École Polytechnique Fédérale de Lausanne

A study of SFI through safe intermediate representations

by Francesco Berla

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Msc ETH Lucio Romerio
External Expert

Msc ETH Andrés Sanchez
Thesis Supervisor

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

February 10, 2023

Your fear of looking stupid is holding you back
— Unknown

Dedicated to my mother Daniela, my girlfriend Letizia and all my friends that supported me during those years.

Acknowledgments

I want to thank my supervisor, Andrés Sanchez, for his constant support, the fantastic discussions and the valuable comments. Additionally, I would like to thank my adviser, Prof. Mathias Payer, for his support during the project and for making the HexHive such a wonderful place!

I would also to thank all the members of the HexHive that made the last four months a remarkable experience. I really enjoyed the joyful atmosphere of the open-space, the small talks at the coffee machine and the fascinating discussions during the reading group. You made writing my master thesis a very fun experience, thank you!

Next I want to thank all my friends and family, specially my mother, that supported me during my long academic journey. Clearly, an important thank you goes to my girlfriend Letizia, that supported me in every step and worked in order to make my life easier every day.

The last "Thank You" goes to every schoolmate and Polygl0ts team member that kept my passion for this field alive and made me enjoy otherwise boring subjects.

Lausanne, February 10, 2023 Francesco Berla

Abstract

Software applications commonly use multiple libraries for achieving their functionality. Some of those libraries handle user-input and can be considered critical for the security of the whole program. A mitigation technique for the security risks of libraries is software fault isolation (SFI). SFI provides memory isolation and control flow restrictions, isolating the library in a sandbox. State of the art SFI technologies lead to excessive performance overheads or are very difficult to apply to large applications.

We propose a new technique for isolating native libraries using WebAssembly for SFI. Our technique embeds a WebAssembly runtime and the original library compiled to WebAssembly inside the linux Shared Object. The generated Shared Object exposes the same interface of the original native library by adding a code layer that redirects each function call to the WebAssembly runtime. By exposing the same interface, no source code modifications are required for changing from the original native library to the one isolated by WebAssembly.

The creation of a new Shared Object that exposes the same interface while providing isolation, is possible thanks to a layer of code that handles the function invocation, creates the correct data structures and invoke the corresponding function inside the WebAssembly runtime. As the creation of this code layer requires a considerable amount of time and is error-prone, we created a rewriter tool that is able to automatically generate the required code for simple libraries.

Libraries created using our technique have an execution time that ranges from 50% to 200% of the original libraries depending on the computational characteristics of the library. The large extent of performance ratios between original libraries and isolated ones are due to the effects of the Just In Time compiler of our WebAssembly runtime. On average, the performance overhead of our solution is small, making it a valid substitute for native libraries of security critical applications.

Contents

Chapter 1

Introduction

Modern applications are extremely complex and increasingly large. The vast dimension of those applications directly leads to a large attack surface. In addition to that, a considerable part of a regular codebase is constituted by third-party libraries that are not under the developers control, increasing even more the security risks. As those applications are a single entity with no security boundaries, a bug in any component can lead to a compromise of the whole application. The resulting scenario is worrisome: a single bug among a big attack surface can threaten the security of the entire application.

To help mitigate the security risk of large codebases that might even include third-party libraries, a useful technique is *fault isolation* [15, 24]. Fault isolation is a software engineering practice that limit the impact of bugs by segmenting the application in multiple *fault domains* with limited and well-defined connections with the rest of the application or the outside world. If applied correctly, *fault isolation* can effectively prevent many exploits by limiting the effect of bugs to harmless sections of programs or to uninteresting memory regions. To apply *fault isolation* correctly, the program has to be divided in such that the input parsing code is separated from the segments of the program that contains secrets or sensitive functions.

Traditionally, address space separation via process isolation has been used as standard isolation technique, considering each process a single fault domain. Process isolation allows every fault domain to run at native speed, but it incurs in high overhead when it invokes another fault domain's functionality, as it lead to context switch to the other process that involves the OS. *Software Fault Isolation* (SFI) is a technique that allows to provide fault isolation in the same address space, reducing significantly the performance overhead of switching to a different fault domain at the cost of increasing the execution time of the code inside a fault domain [34]. SFI allows fault isolation to have a significant less performance overhead for applications that communicate fault domains frequently. Nevertheless, fault isolation is still used sporadically,

in a coarse-grained fashion and after extensive evaluation for two main reasons: excessive performance overhead and usability concerns.

The performance overhead of a single fault domain created through SFI depends on the specific mechanism used to implement it. A fault domain using SFI blocks reads and writes to memory regions and code jumps to code that are outside the fault domain. There are multiple strategies to implement the checks that block the fault domain from accessing memory regions or jump to code paths outside its domain. Each of those strategies has a different performance overhead that determines the performance overhead of the SFI mechanism. Google Chrome NaCl [42] implements memory isolation thanks to 80386 segments and by limiting control flow operations. NaCl incurs in a performance overhead raging from 5% to 15% [31] compared to native applications, but it provides a very minimal *Control Flow Integrity* (CFI) mechanisms and it supports only C and C++. XFI [10] employs software guards and uses static analysis to verify all execution paths for possible violations of the memory sandbox. Additionally, XFI employs two stacks: a *scoped stack* and an *allocation stack*. The *scoped stack* is inaccessible outside the scope of each function or by computed memory references and serves as regular execution stack. The *allocation stack* holds shared data. XFI shows a performance overhead from 5% to 100% depending on the workload.

Orthogonally to the choice of SFI technology, it is generally possible to tune the granularity of the isolation to allow to sandbox portions of code of the desired size and the number of them. Theoretically, having more small fault domains allows reducing the effect of a vulnerability to a smaller memory region and code section. The communication between fault domains has to go through the host instead of going directly to the other fault domain. Similarly, every code jumps that bridges the fault domains has to be mediated and checked by the host. The additional host intervention compared to non-isolated programs creates a performance overhead every time a fault domain has to interact with another one [34]. Having more fault domains increases the number of occurrences of inter-communication between fault domains and therefore the total performance overhead. The security advantage and the performance overhead of having more fault domains leads to a security-performance trade-off that each developer has to take into account. As shown with NaCl and XFI, we can conclude that state of the art SFI solutions have a performance overhead that can be further increased by the number of fault domains employed. This performance overhead is an important factor in reducing the adoption of SFI technologies.

Similarly, the adoption of SFI in real-world applications has been held back by usability problems. Generally, we can define four categories of usability issues that have been identified in recent research or industry SFI technologies: unstable technologies, lack of maintenance, high engineering effort to adopt and restricted set of supported programming languages or hardware [3, 14, 24, 45]. It is important to note that not all SFI technologies display all of those problems, but many of those are very common. We will detail all the usability issues

in subsection 2.1.2.

As we have mentioned, state of the art SFI technologies present some problems in respect to security, performance and usability. Those problems limit the adoption of SFI in native applications, therefore denying the security benefits that fault isolation could provide to applications. In our opinion, the current SFI technologies do not have a good balance between security, performance overhead and usability. This opinion is supported by the low adoption of SFI in real-world applications. To improve adoption, it is required to have an SFI technology that improves on the security, performance, and usability trade-off.

In this thesis, we present a new technique for isolating libraries in native applications using WebAssembly [27] for SFI. The developed technique answers to the initial research question "Is WebAssembly a feasible technology for SFI in native applications?". As we will discuss in section 2.2, WebAssembly is a very promising SFI technology that is used extensively on the web and in Web Browsers [23]. At the best of our knowledge, only Gobi [24] has tried to use WebAssembly to isolate libraries inside native application. Their approach has limited applicability and require extensive modifications to the compiler toolchain. In this work, we claim that with our technique that works with any WebAssembly runtime we can attain even better results without requiring any modification to compiler toolchains.

The goal of our technique is to transform an application that uses a library to a new application that uses a sandboxed library by modifying only the latter. The result is a native library that any application can link to as it exposes the same interface as the original library while having each call invoked in a WebAssembly sandbox. In section 3.2, we will articulate the design of the transformed library.

Before developing our technique for native libraries, we analyzed the security properties given by WebAssembly, and we evaluated their fitness for being used as SFI mechanism. To accomplish it, we reviewed the literature that covers WebAssembly and we propose some personal considerations. We will detail our findings in subsection 2.2.3.

After the security analysis, we developed a tool to simplify and partially automate the process of creating native libraries isolated from the consumer application. This tool aims to reduce the engineering effort to employ WebAssembly as SFI mechanism in native applications. We will describe in more detail the design of the tool in section 3.3.

