UNIVERSITÀ DEGLI STUDI DI BERGAMO

Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

Corso di Laurea Magistrale in Ingegneria Informatica

Classe LM-32

Enhancing WASI Sandboxes

Restricting WebAssembly Runtimes with Linux Security Modules

Relatore

Chiar.mo Prof. Stefano Paraboschi

Tesi di Laurea Magistrale

Michele Beretta

Matricola n. 1054365

ANNO ACCADEMICO 2021/2022

Abstract

Nowadays, web applications' needs are more sophisticated and demanding than in the past. Efficiency and security are hence important points that must be addressed. However JavaScript, the de facto standard of scripting languages on the web, is not able to meet these requirements.

In order to compensate for JavaScript's downsides, *WebAssembly* (WASM) was introduced. WASM is a new language that is "designed for efficient execution and compact representation of code on modern processors including in a web browser" [36], and strives to improve both performance and power consumption. Although WASM was conceived to run in browsers, it can also be run directly on the host with the aid of runtimes, such as *Wasmtime* and *Wasmer*, that do provide a fair level of security. In some use cases, however, the provided security level is not enough and solutions to enhance it are necessary.

The aim of this thesis work is to explore how these runtimes can be further restricted through the aid of Linux Security Modules so to improve security and give more control to the user.

Contents

1 Introduction **1**

1.1 The web . . . 1

1.1.1 Basic structure . . . 1

1.1.2 JavaScript . . . 1

1.1.3 Web browsers and JavaScript runtimes . . . 3

1.2 WebAssembly . . . 4

1.2.1 Description and motivations . . . 4

1.2.2 Overview of the language . . . 5

1.2.3 Problems with WebAssembly . . . 7

1.2.4 WASI – WebAssembly System Interface . . . 8

1.2.5 WebAssembly as a fault isolation tool . . . 11

1.3 Linux Security Modules . . . 12

1.3.1 Landlock . . . 12

1.3.2 eBPF . . . 15

2 Restricting the WASI sandbox with LSM **19**

2.1 Goals . . . 19

2.2 Restricting the WASI sandbox with Landlock . . . 20

2.2.1 Code architecture and description . . . 20

The `args` module . . . 22

The `wasm` module 25
The `landlock` module 28
The `path_access` module 29
2.2.2 Available permission settings 29
2.2.3 Advantages and disadvantages 31
2.3 Restricting the WASI sandbox with eBPF 32
2.3.1 Using *BPFContain* 32
2.3.2 Available permission settings 32
2.3.3 Advantages and disadvantages 36
2.4 Final product 37

3 Performance **39**

3.1 Testing plan 39
3.1.1 System and hardware 41
3.1.2 Test scenarios 41
3.1.3 Performance indicators 42
3.2 Comparison between different sandboxes 43
3.2.1 A computational-heavy program 44
3.2.2 Opening a file 46
3.2.3 Accessing the file system 47
3.3 Internal analysis of the various sandboxes 50
3.3.1 Execution times of the developed runtime 51
3.3.2 Landlock performance scaling 54
3.3.3 eBPF performance scaling 55
3.4 Conclusions and observations 58

4 Conclusions **61**

4.1 WebAssembly and LSM 61
4.2 The developed project 61
4.3 Possible improvements and future work 63

Acknowledgements **65**

Bibliography **67**

Chapter 1

Introduction

1.1 The web

1.1.1 Basic structure

The *World Wide Web*, or more commonly *web*, is a collection of documents and other resources that can be shared among various devices via the *Internet*.

The resources shared are usually in the form of *web pages*, documents whose structure is defined with HTML (*Hypertext Markup Language*) and whose styles are declared with CSS (*Cascade Style Sheet*). Other kinds of resources include anything ranging from images, videos, audio, software, and source code.

However, web pages are *static* if they only use HTML and CSS, and cannot do much aside from showing content. A programming language is hence needed to make the web *dynamic*. Among all technologies and languages, nowadays only JavaScript survives as the de facto standard language, used for both simple scripts and complex applications.

