/.

[23] A. Szanto, T. Tamm, and A. Pagnoni. Taint tracking for webassembly. In *arXiv preprint arXiv:1807.08349*, 07 2018.

[24] C. Reis, A. Moshchuk, and N. Oskov. Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019.

[25] Google. *Chromium Site Isolation*, 2019. https://www.chromium.org/Home/chromium-security/site-isolation.

[26] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song. Shadowcrypt: Encrypted web applications for everyone. *Proceedings of the ACM Conference on Computer and Communications Security*, 11 2014.

[27] Mozzilla. *Shadow DOM*. https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM.

[28] M. Freyberger, W. He, D. Akhawe, M. Mazurek, and P. Mittal. Cracking shadowcrypt: Exploring the limitations of secure i/o systems in internet browsers. *Proceedings on Privacy Enhancing Technologies*, 04 2018.

[29] K. Shanmugapriya and C. Geetha. Patterns for it sandbox innovation. In *International Journal of Pure and Applied Mathematics*, 01 2018.

[30] Google. *Meltdown/Spectre*, 2018. `https://developers.google.com/web/updates/2018/02/meltdown-spectre`.

[31] I. Sanchez-Rola, I. Santos, and D. Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *26th USENIX Security Symposium (USENIX Security 17)*, August 2017.

[32] M. Weissbacher, E. Mariconti, G. Suarez-Tangil, G. Stringhini, W. Robertson, and E. Kirda. Ex-ray: Detection of history-leaking browser extensions. In *Annual Computer Security Applications Conference*, december 2017.

[33] Q. Chen and A. Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[34] Chromium. *V8 Engine*. `https://v8.dev/`.

[35] B. McFadden, T. Lukasiewicz, J. Dileo, and J. Engler. Security chasms of wasm. NCC Group Whitepaper, August 2018. URL `https://i.blackhat.com/us-18/Thu-August-9/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native_Exploits-On-The-Web-wp.pdf`.

[36] M. Vassena and M. Patrignani. Memory safety preservation for webassembly. July 2019. URL `https://arxiv.org/abs/1910.09586`.

[37] V. Costan and S. Devadas. Intel sgx explained. 2016. URL `https://eprint.iacr.org/2016/086.pdf`.

[38] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Nov. 2016.

[39] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza. Trustjs: Trusted client-side execution of javascript. In *Proceedings of the 10th European Workshop on Systems Security*, 2017.

[40] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. G. Jr., E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh. Fidelius: Protecting user secrets from compromised browsers. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[41] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*, Aug. 2017.

[42] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Oct. 2014.

[43] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. *SIGPLAN Not.*, Mar. 2008.

[44] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: Secure applications on an untrusted operating system. *SIGPLAN Not.*

[45] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, Aug. 2015.

[46] ARMSecurityTechnology. *Building a Secure System Using TrustZone Technology*, 2009. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html`.

[47] N. Asokan, J.-E. Ekberg, and K. Kostiainen. The untapped potential of trusted execution environments on mobile devices. In *Financial Cryptography and Data Security*, 2013.

[48] ARMSecurityTechnology. *TrustZone for Cortex-A*. `https://developer.arm.com/ip-products/security-ip/trustzone/trustzone-for-cortex-a`.

[49] S. Pinto and N. Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Comput. Surv.*

[50] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, Aug. 2016.