As part of our analysis, we evaluated the simplicity and the requirements of the process both with and without our helper tool. Following the porting, we tested the application using the isolated library for performance regressions and security improvements. To achieve that, we ran a set of benchmarks comparing the performance of the original native application with our application using the WebAssembly isolated library. For the security testing, we took a set of high severity CVEs that were affecting the original library and we tried to replicate them

against the isolated application.

During this project, we faced many challenges. The library transformation from standard native library to sandboxed library using SFI required an accurate design. The difficulty lay in the fact that we want to keep the external interface identical for avoiding code modifications on the application side without compromising performance excessively. The constraint of providing the same API forces the wrapper code around the sandbox to handle potentially different pointer sizes (WebAssembly uses 32 bit pointers currently, even if a proposal for 64 bit indexed memories exists [28]), to handle memory copying automatically and to work with an incomplete C API for interacting with WebAssembly runtimes.

Working with WebAssembly and its toolchain has proven to be more challenging than anticipated. WebAssembly is a new technology that undergoes constant changes and is still lacking many components to be considered a finished product. Documentation is totally absent for usages outside the Web or for non-trivial native use cases. We discovered that WebAssembly runtimes and compilers implements many features without listing them, forcing developers to read source code for discovering them and understanding their usage. The last major challenge stem from the multitude of coding styles that characterize C code-bases. The rewriter should be able to deal with different API design, C sintax flavors (such as K&R C syntax) and heavy macro usage.

In this thesis, we created a procedure that allows in a simple way to port a native library to WebAssembly and integrate it in an existing application. Additionally, we created a program that can automatically generate the "glue code" necessary for the integration in the host application allowing an easier and faster transition to isolated libraries. More generally, we showed that WebAssembly is a viable option as Software Fault Isolation mechanism from the security, performance overhead and usability point of views.

Chapter 2

Background

In this chapter, we present some notions related to Software Fault Isolation, WebAssembly and Control Flow Integrity that, we think, are necessary in order to understand our work and the challenges involved.

2.1 Software Fault Isolation

2.1.1 Overview

Fault Isolation [15] is a technique used to improve software reliability by isolating faults (errors or bugs) within separate parts of a system. This is achieved by dividing a system into smaller, isolated components and creating boundaries between them, such that if an error occurs in one component, it does not affect the functionality of the rest of the system. This approach reduces the risk of system failure, improves security, and makes it easier to identify and fix faults. Traditionally, *Fault Isolation* in software applications is achieved by dividing the program in multiple fault domains and running them in separate processes. The memory protections for process isolation given by the Operating System allows to isolate each process as single fault domain and therefore create a *Fault Isolation* system. When a fault domain wants to interact to another one (e.g., calling code in the other fault domain), the Operating System has to handle the context switch and it is an expensive operation.

Software Fault Isolation (SFI) [34] is a technique that allows to have fault domains inside a single address space with the goal of reducing the performance overhead. SFI prevents memory accesses and code jumps from one fault domain to another one unless they are redirected and checked by the host. The host is the software responsible for managing and, as we mentioned

before, filtering all request of memory accesses and code jumps from one fault domain to another one. A Fault Domain is never allowed to access host memory, therefore creating a sandbox that prevents an isolating library from affecting the host system. A sandbox is the technique of restricting the software component considered at risk or not trusted from accessing the rest of the system [16].

2.1.2 Challenges

In this subsection we will discuss the two main challenges of state of the art SFI technologies: performance overhead and usability.

There are multiple mechanism for implementing memory separation and code isolation between fault domains in SFI. At present time, all mechanisms that have been presented for SFI have a significant overhead.

As we have anticipated in chapter 1, SFI technologies suffers from an usability problem that hinder their adoption. We can group those problems in four categories: unstable technologies, lack of maintenance, high engineering effort to adopt and restricted set of supported programming languages or hardware [3, 14, 24, 45].

Unstable technology An unstable technology is a technology that has the potential to change unexpectedly and quickly. The changes force the software that uses that technology to adapt. The requirement of frequent and unexpected changes in the consumer software due to the variation on the underlying technology are generally not tolerable for stable software [2]. For this reason, the software engineering industry considers instability in dependencies a problem and tries to avoid it. If a technology is not considered stable, it is considered not suitable for being used for software products. Many state of the art SFI technologies are still in early development and they face drastic and constant modifications, making them a problematic solution in practice.

Lack of maintenance Another issue for industry adoption of SFI technologies is that many promising research projects are not maintained in the long run. Those projects generally serve only as proof of concept for a conference paper and the research groups behind them don't have the resources to keep working on them afterwards. Clearly, adopting a technology that is, or will, not be maintained is a very risky choice that very few companies are willing to take. Projects like Gobi [24], *Portable Software Fault Isolation* [14] and vWasm [3] have been presented with conference papers and have never been updated since. NaCl [31, 42] has been deprecated in favor of WebAssembly [13].

High engineering effort for adoption The next usability problem is the high requirement of engineering effort to adopt. Most of the advanced SFI technologies don't have as

goal simplicity or easiness of adoption. This result in solutions that are very difficult to integrate to existing or future applications due to very high knowledge or engineering time requirements. Generally, the process is made even harder by the lack of documentation, that is often not present for research project or new technologies.

Limited support An additional obstacle in the adoption of state of the art SFI technologies is the limited range of programming languages, platforms, or processors supported. Gobi [24] supports only C/C++, *Portable Software Fault Isolation* [14] supports only C on PowerPC, ARM and x86-32, XFI [10] supports only Windows on x86. The limited number of use-cases that can be addresses by single technologies reduce the adoption and that, in turn, reduces the traction that a project has and therefore limit additionally the adoption.

2.2 WebAssembly

In this section, we will explain all the required notions related to WebAssembly that are required for understanding our work. We will start by presenting an overview of WebAssembly in subsection 2.2.1, we will continue with its new extension for interacting with the host system in subsection 2.2.2 and we will finish with the main security features provided by WebAssembly in subsection 2.2.3.

2.2.1 Overview

WebAssembly (often abbreviated as Wasm) is a low-level, binary format for executing code in web browsers. It is designed to be a portable, efficient, and secure target for web applications [37], allowing developers to write code in multiple programming languages (such as C, C++ [9, 20], Rust [30], etc. [12, 21]) and run it on the web with near-native performance [32]. WebAssembly is a web standard which is supported in modern browsers [39], providing a way to run high-performance and complex applications directly in the browser without the need for plugins.

Since its inception as web technology, WebAssembly has evolved and it is used as compilation target for native applications. The WebAssembly binary instruction format runs on stack-based virtual machine that is designed to be encoded, loaded and optimized efficiently. The execution of programs compiled to WebAssembly is possible thanks to runtimes that interpret, *Just In Time* (JIT) compile or compile *Ahead of Time* (AoT) to native code. The most popular runtimes are: Wasmtime [6], an open source project that is supported by the Bytecode Alliance [4] (composed by Amazon, Arm, Cisco, Google, Mozilla, Microsoft and Intel amongst many others), Wasmer [35], an open source project created by an homonymous start-up and WAMR/iwasm [5], another Bytecode Alliance open source project that is aimed to be lighter weight compared to

Wasmtime, and WasmEdge [8], an open source project backed by the Cloud Native Computing Foundation that target cloud native applications.

An application compiled to WebAssembly is called a *module*. Each module has zero or more *imports* and *exports*. Import are functions that the module can use but are provided by the host and are not defined by the module. Exports are functions that the module implements and exposes to the host to be invoked. A standalone application exposes only the main function and a library module exposes all the functions that are part of its interface.

WebAssembly is still a new technology that is constantly evolving and many procedures and tools are evolving with it. WebAssembly 1.0 has been standardized only in 2019 [40] and the WebAssembly group is already working on the 2.0 version [41] that includes multiple improvements and changes. Using WebAssembly for targeting non-Web environments requires more efforts from the developers due to the smaller documentation compared to the Web counterpart.

2.2.2 WASI

WebAssembly System Interface (WASI) [7] is a standard interface that provides a secure system interface for WebAssembly programs. It defines a set of system calls and functions that Wasm programs can use to interact with the underlying operating system and perform common tasks like file I/O, networking, and system information queries. WASI is designed to be portable and implementable on a wide range of platforms, making it possible for Wasm programs to run in a variety of environments, including the web, cloud, and native applications.