1.1.2 JavaScript

JavaScript, or *JS* when abbreviated, is a high-level, dynamic, weakly typed programming language that together with HTML and CSS is at the core of the World Wide Web. It conforms to the ECMAScript standard [8] and it can be both in-

terpreted and *just-in-time compiled*. A notable language which is a superset of JavaScript is *TypeScript*, developed by Microsoft, that introduces static typing and can be transpiled to JavaScript.

```
1  function fact(n) {
2    if (n == 0) return 1;
3    return n * fact(n - 1);
4  }
5
6  function showFactorial() {
7    let div = document.getElementById('output');
8    let input = document.getElementById('number');
9    let n = parseInt(input.text);
10
11   if (n < 0) {
12     div.innerHTML = 'Not defined';
13   } else {
14     div.innerHTML = fact(n);
15   }
16 }
17
18 document
19   .getElementById('get-factorial')
20   .addEventListener('click', showFactorial)
```

Listing 1.1: A simple example of JavaScript.

However, JavaScript is ill-equipped when it comes to performance-critical modern web pages – from these needs were born technologies such as *asm.js*, *Native Client*, and eventually *WebAssembly*. The main downsides of JavaScript that brought the need for WebAssembly are:

- JavaScript is *always* represented as a text format, so it must be parsed by the user's browser for it to be compiled and enhanced;
- JavaScript is dynamically typed, hence optimisations for JS (when possible) have to occur in the browser, when all JS files have been received; for example, JS has only two numeric types (a double-precision IEEE 754 value and a *BigInt*

for numbers bigger than 2^{53}) and this restricts the usable CPU instructions;

- JavaScript has a number of security vulnerabilities that WebAssembly strives to eliminate by design; a common security vulnerability[1] in the JavaScript world is *cross-site scripting* (XSS), which is a violation of the same-origin policy and occurs when an attacker is able to inject a malicious script in a target website; another vulnerability is *cross-site request forgery* (CSRF), in which malicious code on an attacker's website tricks the user's browser, so that it takes actions not desired by the user.

1.1.3 Web browsers and JavaScript runtimes

A *web browser* is a piece of software for accessing web pages shared on the World Wide Web, or a local website. Browsers natively support the rendering of HTML and CSS, and feature an engine able to run JavaScript or, more commonly, to just-in-time compile it in order to improve performance.

Browsers can also mitigate the risk provided by the common JavaScript vulnerabilities mentioned in Section 1.1.2 by using two main methods – *sandboxing*, so that JS isn't allowed to perform general-purpose programming tasks like working directly with files[2], and *same-origin policy* constraints, in which scripts from one website have access to data from the same website but not from others.

JavaScript can not only be executed in a browser, but also on so called "back-end" runtimes[3]. These runtimes offer custom libraries that permit JavaScript to access the file system, to make network requests, spawn child processes and build complex applications, such as web servers. Two notable open-source runtimes are *Node.js*[4] and *Deno*[5], both built on top of Google's V8 engine. The first is older, initially released in 2009, and has a package manager called *npm* in order to manage its

[1] Software vulnerabilities are the most common, but hardware vulnerabilities have also been found [16].

[2] Although, there are APIs that can grant access to a "virtual drive" only when explicitly allowed by the user [10], so direct access to the user's file system is not possible.

[3] Opposed to "front-end", which usually pertains to the browser.

[4] `https://nodejs.org/`

[5] `https://deno.land`

vast ecosystem of JS packages. Deno, on the other hand, is a more modern runtime written in Rust, includes support for TypeScript out of the box and is *secure by default* since it blocks all file, network, and environment access unless explicitly enabled.

1.2 WebAssembly

1.2.1 Description and motivations

WebAssembly [33] is a binary instruction format for a stack-based virtual machine. It is designed to be a portable compilation target, so that different languages can be deployed on the web. It was announced in 2015 and implemented by major browsers by 2017, and as of May 2022 is supported by 93% of all browsers [1, 34]. WebAssembly main goals are the following [13]:

- *security*, since code on the web originates from untrusted sources;
- *speed*, by using ahead-of-time optimisations in a similar manner as native machine code;
- *portability*, since the web spans many devices, architectures, operating systems, and browsers;
- and finally *compactness*, because code is transmitted over the network and must reduce load times as much as possible.

There were previous attempts at solving the problem of having safe, fast, and portable low-level code on the Web, such as *ActiveX*, *Native Client*, and *asm.js*.

ActiveX [2] was a Microsoft's technology for code-signing x86 binaries to run on the web, and relied only on this signing. Hence, it achieved security through a trust model instead of technical construction. *Native Client* [23] introduced the first sandboxing technique for x86, ARM or MIPS machine code, which was statically validated. Lastly, *asm.js* [5] is a specialised subset of JavaScript, which is one of the

target languages of *Emscripten* [9], a compiler toolchain able to compile C/C++ applications in order to have them run on browsers or on other JavaScript runtimes.