Before the introduction of WASI, WebAssembly programs running outside the Web were very limited in the system functionalities they could access. Additionally, they generally had to rely on a specific API provided by the runtime used, limiting therefore portability and stability. WASI allowed runtimes to leverage a standardized specification and forced them to provide a common interface that all WebAssembly programs can use and expect.

A WebAssembly module using WASI can be either a *command* or a *reactor*. A *command* is a program with an entry point that has to be executed as first invocation. The mandatory entry point of a *command* has to be called `_start` and it should not take any argument nor have any return value. A *command* can be seen as a normal application that is invoked via its *main* function. A *reactor* has no entry point, but it might optionally export a function called `_initialize`. If the `_initialize` function is present, the function has to be run before everything else, and it can be invoked at most once. After that initial invocation, a *reactor* can invoke any other of its exports any number of times, preserving state between each invocation. A *reactor* can be seen as a library in the context of WASI.

Currently, WASI has a major limitation: it can be leveraged only from C, C++ and Rust. Even if the three languages just mentioned are very popular and established system languages, WASI ecosystem and its adoption would benefit from the possibility to be used by more languages.

WASI implements a capability-based security model, which allows it to enforce strong security boundaries between Wasm programs and the underlying operating system. A capability is a security token that represents a right to perform a specific action, such as reading or writing a file, or making a network connection. In the context of WASI, each Wasm program is granted a set of capabilities when it is executed, which determine the actions it is allowed to perform. These capabilities can be narrowly scoped and controlled, so that Wasm programs have only the access they need to perform their intended functions, and no more. This capability-based security model provides several benefits:

Isolation Wasm programs are isolated from one another, and from the underlying operating system, which helps to prevent security vulnerabilities from spreading.

Verifiability The security of a Wasm program can be verified by inspecting the capabilities it is granted, making it easier to detect and prevent security vulnerabilities.

Flexibility The capability-based security model is flexible, allowing Wasm programs to be granted different sets of capabilities based on their intended functions and the environment they run in.

Portability The capability-based security model is portable, allowing Wasm programs to run in a variety of environments with consistent security guarantees.

Compared to Linux Capabilities [11], the advantage of WASI is that a program can have a finer grained capability security, as the capabilities are granted per WebAssembly module and not on a per-process basis. This allows a single process to have multiple WebAssembly instances with different capabilities. An application could have multiple instances of the same module with different filesystem access to specialize them. Alternatively, a program can run multiple Wasm modules with each of them a different set of capabilities based on their requirements.

Overall, the capability-based security model of WASI provides a foundation for building secure Wasm-based applications that can run in a variety of environments.

2.2.3 Security

In this subsection, we are going to present an overview of the security-related features and problems of WebAssembly.

WebAssembly as an SFI technology has to provide two isolation properties: memory isolation and control-flow restriction. Memory isolation prevents memory reads and writes of memory regions outside the SFI domain. Every fault domain protected using SFI, has a memory region that it can write and read. In the SFI context, it is possible to have partitions of this memory regions in multiple smaller chunks that can be read or write only. WebAssembly memory is divided in unmanaged and managed memory. Unmanaged memory is completely controlled by the application without supervision by the runtime. At the moment, WebAssembly unmanaged memory is a single zero-initialized readable and writable segment, called *Linear Memory*. The WebAssembly Community Group has an active proposal for having multiple memories with different read and write permissions [26]. The lack of read only memory sections allows attacks primitives based on memory overwriting [17]. Managed memory is a dedicated storage handled directly by the runtime. WebAssembly code can interact with managed memory only indirectly via instruction but not directly interacting with the memory. Managed memory includes local variables, global variables, return addresses and values on the evaluation stack.

Control-flow restrictions blocks any control-flow transfer (jumps, function invocations or returns) to code regions outside the SFI one. This mechanism restrict the SFI code and block any interaction with the host system. Additionally, WebAssembly has only structured control-flow. Functions have limited target for jumps: end of the function block, any location inside the function block or destinations allowed by a statically computed jump table. Jumps to arbitrary addresses are not allowed.

SFI restrictions are generally implemented in one of two ways: *Dynamic binary translation* or *Inline reference monitors* (IRM) [33]. Dynamic binary translation works by interpreting each instruction and checking if it respects both the memory and the control-flow policies. IRM rewrites the program and adds all the checks inlined in the functions invocations. IRM offers better performance compared to *Dynamic binary translation* because interpreting code is expensive and thanks to static analysis there are multiple optimizations possible for optimizing the checks in IRM rewritten programs. State of the art SFI technologies, including WebAssembly use IRM [22, 31, 42–44].

In addition to SFI control-flow restrictions, WebWassembly has *Control Flow Integrity* (CFI) [1]. There are three types of external control flow transitions that need to be protected: direct function calls, indirect function calls and returns. Direct function calls and returns are protected by WebAssembly semantics. Direct function calls are protected by indexes for functions entries in the function section. Correct returns are guaranteed by a protected call stack. Indirect function calls are controlled at runtime with a coarse-grained type-based CFI. Generally, the type-based CFI at Webassembly level can be considered the bare minimum and not enough to stop control flow hijacks against advanced attackers [17]. Clang/LLVM includes a CFI implementation of fine-grained CFI that can be leveraged from languages targeting WebAssembly from this compiler. It has an improved mechanism for protecting indirect function

calls and provides a better type-based CFI thanks to operating the signature comparison at the C/C++/Rust language level instead of the WebAssembly one. The difference is significant because WebAssembly has only four native types, leading to small difference in type signatures between functions.

In contrast to the security features that WebAssembly implements, it is lacking some mitigations that are now considered standard for the majority of contemporary software. WebAssembly does not have stack canaries, this allows a buffer overflow to reach memory regions past the scope of the current functions. Even if the return address is protected, it is still a potential source of vulnerabilities. Additionally, there is no guard page between stack, heap and data sections. The result of this absence is that a buffer overflow can corrupt values in other memory sections without triggering any fault [17]. Generalizing, the lack of those basic mitigations increases the impact of a bug in the source code that is compiled to WebAssembly.

Chapter 3

Design

In this chapter, we will present the design of our project. We will start by enumerating our goals in section 3.1, then we will proceed by explaining the idea of isolating library using WebAssembly and we will conclude with the design of the rewriter tool.

3.1 Goals

The main goal of this project is to evaluate Wasm as an intermediate representation (IR) for SFI. To attain our main goal, we aim to isolate native libraries using WebAssembly and evaluate the feasibility of the process and the resulting library. Figure 3.1 shows the transformation that we would like to reach. For isolating libraries, we have designed a list of properties that we would like to have:

Low overhead The SFI isolated library should have a low overhead compared to its native, original counterpart.

Minimal effort for consumers The developer that will use the library in an application should be able to link the new application with minimal effort.

System Interface capabilities Native libraries might require to access system functionalities, WebAssembly has a system interface in WASI and we should be able to leverage it.

Moderate effort for library maintainers Library maintainers would need to create a new version of their library that leverages WebAssembly for SFI. The effort required to port the library should be manageable and be required only once, with minimal incremental updates when the API of the library changes.

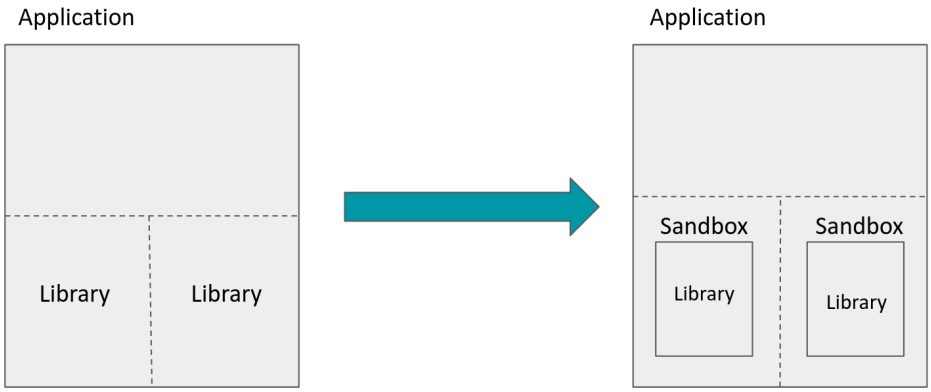

Figure 3.1: Architecture transformation from an application using native libraries linked without isolation to an application using native libraries isolated in the same process thanks to SFI.

3.2 Library isolation using WebAssembly

In this section, we will describe the design for the process of isolating native libraries using WebAssembly. We will start by describing the macro design in subsection 3.2.1 and we will finish by the micro design in subsection 3.2.2.

3.2.1 Macro design

To fulfill our goals, we propose to create a new shared object (SO) that exposes the same interface as the original library. The new library file embeds a WebAssembly runtime and a copy of the original library compiled to WebAssembly. The created library has two layers: an outer layer that exposes the interface and an inner layer that executes the function code. The outer layer is written in the same language as the original library and its purpose is to call the corresponding function with the right arguments in the inner layer. The inner layer is a WebAssembly runtime that executes the original library compiled to a Wasm module.

The outer layer is the interface of the library to the native application that consumes it. As we have mentioned, it exposes the same interface at the original library by having the same exact function signatures that the original library has. The outer layer can be written in any language that supports the following two requirements: ability to compile to a shared object and availability of language binding for interacting with a WebAssemly runtime. The ability to compile to a shared object is fundamental to be able to recreate a new library that can be linked instead of the original one. Similarly, the outer layer language needs to be able to interact with a WebAssembly runtime. Being able to interact with a runtime means the ability to create a new instance from scratch, load a wasm file as module, extract the export function

and invoke them.

The inner layer is composed by a WebAssembly runtime executing the original library compiled to a Wasm module. The runtime should be a WebAssembly compliant runtime that is preferably designed for embedding in applications. A runtime that is designed for embedding is generally smaller and omit features that are not strictly necessary for executing a WebAssembly module. Such undesired features include utility features like module inspection, module conversion, module testing and runtime upgrade [5, 6, 35].

As we have discussed in subsection 2.2.1, a runtime can interpret, JIT compile or AoT compile the WebAssembly module in order to execute it. In our project, we assume that the library is long-lived and intended to be called multiple times during the life of the application. For this reason, we compile Ahead of Time (AoT) the WebAssembly module before execution. The AoT compilation gives the best performance on average at the price of a slower startup. We consider that in the context of a long-lived application, the startup time is a secondary factor compared to execution performance of function calls. It is important to remark that the WebAssembly binary format is designed to be efficiently and fast compiled and optimized. This critical design property allows the startup overhead caused by AoT compilation to be small compared to other binary formats.

The last component of the inner layer is the actual WebAssembly module itself. The module is the original library compiled to Wasm. It has to be compiled in such a way that it exports all the functions that compose its API. Some libraries might need to be compiled with WASI support, making the procedure more complex.

3.2.2 Micro design

A C/C++ library is composed by two types of files: header files and the source files. The header files define the interface of the library and the source files contain the actual implementations of the functions exposed. The WebAssembly isolated library uses the same header files as the original library while replacing the implementation of each function with an invocation to the WebAssembly runtime for the same function. The result is a library executable that can be linked in any native applications in the same way as the original compiled library.

Recreating a shared object that embeds a WebAssembly runtime while exposing the same interface requires the next steps:

1. Compile the library source code to WebAssembly. Multiple libraries interact with the Operating System and therefore must be compiled with WASI support as reactors.
2. Create the C/C++ code that implements the functions exposed in the Application

Programming Interface (API). The implementations must invoke the equivalent function in the WebAssembly runtime with the correct parameters and export the optional return value.

3. Compile the original header files coupled with the newly created source files into a Shared Object library.

4. Link the new library to the original application.

WebAssembly version 1.0 supports only four basic types: `i32`, `i64`, `f32` and `f64`. Those four types represent respectively: 32 bit integer, 64 bit integer, 32 bit floating point value, 64 bit floating point value. WebAssembly pointers are expressed as 32 bit integer values. Some parameters are not of a type of one of the four available, in such case, WebAssembly declares them as i32 values, the value representing a WebAssembly pointer. WebAssembly declares the value as pointer because it cannot pass it by value as no type exists for representing it, and it is therefore obliged to pass the value by reference. A value that should be passed by reference may be allocated originally either on the memory of the host or in the linear memory of the WebAssembly module. Yet, a WebAssembly pointer can point only to a location inside its linear memory. If the value is already in the linear memory of the module, the value of the reference is simply the pointer of that element as it would be in standard C code. On the other hand, if the value is in the host memory, it is required to copy the value to the linear memory beforehand. The C code of the isolated library that composes the outer layer has to take in account those two possible scenarios. If the value is in the host memory, the library allocates the necessary space in the linear memory of the module, copy the value to the newly created allocation and uses the pointer as parameter for the function invocation.

The runtime needs to load the Wasm module during the initialization phase. For this reason the module should be available to the runtime at any moment. There are two possible ways of shipping the module to the runtime: keep it separate from the library and available on the filesystem or embed it in the library. In the first option, the runtime embeds a filesystem path that represents the location of the Wasm module on the computer. This solution has one advantage and one disadvantage. The advantage is that the wasm file can be modified separately from the rest of the library. If the library keeps the interface identical between versions, it would be sufficient to recompile it to a new Wasm module and no modifications to the Shared Object file would be needed. The disadvantage is that it may lead to version incompatibilities if the version of the wasm file does not match the version of the library of the SO file. We think that the risk of incompatibilities that arise from the need of synchronization of the wasm and the SO file are far greater than the advantage of reduce recompilation. For this reason, we opted for embedding the Wasm module inside the library file.

3.3 Rewriter

In this section, we will present a rewriter tool that aims to reduce the engineering work from library maintainers to create a new library embedding a WebAssembly version of their code. The rewriter tool automates the writing of the C code that invokes a specific WebAssembly function with the correct parameters. For each function that is part of the API of the library that we want to port, the rewriter extracts the parameters and writes the C code required for invoking the function in the WebAssembly runtime and return the optional return value to the caller. The code generated by the rewriter tool replaces the original implementation of the library.

As we have mentioned above, the rewriter tool takes a list C header files and parse each function definition to extract the name of the function, the parameters and the return value. To invoke a function of a WebAssembly module running in an embedded runtime from the C programming language, it is necessary to prepare the data structures that will be passed and carefully select the correct corresponding types. The information retrieved during the parsing of function signatures is necessary for filling those data structures. The rewriter links each parameter in the function signature to a data structure to be passed to the runtime.

The functions and the data structures required for interacting with a WebAssembly runtime are runtime specific. Nevertheless, it exists an active project that provides a runtime-agnostic API [38]. Our rewriter tool uses the runtime-agnostic API as much as possible and uses runtime specific functionalities only when needed. Additionally, the template used for invoking the WebAssembly functionalities is designed to be decoupled from the all the other components and therefore adapted to specific runtimes or developments of the WebAssembly C API.

The rewriter tool is not necessary for achieving the goal of recreating a native library isolated using WebAssembly, but it automates a workload that, otherwise, would have been done manually. We think that automating the process is very important for two reasons. Firstly, manually writing all the required code is very time consuming and a repetitive process. Secondly, the structure of the code required for interacting with the WebAssembly runtime is very error-prone and cannot checked by the compiler for the most part.

Chapter 4

Implementation

This chapter is dedicated to present the implementation details of our project. The implementation follows the design presented in the previous chapter. In order, we present the implementation of the process for porting native libraries to WebAssembly in section 4.1, the source code rewriting of the libraries in section 4.2 and the details of the rewriter tool in section 4.3.

4.1 Library porting to WebAssembly

The difficulty of compiling a library to WebAssembly largely depends on the OS interfaces that the library uses. The libraries that do not require WASI are generally easy to compile using a recent version of Clang. On the contrary, libraries that need to leverage WASI for archiving their functionalities require more care. Creating a library in the WASI context requires the module to be compiled as reactor as we have explained in subsection 2.2.2. Whether the library needs WASI support or not, we need to compile the Wasm Module such that it exports all the functions exposed originally.

Libraries that do not require WASI support are relatively easy to compile to WebAssembly. LLVM has support for targeting Wasm since version 8. Compiling a single file requires only specifying `wasm32` as target. The libraries considered in this project are tipically composed by multiple source files. For this reason we need to compile the file to intermediate object files and then link them together. Compiling to object files is as simple as compiling directly to WebAssembly files in the case of a single source file. The only requirements is to specify the files extensions as *.o* and not invoke the linker. After the object files have been created, the WebAssembly version of *ld* [19] can be invoked to link all the object files into a single wasm file. When invoking the linker for creating a library, it is necessary to pass some addi-

tional flags. The first necessary flag is `-no-entry` that specify that no entry point has to be exported, therefore, defining the WebAssembly module as a library. The second set of flags is necessary to export the functions that are part of the library API. There are two options for exporting the functions required: `-export-all` or specify every single function to be exported via `-export=`*"function_name"*. The first option exports every symbol present in the object files, including global variables, library functions and WebAssembly specific internal functions. The second option allows to export only the functions and variables required. In our project, we export explicitly all the symbols that we think are needed, even if it requires to explicitly specify all of them. Exporting all the symbols, even if convenient, has two major drawbacks. The first negative aspect is that all functions and symbols are exported, clobbering the interface and potentially exporting functions that should be kept private and not accessible from the outside. Secondly, the internal WebAssembly symbols that are exported may be unexpected by the runtime and led to a crash as they are no formally part of the allowed exports by the specification [29].