WebAssembly is available as a target for various languages, such as C/C++ with the aid of Emscripten, Rust, AssemblyScript[6], Go, Kotlin, Swift, and Zig. The compiled binary can be then used from JavaScript on the web, or with Node.js or Deno, or even as a CLI application with the aid of *WebAssembly runtimes* and the *WebAssembly System Interface* for accessing system resources (see Section 1.2.4).

WebAssembly resolves many of the problems of JavaScript:

- being represented with a *binary* format, parsing on the client is faster and simpler;
- having a static type systems, optimisations can be done earlier at compile time, and not on the browser once all the files have already been transferred.

```
1  (module
2    (func $fact (param $x i64) (result i64)
3      (if (result i64) (i64.eqz (local.get $x))
4        (then (i64.const 1))
5        (else
6          (i64.mul
7            (local.get $x)
8            (call $fact
9              (i64.sub (local.get $x) (i64.const 1)))))))
10   (export "fact" (func $fact)))
```

Listing 1.2: Recursive factorial written in WebAssembly S-expressions.

1.2.2 Overview of the language

Although WebAssembly is a binary code format, it can also be written with *S-expressions* in order to be more readable, as seen in Listing 1.2. Each binary takes the form of a *module*, which contains *functions*, *global* variables, *tables*, and *memories*. Each one of these can be exported to be used in the embedder, and a module can

[6] A language with a syntax similar to TypeScript.

also import functionalities. While a module is a static representation of a program, an *instance* is the dynamic one. A module can be instantiated by the embedder (e.g., a JavaScript virtual machine).

WebAssembly is a typed language, but there are only four basic *value types* – integers and IEEE 754 floating point numbers, each in 32 and 64 bit variants. There are no distinctions between signed and unsigned integers, but instructions that depend on the presence of the sign are marked with an explicit suffix.

Functions are typed, take a sequence of values as a parameter, and then return another sequence of values. They cannot be nested inside each other. The contents of the call stack for execution are separate from the data portion of the memory, and cannot be accessed directly by WebAssembly. The code inside a function consists of a series of *instructions* that modify an implicit operand stack, either by declaring local variables or applying operations to the values already on the stack. Functions can be called either *directly* by using an index that identifies a function, or *indirectly*, i.e., dynamically through a global *table* as shown in Figure 1.1. In the second case, only the function's type signature is validated.

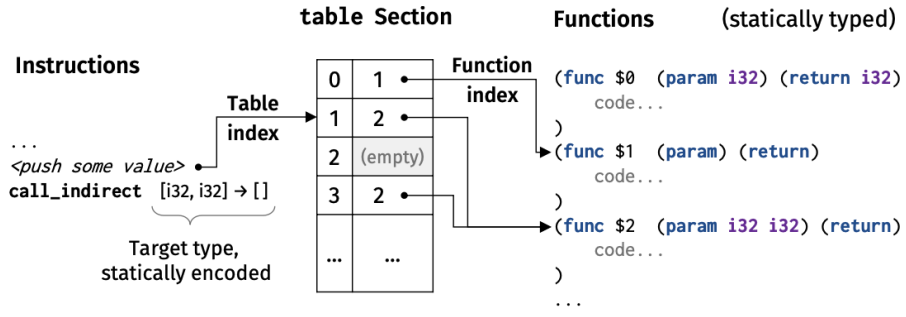


Figure 1.1: The structure of the indirect functions table [18].

WebAssembly has also support for *traps* – in a similar manner to an exception, a trap aborts the current computation, and control is given back to the embedder. For example, when embedded in JavaScript, a trap will throw a JavaScript exception.

The main memory layout that WebAssembly uses is a *linear memory* (or simply *memory*) represented by an array of bytes. A *linear memory* is a simple addressing technique in which memory appears to the program as a single contiguous access space [12], which the CPU can access both directly as well as linearly without re-

sorting to memory segmentation or pagination schemes, enhancing flexibility and reducing latency. Each module can define no more than one *memory*, a space which can grow by one or more *pages*[7] if the need arises. Access to the individual locations is done through addresses, represented as unsigned integers, which are dynamically checked against memory size[8] (out of bounds accesses result in a trap). Linear memory also brings security benefits – it is disjoint from the code space, the execution stack and the engine's data structures [13]; hence, compiled programs cannot directly corrupt their own execution environment or perform undefined behaviour.

Lastly, WebAssembly doesn't provide arbitrary jumps but *structured control flow*, so that the code can be validated in a single pass and common control flow attacks can be prevented.