Libraries that require WASI support necessitate effort to be compiled. The first step when compiling a library that interacts with the system interface is to verify that all the required system functionalities are supported by WASI and by the selected runtime. In complex projects that have many compile-time flags, it may be required to experiment as WASI can be considered a new platform. Following the initial evaluation step, the following two steps are similar to non-WASI projects. All files are compiled to intermediate object files by specifying a sysroot folder that contains all standard library files ported to WebAssembly. Finally, the object files are linked together in a reactor with the `mexec-model=reactor` flag in addition to the linker flags that we have mentioned in the discussion of non-WASI compilation.

For every library that we ported to WebAssembly, we create a single Makefile that compiles the source code to WebAssembly with all the correct exports. The Makefile is structured in a way that the library maintainer has to do the configuration work only once and then incrementally update it if relevant changes on the source code or toolchain happens. Every source file is compiled with the `-O3` optimization flag using Clang. For WebAssembly targets, the WebAssembly toolchain does not rely uniquely on the optimizations made by the LLVM framework but it performs an additional WebAssembly specific optimization pass by invoking *binaryen* [36]. *binaryen* is a WebAssembly specific optimizer that loads a Wasm binary file and output an optimized version of it. *binaryen* allows performing WebAssembly specific optimizations that are not part of the standard LLVM pipeline.

4.2 Application rewriting

In this section, we will present the process of rewriting the source files from the original implementation to the invocation of the corresponding function in the WebAssembly runtime.

The application rewriting is split in two components: the addition of the runtime initialization code and the rewriting of the function bodies.

4.2.1 Initialization code

To be able to embed a WebAssembly runtime in the application, it is necessary to declare it in the source code and initialize it. In our project we decided to decouple the initialization code from the rest of the library. For this reason we have the initialization code in a separate header and source file that we compile together with the rest of the source and header files. The initialization of the runtime is done in a single function for simplicity and it is composed of four major steps: creation of the engine, creation of the storage for the linear memory and linkage to the engine, reading, and compilation of the WebAssembly module and loading of the module into the runtime.

The engine represents the executing core of the runtime and the compilation toolchain and it is therefore the first element to initialize. At the engine creation, it is possible to specify some parameters. The most important parameters are related to the later step of compilation of the module. It is possible to specify the optimizations levels and, in the engines that support it, the strategy (JIT or AoT). Following the arguments presented in subsection 3.2.1, we configured our engine to compile the modules Ahead of Time to improve performance.

Following the engine creation, we had to allocate the space for the linear memory. The linear memory can be populated before running, but in our project we never needed this feature, and therefore we decided to simply zero-initialize the memory. After the creation, the memory has to be linked to the engine to be usable from our module.

Following the creation of the memory, the intended WebAssembly module has to be read and compiled. As we have discussed before, we chose to compile the module Ahead of Time. The product is a binary file that can be executed by the runtime. Generally, the Wasm module is stored as a separate file and helper functions exists for compile a module from a file. We detailed in subsection 3.2.1 that we want to embed the file in the library to avoid to shipping two separate files. To link the wasm binary file to the library, we use the linker to create an object file from the WebAssembly file and then link it together with all the other objects file to the Shared Object. In the source code, we have to treat the binary file as an external symbol and use it as binary data.

The last step of the initialization is loading the compiled module into the runtime and start executing it. The resulting code can be seen in Listing 4.1.

```
void init() {
  // Set up our context
  engine = wasm_engine_new();
  store = wasmtime_store_new(engine, NULL, NULL);
  context = wasmtime_store_context(store);

  // Create a linker with WASI functions defined
  linker = wasmtime_linker_new(engine);
  error = wasmtime_linker_define_wasi(linker);
  if (error = NULL)
    exit_with_error("failed to link wasi", error, NULL);

  wasm_byte_vec_t wasm;
  FILE *file = fopen("insider04.wasm", "rb");
  ...
  fclose(file);

  // Compile our modules
  module = NULL;
  error = wasmtime_module_new(engine, (uint8_t *)wasm.data, wasm.size, &module
    );
  if (module)
    exit_with_error("failed to compile module", error, NULL);
  wasm_byte_vec_delete(&wasm);

  // Instantiate wasi
  wasi_config_t *wasi_config = wasi_config_new();
  ...
  error = wasmtime_context_set_wasi(context, wasi_config);
  if (error = NULL)
    exit_with_error("failed to instantiate WASI", error, NULL);

  // Instantiate the module
  error = wasmtime_linker_module(linker, context, "", 0, module);
  if (error = NULL)
    exit_with_error("failed to instantiate module", error, NULL);

  // Extract linear memory
  wasmtime_extern_t item;
  bool ok = wasmtime_linker_get(linker, context, "", 0, "memory", strlen("
    memory"), &item);
  memory = item.of.memory;

  hostLinMem = wasmtime_memory_data(context, &memory);

  extractFunctions();

  initialized = 1;
}
```

Listing 4.1: Code snippet that shows the initialization function.

4.2.2 Rewriting of the function bodies

The rewriting of the function bodies is the more complex and important part of our design. For every function implementation, we substitute the content with an invocation to the WebAssembly runtime. Each rewritten function body is composed by five blocks of code that represents the steps necessary for correctly calling the function via the runtime.

The first code block is composed by the check that test if the initialization has already taken place (see Listing 4.2). If the check fails, the initialization function is called. This step is essential because the library has no defined entry point as any exported function could be called as first and before continuing it is necessary to have already initialized the WebAssembly runtime. This means that the first function invoked has to call the initialization function. At the end of the initialization, a global variable is set to symbolize that the process has been completed.

```
1 void functionName() {
2   if (initialized == 0) {
3     init();
4   }
5   ...
6 }
```

Listing 4.2: Code snippet that shows the initialization check at the beginning of each function

The second step is to extract the corresponding function reference from the running Wasm module. There are two possible methods for encoding the function extraction. The first option is the simplest: write the extraction code block inside the body of the function. This approach guarantees that the function reference is available before the call, and it does not require any additional global data structure but it leads to repeat the extraction every time that the function is called. The second option consists in creating a global function table that each function can lookup. Creating a function table requires to coordinate the function invocation in each function block with the structure of the table to guarantee that the right function is called. Additionally, the table has to be populate before the call. We designed two strategies for ensuring that the table contains the function we are going to invoke. The first strategy consists in initializing completely the table during the initialization phase (see subsection 4.2.1). This option guarantees that every possible function is present in the table at the end of the initialization. From a performance perspective, filling the table increases the startup time but does not increase the performance overhead of single function invocations. The second strategy that we have developed consists in filling the table in an opportunistic way: at each function invocation a check is performed, if the function is not defined in the table, it is extracted and the entry is filled. The second strategy allows a faster startup time and potentially saves time by not extracting functions that will never be called. Yet, initializing the table in an opportunistic way, increases the complexity of the function body and the execution time of the

first invocation of each function. For our libraries, we implement our function extractions using a function table filled during initialization because we think that it gives better performance compared to extracting it every time and it is a simpler approach compared to opportunistic extraction.

The third step consists of the creation and filling of the data structures that represent the arguments of the function that we are calling. In order to do that, we link the parameters passed to the function to the parameter that we will use for the invocation of the WebAssembly function. Every parameter requires a data structure that specifies the value and the type of the parameter. Every parameter is then aggregated in an array to be passed to the runtime. The return value requires a similar data structure.

The fourth step is the actual invocation of the function inside the runtime. The invocation requires that all previous steps have been completed successfully. We invoke the function with all the global variables set up during the initialization phase, the function reference and the data structure filled for the parameters and the optional return value.

The last step is to extract the value from the return data structure, cast it to the required return value and actually returning it. Below, we show a comparison between an example function body and its rewritten version that includes all the steps mentioned above.

```c
int sum(int a, int b) {
  return a + b;
}
```

Listing 4.3: Original function

```c
int sum(int a, int b) {
  if (initialized == 0) {
    init();
  }
  wasmtime_func_t func = funcs[0];

  wasmtime_val_t params[2];

  params[0].kind = WASMTIME_I32;
  params[0].of.i32 = a;
  params[1].kind = WASMTIME_I32;
  params[1].of.i32 = b;

  wasmtime_val_t result[1];

  error = wasmtime_func_call(
    context, &func, params, 2,
    result, 1, &trap);
  if (error = NULL || trap = NULL)
    exit_with_error("failed to call
     function", error, trap);

  return (int)result[0].of.i32;
}
```

Listing 4.4: Rewritten function

4.3 Rewriter Tool

In this section, we detail the implementation of the rewriter tool. As we have described in section 3.3, the purpose of the rewriter is to automate the process of rewriting the library source code as described in section 4.2. The program is written in the Python [25] programming language.

The rewriter takes a list of header files and processes them individually, writing the corresponding source files one by one. The generated source files have the same name as the header file with the file extension changed. The rewriter creates individual source files instead of a single large one in order to maintain clarity and easiness of debugging. For each header file, the tool parses all the functions definitions and extracts three information: the name of the function, the types, and the names of each argument and the return type. This information is later stored in a data structure for efficient parsing. The next phase consists of writing the new source files corresponding to each header. The user of the rewriter tool needs to provide a template for the body of the functions that will constitute the new rewritten ones. This allows to adapt easily the tool to changes in the runtimes API, to specific requirements or to

different implementation strategies (for example, it is possible to modify the extraction strategy, see subsection 4.2.2). For each parameter of the original function, the writer links it to the required data structure in the template and writes the corresponding code. The rewriter has to transform the original type in one of the four numeric types supported by WebAssembly. Everything that is not convertible to one of those four basic types has to be converted to a pointer and passed by reference. Every time a conversion to pointer is made, the rewriter adds the corresponding code for copying the value from the host memory to the runtime linear memory. After the invocation of the function from the WebAssembly runtime, the rewriter adds the optional line for the return value. Generally, a cast from the Wasm type is necessary, in that case the rewriter determines the correct cast based on the original return type and it applies it to the return value.

The second task of the rewriter is to automatically create the initialization files and functions. The initialization files are composed by an header file and a source file, following the design that we presented in subsection 3.2.2. Similarly to the body of each individual function, the initialization function code is based on a customizable template.

Chapter 5

Evaluation

5.1 Comparison against native code

In this section, we will compare a library created using our method that leverages WebAssembly for intra-process isolation, to an identical library directly compiled to a Shared Object. We will start by comparing performance in subsection 5.1.1 and finish by evaluating usability from the point of view of the developer using the library in subsection 5.1.2.

5.1.1 Performance

The performance evaluation is done by comparing a C library directly compiled to a Linux Shared Object (SO) and the same library compiled using our method explained in subsection 3.2.1. The library exposes functions for mathematical computation, struct allocations on the heap and their manipulation and functions that combine them. The functions represent groups of possible programs (simple arithmetic computations, heap memory interaction) and cover the executions that represent a difference from native libraries and libraries embedded in a WebAssembly runtime. The most important difference between native libraries and embedded ones is the interaction with memory. In native application there is only one memory, while in the sandbox, memory has to be copied back and forth from the sandbox to the consumer.

Software	Clang	gcc	Wasmtime	binaryen	Wasi-sdk	Linux
Version	15.0.7	12.2.1 20230111	5.0.0	111	14	6.1.7-hardened

Table 5.1: Software setup for the evaluation.

The software setup of the experiment can be seen in subsection 5.1.1. The hardware setup is shown below:

Computer model Lenovo ThinkPad X1 Carbon 9^{th} generation

CPU 11^{th} Gen Intel Core i7-1165G7 @ 2.80Ghz

RAM 32 GB

The WebAssembly runtime is instantiated with the default configuration. The code generation profile is "WASMTIME_OPT_LEVEL_SPEED" that indicates that the generated native code during AoT compilation and JIT is optimized for performance.

The experiment consists of 4 programs:

1. MathHigh: mathematical heavy computation with high context switch from the runtime and the host (around 100000 context switches per second). The program is composed by a short initialization followed by 2000000 invocations to the runtime of a non optimized recursive fibonacci algorithm.

2. MathLow: mathematical heavy computation with low context switch from the runtime and the host (around 10 context switches per second). The program is composed by a short initialization followed by 40 invocations to the runtime for a function that performs 9000000 optimized (no recalculation) fibonacci computations of the number passed as parameter.

3. MemHigh: memory heavy computation with high context switch from the runtime and the host (around 100000 context switches per second). The program allocate a structure on the heap inside the runtime, alter a single value of it 10000 times and free the whole structure 10000 times in a row. Note that all of those operations are executed on individual runtime functions invocations.

4. MemLow: memory heavy computation with low context switch from the runtime and the host (around 10 context switches per second). The program allocates a data structure inside the runtime memory and then invokes a single runtime function that internally modify all the values of the data structure 1000000 times.

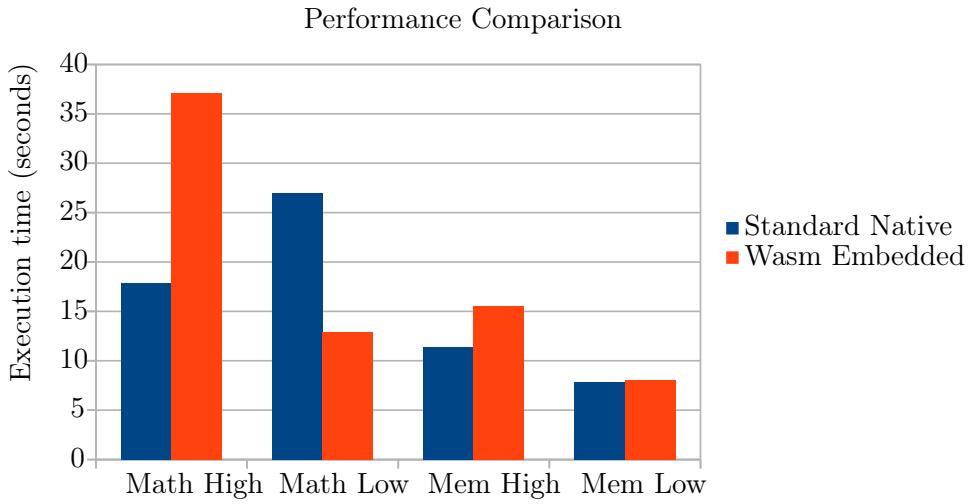

Figure 5.1: Results of the performance evaluation. The values express execution time measured in seconds. The "Math" prefix represents computational intensive programs, the "Mem" prefix memory intensive ones. The "High" and the "Low" keywords represents, respectively, programs with numerous context switches and programs with low context switches.

Every program has two versions: one with the standard native library linked and one with the embedded WebAssembly runtime linked. We consider a "context switch" the invocation of a function that requires a call to the Wasm code in the native library that embeds a WebAssembly runtime.

Our experiments compare the average execution time of each of the 4 pairs of programs and we present the results in item 5.1.1.

From the results, we can see that there is an important difference of relative performance between workloads. We can generalize the results in: if there is high context switch between the consumer and the library code, the execution time overhead reaches 100%, but on computationally heavy programs with low context switches, the Wasmtime JIT compiler can halve the execution times. We postulate that the JIT engine is capable of generating more efficient machine code due to the possibility of leveraging architecture-specific optimizations and due to the repetitive patterns in our programs. While we think that our programs represent extreme examples, our data shows that Wasmtime JIT compiler can beat native code for specific code patterns. This advantage contributes to mitigate the overhead that SFI imposes to a more complex and variegated program. Our test programs represent specific and isolated workloads that are generally mixed in regular applications. Nevertheless, we show that the performance overhead is dependent on workload and cannot be established beforehand. This result suggests that our approach is viable in practice and has to be evaluated by the developer for his specific workload on a case by case basis.

A significant aspect of the final performance for WebAssembly enabled libraries is the overhead added by the code introduced to invoke the functions correctly. If the function requires only numerical parameter passing, the overhead is small. On the other hand, if the function works on a data structure that is accessed both by the runtime and the host program, it is necessary to mirror every memory modification made inside the runtime. Both the identification of the modifications and their replication on the data structure on the host memory may lead to an important computational overhead.