1.2.3 Problems with WebAssembly

Since languages with manual memory management, such as C/C++, can be compiled into WebAssembly, it is reasonable to ponder how memory vulnerabilities affect WebAssembly binaries. In the original paper it is mentioned that "a buggy or exploited WebAssembly program can make a mess of the data in its own memory" [13], in light of the fact that the call stack is not accessible by the program.

However, even though WebAssembly strives for security, binaries can be still exploited with both traditional attacks, such as buffer overflows, and attacks that aren't applicable in traditional native binaries, e.g., overwriting string literals in memory [18].

A paper by Lehmann, Kinder, and Pradel [18] shows that it is possible to obtain a write primitive given a WebAssembly binary compiled from vulnerable C/C++ code, by means of stack-based buffer overflows and heap metadata corruption, and it is possible to overwrite stack, heap, and "constant" data, since the linear memory does not have a read-only section. It would be possible then, for example, to overwrite the name of a file that was encoded as a string literal in the original program.

[7] A page is 64 KiB large.

[8] On 64 bit platforms the WebAssembly engine can make use of virtual memory techniques to eliminate these checks, see [13].

Another problem analysed in the same paper is the redirection of indirect calls. Since WebAssembly allows indirect calls to functions through a function table, an attacker could divert the execution by overwriting an integer in linear memory that serves as an index into the table section. WebAssembly limits this problem with two mechanisms – not all functions may appear in the table, but only those that can be indirectly called, and all functions call are type checked. So, redirecting indirect calls is possible only within the class of functions with the same type.

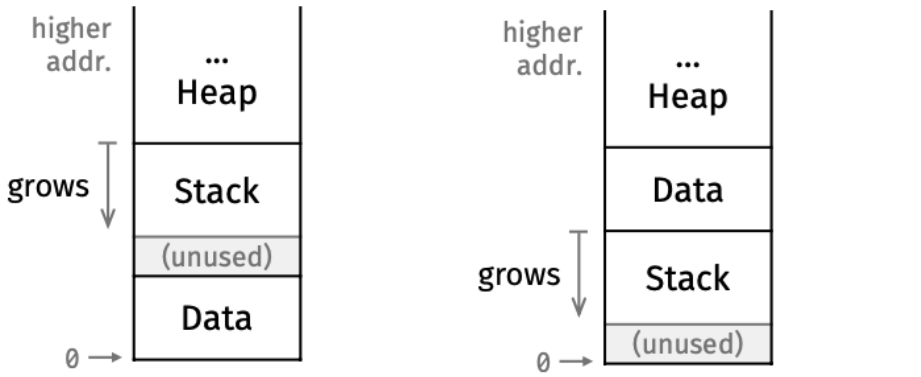
Lastly, it is possible to enable remote code execution when including vulnerable WebAssembly in an application. This is because functions that have different types in one language can be mapped onto functions with the same type in WebAssembly. For example, a log function that in C has the signature `void log(int)`, and a function such as `void exec(const char* cmd)` both become functions with only one `i32` parameter in WebAssembly. This enables the indirect diversion described in the previous paragraph if the two functions can be indirectly called.

1.2.4 WASI – WebAssembly System Interface

The *WebAssembly System Interface* (WASI) [28] is a modular system interface for WebAssembly. It focuses on security and portability so that WebAssembly binaries can be targeted by different languages and then be safely run on different platforms. Its API provides access to several OS-like features, such as file systems and Berkeley sockets. The two languages that have good interoperability at the moment are Rust, where the compiler directly supports targeting both WASM and WASI, and C/C++, either through Emscripten or a custom prebuilt *Clang* toolchain[9]. These compilers can produce very different memory layouts, as shown in Figure 1.2.

A binary that uses WASI can be run with a CLI runtime, such as *Wasmtime* [32] or *Wasmer* [31], or through a browser polyfill (an example is visible in [29]). The approach taken for the sandboxing by these runtimes is based on a *capability-based security model* [25], so that access to the file system and other resources must be explicitly given. Capabilities can be either *static*, which are represented by the

[9] `https://github.com/WebAssembly/wasi-sdk`



(a) Emscripten 1.39.7, Clang 9 (without `stack-first`).

(b) Clang 9 (with `stack-first`), Rust 1.41 (WASI target).

Figure 1.2: Examples of WebAssembly memory layouts obtained from different compilers [18].

list of imports of the WebAssembly module to run, or *dynamic*, i.e., flags specified with custom command-line arguments at execution time. Moreover, when given permissions to write to a terminal, a program could print characters recognised as control sequences that may have side effects and confuse or mislead users (a harmless example is clearing the screen). Hence, all writes to output streams are filtered in order to prevent this type of behaviour.

CLI runtimes have some particular features and properties:

- it is possible to *sandbox* entire directories by *preopening* them, and possibly mapping them to custom paths, in order to have the file descriptors available; it is not possible to escape this sandbox using the parent directory `..` or soft links, but it is possible to escape using hard links; this is because hard links point directly to the *data* on disk, and behave as a full fledged copy of the linked file, while soft links only point to an existing *file* – this is why using hard links makes it possible to "escape" the sandbox;

- the *directory* is the finest granularity available to the preopen action – once access is given to a specific directory, all files and subdirectories are visible, and it is not possible to restrict permissions (all files are readable and writable)

when using the command line runtime wrapper[10];

- when using C on Linux, the WebAssembly binary can create files with specific user permissions, e.g., executable files;
- when listing files in a directory, the only files that are visible are the ones that were already present at the preopening of the directory, but files added after the preopen can be "seen" if their name is known (i.e., they can be opened, read or written in accordance with given permissions);
- any change made to a file by any process is reflected in what the compiled WebAssembly binary is able to read;
- environment variables are not accessible by default, but they must be explicitly declared and given access to;
- the WebAssembly memory is effectively sandboxed – the compiled program cannot access memory outside its sandbox, otherwise a trap is raised;
- not all "classic" functionality is available – when compiling C with the custom prebuilt toolchain, functions such as `system`, `execv`, and `fexecve` are not available, so it is not possible to execute arbitrary commands or files;
- fileless execution isn't feasible, since `memfd_create` is not available due to the missing `sys/mman.h`[11].

By using the *wasm2c* tool available in the *WebAssembly Binary Toolkit*[12], C code can be compiled into WebAssembly and then converted back to C. This way, a traditional C program can call the functions made available from the original C library with an added sandbox that allows code isolation. The sandboxed library cannot access locations outside its allocated memory without causing a trap, halting the execution, and giving back control to the embedder. Similarly, the user C code

[10]It is possible to specify a basic set of permissions when using the WASI runtime library in another language, such as in Rust, see example at [30].

[11]This library can be emulated, but `memfd_create` is still not available.

[12]`https://github.com/WebAssembly/wabt`

cannot access the sandboxed memory directly, and, if the code were to attempt to do so, it would result in a segmentation fault.

Lastly, as highlighted in [25], Spectre mitigations are not yet implemented, but are a topic of ongoing research.

1.2.5 WebAssembly as a fault isolation tool

Since WebAssembly is *sandboxed*, it is possible to isolate libraries that could be a frequent source of vulnerabilities by compiling them to WebAssembly before being used.

An example is shown with the *RLBox* project [22, 24], a framework that supports efficient sandboxing with modest performance overhead, that has been integrated into Firefox in order to restrict the `libGraphite` font shaping library. RLBox ensures that whatever library is sandboxed is also memory isolated from the rest of the application, and all the boundary crossings are explicit. The isolation is enforced by specific mechanisms, such as WebAssembly [14], and to ensure that application code doesn't use unsafe values originated in the sandbox, RLBox differentiates between *tainted* and *untainted* values through the use of the C++ type system.

Even the WASI interface is able to segregate a WebAssembly binary, since it is effectively sandboxed – all interactions with the embedding environment must be done through explicit exports and imports, and the memory is bound checked at runtime. Hence, the WebAssembly program cannot access data outside its assigned memory, and the embedder cannot access memory assigned to the WebAssembly code.

1.3 Linux Security Modules

The *Linux Security Modules* (LSM) [37, 19] is a lightweight, general purpose, access control framework for the Linux Kernel. It provides a mechanism for various security checks to be hooked by kernel extensions. These extensions are not loadable kernel modules, but they can either be chosen at compile-time via specific flags, such as `CONFIG_DEFAULT_SECURITY`, or overridden at boot-time.

The LSM is used primarily by *Mandatory Access Control* (MAC) extensions to provide a security policy. However, other extensions can be built with the LSM framework in order to implement specific changes when they cannot be obtained with the functionality within Linux itself.

Some projects that use LSM include:

- *SELinux* [26], i.e., *Security Enhanced Linux*, that provides a mechanism for supporting advanced and fine-grained access control policies, as well as MAC;
- *Smack* [27], a kernel based implementation of MAC with simplicity as one of its primary goals;
- *AppArmor* [3], a MAC style security extension that implements a task centred policy.