In our evaluation presented above, we used the extraction strategy based on a table of functions as described in subsection 3.2.2. Afterwards, we repeated the tests by using the naive approach of extracting the functions pointer at every function invocation. Surprisingly, we found no statistical difference in performance between the two solutions for our test programs. We theorize that the lack of difference is due by two factors: the Wasmtime implementation of the function extraction is very efficient and the JIT compiler can recreate the table mechanism that we implemented manually.

5.1.2 Usability

We evaluate the usability of the libraries created with our procedure because usability is one of the goals of this project as we mentioned in section 3.1. Our usability evaluation has two points of views: the program developers who will link the library in its program and the library maintainer that has to create the new version that embeds a WebAssembly runtime.

As in subsection 5.1.1, we created two Shared Object libraries that a program developer can use in their programs. The resulting libraries are indistinguishable from the point of view of the user that aims to link them in their program. Both applications can be linked in the standard was using the `-lname`. For this reason, we argue that the usability of our solution for the end developer is identical to a normal standard library.

The second usability point of view is constituted by the library maintainer that has to create the new library. In subsection 3.2.2 we presented the steps necessary for creating a Shared Object with our design. The procedure has three steps:

1. compile the library source code to WebAssembly
2. write the code that replicates the API by invoking the runtime
3. compile to a Shared Object.

Compiling the library to WebAssembly is relatively easy if the developer knows the library requirements, its build system and its inner workings. We consider that the maintainer of the

library satisfies those requirements and therefore we argue that the compilation to WebAssembly step has a good usability. Afterwards, the maintainer has to create the "wrapper code" around the sandbox. This procedure is very repetitive and error prone for big libraries. For this reason, we developed the rewriter tool (see section 3.3) to automate a large part of this task. The "wrapper code" has to take into account various memory operations that may be needed for synchronizing the memory state of the host with the memory state inside the Sandbox. This procedure is complex and requires in-depth knowledge of the library functions and data structures. The code generated by the rewriter tool can not yet perform such heuristics and therefore the library maintainer has to add the corresponding code. Even with the development of the rewriter tool, we consider this step to be by far the most negative for usability. It is important to note that the code for encapsulating the library has to be done only once and incrementally adjusted when there is a change in the API. The last step of the porting procedure is to recompile the created files to a new Shared Object. The procedure for performing the last step is identical to the compilation of any other Shared Object and therefore is not a detriment to the usability. We conclude by stating that the usability for the library maintainer is medium as the second step of the porting procedure can be challenging for certain libraries.

Chapter 6

Discussion and Related Work

6.1 Discussion

Our approach for isolating native libraries proven to be viable in subsection 5.1.1. It is important to notice that the performance of the new created library, compared to its original counterpart, is highly dependent on the workload. At the same time, we have proven in subsection 5.1.2 that the developer can quickly create two copies of his program: one using the original Shared Object and one using the WebAssembly isolated one. This process is made possible because the linking process between the two libraries is identical. For this reason, the developer can benchmark its application using either of the two libraries and if the performance overhead (or gain, for specific workloads) is acceptable compared to the security benefits, they can chose to use the one isolated as WebAssembly code.

During our attempts at porting various libraries like `OpenSSL` or `Zlib`, we identified a major challenge in our approach: mirroring memory between the host and the sandbox. Some libraries expose internal data structures and allows the consumer programs to modify them between functions invocations. If this happens, the code wrapping the WebAssembly runtime, has to perform this memory mirroring before and after each function call. The procedure for mirroring memory is even more complex for complex and nested data structures, as the copy has to be done recursively or be triggered by multiple references to the same structure. This problem has been already highlighted in the literature, and it is considered as defiant without the knowledge of the underlying data structures [18]. Additionally, memory mirroring is a set of memory manipulations that are negative for performance and therefore pose an additional threat to usability in those situations. As there is no way to know only by the function signature, the writer of the "glue code" has to know the inner workings of the library and apply the memory mirroring correctly.

The memory mirroring issue exposes a limitation of the rewriter tool that it is not able to detect the situations where additional memory operations are required. The result is that the rewriter is very useful for libraries that do not present the memory mirroring problem and for writing the majority of the libraries that do. Yet, for the latter libraries, the contribution of the library maintainer is still needed.

6.2 Related Work

Existing projects address the isolation of native libraries using SFI different approaches, each with a design particularity. In this section, we will list the major ones and compare them against our project.

RLBox [23] aims to isolate native libraries inside the Firefox Web Browser using WebAssembly. Their approach leverages the existing WebAssembly engine of Firefox to isolate security sensitive libraries. RLBox is a framework that helps the Firefox programmer isolate libraries in source code. They display a modest performance overhead and they employ static information flow enforcement that contribute to mitigate vulnerabilities stemming from malicious input. Differently from our approach, their framework requires changes in the codebase of Firefox. An additional problem of their approach is that it is tailored for the Firefox codebase and supports only C++, for this reason we question the applicability of their method to different programs.

Gobi [24] presents a set of modifications to compilers and runtimes to create isolated native libraries. The compiler and runtime changes that Gobi performs need to be maintained in the long run, requiring constant effort that is very difficult to achieve in practice. While we think that Gobi approach was excellent in 2019, the support for WebAssembly of compilers, the creation of WASI and the rise of new efficient runtimes makes their solution outdated.

XFI [10] presents a mechanism for SFI for the Windows kernel that could be used instead of WebAssembly. XFI requires support from the kernel making it generally not viable for different platforms. XFI creates very high overhead (up to 800%) and it is a closed source research project that cannot be fully evaluated. Additionally, as specific compiler support is needed, none of the major languages support it and will need to adopt it to be a feasible solution.

NaCl [42] is a SFI solution designed by Google for the Chrome Web Browser. It is a custom runtime and compilation for applications intended to isolate plugins in the browser. The performance overhead compared to native applications ranges from 5% to 15% [31]. NaCl was available only for the Chrome Web Browser and has been deprecated in favor of WebAssembly due to the limited range of languages supported, the lack of support for system interface and the huge engineering requirement for supporting it.

Related work is limited, we attribute this to the fact that the field of using WebAssembly or SFI for isolating native applications has been explored only superficially. Additionally, WebAssembly is a very young and evolving technology that still needs to be researched to mature and express its potential.

Chapter 7

Future work

We think that this project is only a step towards commonly available isolated native libraries, therefore we propose the following possible innovations:

Improved performance evaluation During our project, we evaluated on programs that we wrote. Those programs are characterized by very specific and limited workloads to show the performance differences for different basic behaviors a library can have. Although we believe that our corpus choice correct, a performance evaluation on real world libraries (e.g., zlib, libpng, lipjpeg) and consumer programs would further support our solution's validity.

Strategy for memory mirroring As we have described in section 6.1, memory mirroring is a hard problem both for human programmers and automated tools. In future work, we would like to explore techniques for detecting and automating the memory mirroring requirement. This would benefit both the usability and the applicability of our solution to real world applications.

Improved rewriter The rewriter tool is capable of creating the template of the code needed for wrapping simple function calls but at the moment is not capable of detecting and inserting automatically none of the memory operations that might be required in certain functions invocations.

Chapter 8

Conclusion

In this project, we designed and evaluated a new technique for isolating native libraries using WebAssembly for SFI. Our approach isolate native libraries by creating a new shared object that exposes the same interface, but invokes his calls from inside a WebAssembly sandbox. Our technique create a Shared Object that can be used exactly in the same way as the original, therefore offering the possibility to use it without requiring any source code modification in the consumer application. The new libraries have a performance ranging from half to double the original native library counterpart. Additionally, we develop a rewriter tool that partially automates the process of transforming a native library to a WebAssembly isolated one. Finally, we present an evaluation of WebAssembly, WASI and their runtimes as SFI mechanisms. Our analysis shows that WebAssembly is an excellent candidate for SFI and that it is already usable with current runtimes. In particular, we showed that our approach is effective and can be a valid alternative for security sensitive applications compared to native libraries.

Bibliography

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. "Control-Flow Integrity Principles, Implementations, and Applications". In: *ACM Trans. Inf. Syst. Secur.* 13.1 (Nov. 2009). ISSN: 1094-9224. DOI: 10.1145/1609956.1609960. URL: https://doi.org/10.1145/1609956.1609960.

[2] Christopher Bogart, Christian Kästner, and James Herbsleb. "When It Breaks, It Breaks: How Ecosystem Developers Reason about the Stability of Dependencies". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. 2015, pp. 86–89. DOI: 10.1109/ASEW.2015.21.

[3] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. "Provably-Safe Multilingual Software Sandboxing using WebAssembly". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1975–1992. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya.