1.3.1 Landlock

Landlock [20, 21] is a security feature available since Linux 5.13 that utilises the LSM framework in order to provide scoped access control so that any process, even when unprivileged, can securely restrict itself. This can help mitigate the security impact of bugs or unexpected/malicious behaviour in user space applications.

Landlock employs the concept of *rule*, which describes an action on an object. An object is (currently) a file hierarchy, and actions are defined with access rights, such as executing, reading or writing files, making symbolic links and so on. A set of rules is called a *ruleset*, and it can restrict both the thread using it and its future

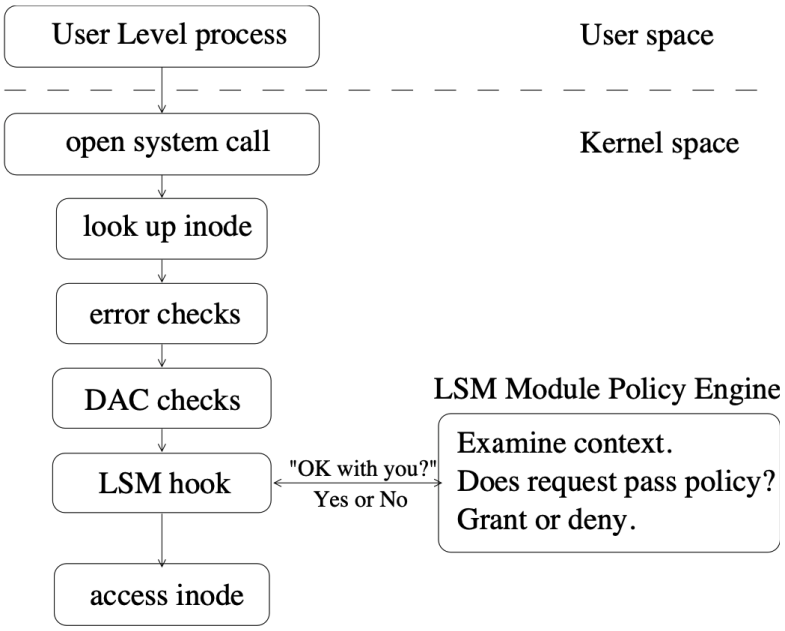


Figure 1.3: LSM Hook Architecture [37].

children, created either by spawning a new thread, as well as using the *fork* system call.

Notably, Landlock does not permit the definition of exceptions. For example, let us suppose we have a directory `dir1`, which contains two files `file1` and `file2`. We can define a ruleset that allows reading and writing files for `dir1`. However, if we then define another ruleset comprising only of read operations for `file1`, the permissions specified for `dir1` are still valid, so when a process restricts itself it is still able to write to `file1`. Other limitations of Landlock include the impossibility for a thread to modify its own topology (e.g., via *mount*), a limit of 16 layers of stacked rulesets and the inability to restrict special file systems, such as kernel file systems (e.g., nsfs).

In case of multiple consecutive self-restrictions, the result is the intersection of all rulesets – if a process first restricts itself allowing all read and writing operations, and then restricts itself again with only reading permissions, the result is equivalent to a single restriction made with a ruleset that permits only reading operations.

Landlock can be used directly when writing C/C++ code in Linux with the

<linux/landlock.h>[13] header, or by using libraries that provide these bindings to other languages, such as `go-landlock` [11] for Go and `rust-landlock` [17] for Rust.

Landlock employs a series of flags to restrict a sandboxed process to a set of actions on files and directories opened after the sandboxing operation (those opened before are not affected). As highlighted in [21], the file system flags for files are these three:

- `LANDLOCK_ACCESS_FS_EXECUTE`, to execute a file;
- `LANDLOCK_ACCESS_FS_WRITE_FILE`, to open a file with write access;
- `LANDLOCK_ACCESS_FS_READ_FILE`, to open a file with read access.

Directories can receive a wider range of access rights, including the previous ones and the following ones. All of them, except the first, are applied to the content of a directory and not to the directory itself[14].

- `LANDLOCK_ACCESS_FS_READ_DIR`, to open a directory or list its content[15];
- `LANDLOCK_ACCESS_FS_REMOVE_DIR`, to remove an empty directory or rename one;
- `LANDLOCK_ACCESS_FS_REMOVE_FILE`, to unlink or rename a file;
- `LANDLOCK_ACCESS_FS_MAKE_CHAR`, to create, rename or link a character device;
- `LANDLOCK_ACCESS_FS_MAKE_DIR`, to create or rename a directory;
- `LANDLOCK_ACCESS_FS_MAKE_REG`, to create, rename or link a regular file;
- `LANDLOCK_ACCESS_FS_MAKE_SOCK`, to create, rename or link a socket;

[13] Examples of code available at [21].

[14] This means that, for example, if `LANDLOCK_ACCESS_FS_REMOVE_DIR` is given to a directory d, it allows the removal of empty directories *inside* of d, and not of d itself.

[15] Applies to subdirectories as well.

- `LANDLOCK_ACCESS_FS_MAKE_FIFO`, to create, rename or link a named pipe;
- `LANDLOCK_ACCESS_FS_MAKE_BLOCK`, to create, rename or link a block device;
- `LANDLOCK_ACCESS_FS_MAKE_SYM`, to create, rename or link a symbolic link;
- `LANDLOCK_ACCESS_FS_REFER`, to re-parent a file hierarchy[16].

This last one is treated differently from the others – in order to prevent privilege escalation, it is necessary that the destination directory hierarchy has the same (or a superset of) restrictions as the source hierarchy, since only creating a rule with such access right is not enough to protect oneself. If that is not the case, this action is denied directly by the operating system.

The behaviour of symbolic links within Landlock is also interesting – hard links can be created only on data which can already be accessed according to the provided rules, while newly created soft links can reference any path on the file system, but the rules that are applied are those of the referenced directory. For example, if a program has a `LANDLOCK_ACCESS_FS_READ_FILE` as well as a `LANDLOCK_ACCESS_FS_MAKE_SYM` on a certain directory in the home of a user, then a hard link pointing to `/etc/passwd` cannot be created, since the program does not have access to it, while a soft link can be created but cannot be read.

Lastly, every new thread resulting from a *clone* call inherits all Landlock restrictions from its parent. These restrictions are also kept after a *fork*. More specifically, when a thread restricts itself, these rules are not applied to other sibling threads but will be enforced on all of the thread's descendants.

1.3.2 eBPF

eBPF [7] is a virtual machine in the Linux Kernel that can run sandboxed programs in a privileged context. It can be used to safely extend the capabilities of the kernel

[16] Available since the second version of the Landlock ABI.

without requiring to change kernel source code or load kernel modules. A generic overview of its structure is described in Figure 1.4.

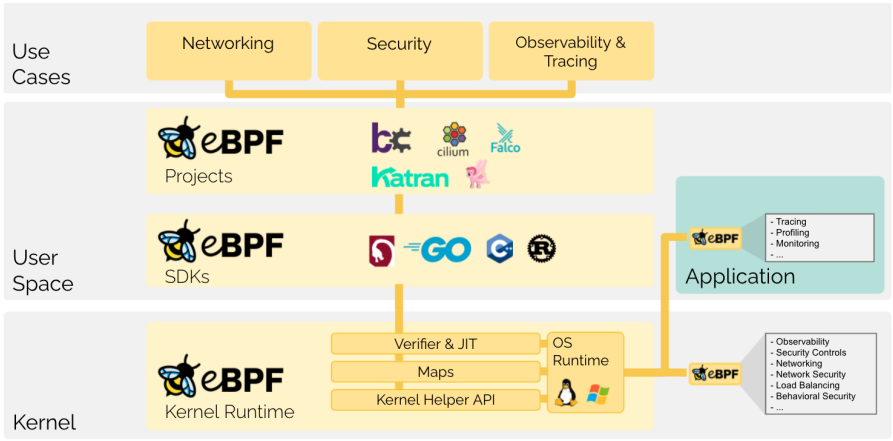


Figure 1.4: A generic overview of eBPF [7].

Some of the most common use cases for eBPF include the implementation of networking, security, and observability of programs. These are usually implemented in the operating system because of the kernel's privileges, but the kernel itself is hard to evolve. On the other hand, application developers can run eBPF programs in order to add additional capabilities at runtime, and then the operating system guarantees security and efficiency as if natively compiled by using a Just-In-Time compiler.

These eBPF programs are event-driven and run when a certain hook point is passed. Some predefined hooks include system calls, network events, and so on. If a hook does not exist, it is possible to create custom kernel probes or user probes. An example of how eBPF interacts with the Linux Kernel is visible in Figure 1.5.