[4] Bytecode Alliance. *Bytecode Alliance*. URL: https://bytecodealliance.org (visited on 10/19/2022).

[5] Bytecode Alliance. *WaMR. WebAssembly Micro Runtime*. URL: https://github.com/bytecodealliance/wasm-micro-runtime (visited on 10/19/2022).

[6] Bytecode Alliance. *Wasmtime. A fast and secure runtime for WebAssembly*. URL: https://wasmtime.dev (visited on 10/19/2022).

[7] Lin Clark. *Standardizing WASI: A system interface to run WebAssembly outside the web*. URL: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/ (visited on 11/10/2022).

[8] Cloud Native Computing Foundation. *WasmEdge Runtime*. URL: https://wasmedge.org (visited on 10/19/2022).

[9] Emscripten contributors. *Emscripten Homepage*. URL: https://emscripten.org/index.html (visited on 10/18/2022).

[10] Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George C Necula. "XFI: Software guards for system address spaces". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 75–88.

[11] Free Software Foundation. *capabilities(7) Linux's User Manual*. Linux Man pages version 6.02. Apr. 2022.

[12] GHC Team. *GHC 9.6.1-alpha2 is now available*. URL: https://discourse.haskell.org/t/ghc-9-6-1-alpha2-is-now-available/5696 (visited on 01/30/2023).

[13] Google. *WebAssembly Migration Guide. (P)NaCl Deprecation Announcements*. URL: https://developer.chrome.com/docs/native-client/migration/ (visited on 01/25/2023).

[14] Joshua A. Kroll, Gordon Stewart, and Andrew W. Appel. "Portable Software Fault Isolation". In: *2014 IEEE 27th Computer Security Foundations Symposium*. 2014 IEEE 27th Computer Security Foundations Symposium (CSF). Vienna: IEEE, July 2014, pp. 18–32. ISBN: 978-1-4799-4290-9. DOI: 10.1109/CSF.2014.10. URL: http://ieeexplore.ieee.org/document/6957100/ (visited on 10/10/2022).

[15] Butler W. Lampson. "A Note on the Confinement Problem". In: *Commun. ACM* 16.10 (Oct. 1973), pp. 613–615. ISSN: 0001-0782. DOI: 10.1145/362375.362389. URL: https://doi.org/10.1145/362375.362389.

[16] Hugo Lefeuvre, Vlad-Andrei Buadoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. "Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software". In: *arXiv preprint arXiv:2212.12904* (2022).

[17] Daniel Lehmann, Johannes Kinder, and Michael Pradel. "Everything old is new again: Binary security of webassembly". In: *Proceedings of the 29th USENIX Conference on Security Symposium*. 2020, pp. 217–234.

[18] Shen Liu, Gang Tan, and Trent Jaeger. "PtrSplit: Supporting General Pointers in Automatic Program Partitioning". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2359–2371. ISBN: 9781450349468. DOI: 10.1145/3133956.3134066. URL: https://doi.org/10.1145/3133956.3134066.

[19] LLVM Foundation. *WebAssembly lld port. LLVM Documentation*. URL: https://lld.llvm.org/WebAssembly.html (visited on 11/28/2022).

[20] LLVM Project Contributors. *LLVM Project Supported Targets*. URL: https://github.com/llvm/llvm-project/tree/main/llvm/lib/Target/WebAssembly (visited on 01/07/2023).

[21] LLVM Project Contributors. *WebAssembly ("wasm") support*. URL: https://github.com/golang/go/issues/18892 (visited on 02/02/2023).

[22] Stephen McCamant and Greg Morrisett. "Evaluating SFI for a CISC Architecture." In: *USENIX Security Symposium*. Vol. 10. 2006, pp. 209–224.

[23] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. "Retrofitting fine grain isolation in the Firefox renderer". In: *Proceedings of the 29th USENIX Conference on Security Symposium*. 2020, pp. 699–716.

[24] Shravan Narayan, Tal Garfinkel, Sorin Lerner, Hovav Shacham, and Deian Stefan. "Gobi: WebAssembly as a practical path to library sandboxing". In: *arXiv preprint arXiv:1912.02285* (2019).

[25] Python Software Foundation. *The Python programming language*. URL: `https://www.python.org/` (visited on 01/13/2023).

[26] *Multi Memory Proposal for WebAssembly*. W3C, Oct. 26, 2022. URL: `https://github.com/WebAssembly/multi-memory`.

[27] Andreas Rossberg. *WebAssembly Core Specification*. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. W3C, Dec. 5, 2019. URL: `https://www.w3.org/TR/wasm-core-1/`.

[28] *Memory64 Proposal for WebAssembly*. W3C, Oct. 25, 2022. URL: `https://github.com/WebAssembly/memory64`.

[29] Jeremy Rubin, Dan Gohman, Nick Fitzgerald, and Till Schneidereit. *Wasmtime fails to run a .wasm built with rust-secp256k1, Wasmer can run it. Issue Number 2587, Wasmtime Github repository*. URL: `https://github.com/bytecodealliance/wasmtime/issues/2587` (visited on 12/07/2022).

[30] Rust Team. *WebAssembly*. URL: `https://www.rust-lang.org/what/wasm` (visited on 10/18/2022).

[31] David Sehr, Robert Muth, Cliff L. Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. "Adapting Software Fault Isolation to Contemporary CPU Architectures". In: *19th USENIX Security Symposium*. 2010, pp. 1–11. URL: `http://code.google.com/p/nativeclient/`.

[32] Benedikt Spies and Markus Mock. "An Evaluation of WebAssembly in Non-Web Environments". In: *2021 XLVII Latin American Computing Conference (CLEI)*. 2021, pp. 1–10. DOI: `10.1109/CLEI53233.2021.9640153`.

[33] Gang Tan et al. "Principles and implementation techniques of software-based fault isolation". In: *Foundations and Trends® in Privacy and Security* 1.3 (2017), pp. 137–198.

[34] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient Software-Based Fault Isolation". In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. SOSP '93. Asheville, North Carolina, USA: Association for Computing Machinery, 1993, pp. 203–216. ISBN: 0897916328. DOI: `10.1145/168619.168635`. URL: `https://doi.org/10.1145/168619.168635`.

[35] Wasmer, Inc. *Wasmer. Run any code on any client*. URL: `https://wasmer.io` (visited on 10/19/2022).

[36] WebAssembly Group. *binaryen. Optimizer and compiler toolchain library for WebAssembly.* URL: `https://github.com/WebAssembly/binaryen` (visited on 10/25/2022).

[37] WebAssembly Group. *WebAssembly.* URL: `https://webassembly.org/` (visited on 10/14/2022).

[38] WebAssembly Group. *WebAssembly C and C++ API.* URL: `https://github.com/WebAssembly/wasm-c-api` (visited on 12/12/2022).

[39] WebAssembly Group. *WebAssembly Roadmap.* URL: `https://webassembly.org/roadmap` (visited on 10/14/2022).

[40] World Wide Web Consortium. *WebAssembly Core Specification. W3C Recommendation, 5 December 2019.* URL: `https://www.w3.org/TR/wasm-core-1/` (visited on 02/02/2023).

[41] World Wide Web Consortium. *WebAssembly Core Specification. Version 2.0, W3C First Public Working Draft 19 April 2022.* URL: `https://www.w3.org/TR/wasm-core-2/` (visited on 02/02/2023).

[42] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. "Native client: A sandbox for portable, untrusted x86 native code". In: *Communications of the ACM* 53.1 (2010), pp. 91–99.

[43] Bin Zeng, Gang Tan, and Úlfar Erlingsson. "Strato: A Retargetable Framework for Low-level Inlined Reference Monitors". In: *Proceedings of the 22nd USENIX Conference on Security*. Berkeley, CA, USA, 2013, pp. 369–382.

[44] Bin Zeng, Gang Tan, and Greg Morrisett. "Combining control-flow integrity and static analysis for efficient and validated data sandboxing". In: *Proceedings of the 18th ACM conference on Computer and Communications Security*. 2011, pp. 29–40.

[45] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. "ARMor: fully verified software fault isolation". In: *Proceedings of the ninth ACM international conference on Embedded software*. ESWeek '11: Seventh Embedded Systems Week. Taipei Taiwan: ACM, Oct. 9, 2011, pp. 289–298. ISBN: 978-1-4503-0714-7. DOI: `10.1145/2038642.2038687`. URL: `https://dl.acm.org/doi/10.1145/2038642.2038687` (visited on 01/22/2023).