When running a program with eBPF, there are many steps that are taken:

1. first of all, eBPF is usually not used directly but via libraries that provide the ability to specify intent-based operations which are then implemented with eBPF; in any case, an eBPF program must be compiled into a specific eBPF bytecode;

2. when a certain *hook* is invoked, this hook has to be identified and then the program can be loaded into the kernel with a system call;

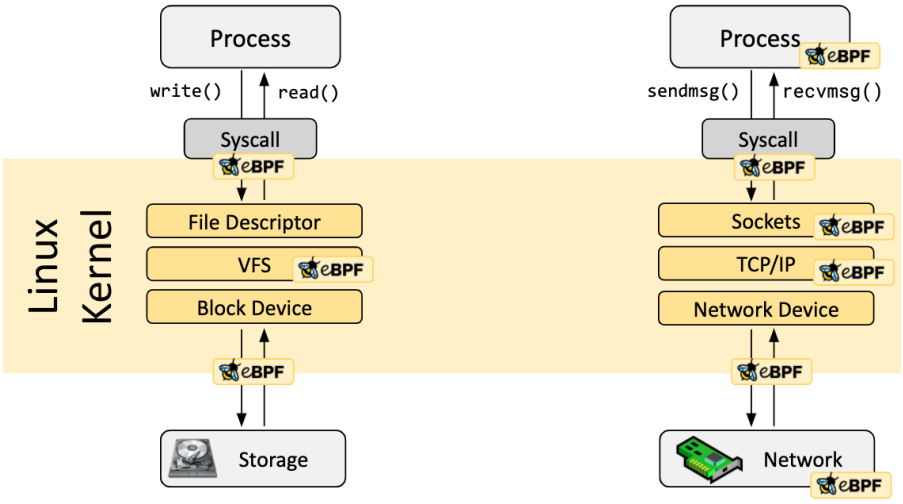


Figure 1.5: An example of two eBPF hooks [35].

3. a *verification* step ensures that the eBPF program is safe to run by validating a set of conditions (e.g., the program has the required capabilities, the program always runs to completion, etc.);

4. finally, the generic bytecode is then Just-In-Time *compiled* into the machine instructions to run the program as efficiently as possible.

Unlike Landlock, eBPF makes it possible to define exceptions, e.g., denying certain access rights on certain files while allowing them on the containing directory. It also allows to specify whether a program has access to devices such as `terminal`. Thanks to `BPFContain` [6] it is also possible to run a container security daemon by leveraging eBPF. In this case, specific policies must be written for the desired commands.

Chapter 2

Restricting the WASI sandbox with LSM

2.1 Goals

The main goal of using *Linux Security Modules* in combination with WebAssembly is to provide a fine grain customisation when choosing how to limit the permissions given to the executable. More specifically, the desired features are:

1. to give the possibility to differentiate between directories and files when giving permissions;
2. to clearly separate different capabilities;
3. to extend the possible set of permissions under the user's control.

Additional desirable features include the possibility to apply exceptions to a certain rule, as well as limiting the access to other resources in the operating systems, such as network, devices, and inter-process communications.

Finally, usability is also taken into consideration, especially from a user's point of view.

Note that the main purpose of these tests is not to replicate all the functionalities given by the WebAssembly runtimes (*Wasmtime* and *Wasmer*), but to see how their basic features, specifically running a WebAssembly compiled binary through WASI,

can be integrated and/or extended with security tools from the Linux Kernel. This means that, for example, running a single function exported from a WebAssembly module won't be taken into consideration.

2.2 Restricting the WASI sandbox with Landlock

For the integration with Landlock, a new project was developed, in order to provide a custom command line interface that allowed the user to specify with more precision what the WebAssembly binary is permitted to do on certain folders and files.

The project is open source and the code is available at `https://github.com/micheleberetta98/rust-wasm-landlock`. In order for the project to compile, the Landlock LSM must be enabled in the Linux Kernel.

2.2.1 Code architecture and description

The code itself is written in *Rust*, and makes use of the *rust-landlock* [17] crate to communicate with Landlock and the *wasmtime* and *wasmtime-wasi* official crates in order to have a runtime environment to execute a WebAssembly binary.

The project is divided into five modules, which are the following:

1. `args`, used to define and parse the command line arguments that specify the WebAssembly binary, the directories to preopen, and the allowed privileges on individual folders and/or files;
2. `landlock`, which handles the creation and update of the permissions' ruleset;
3. `main`, the module that interfaces with the user;
4. `path_access`, containing a helper structure to track permissions for a single path;
5. and finally `wasm`, that communicates with the *wasmtime-wasi* and the *wasmtime* crates in order to run the provided binary, as well as preopening directories.

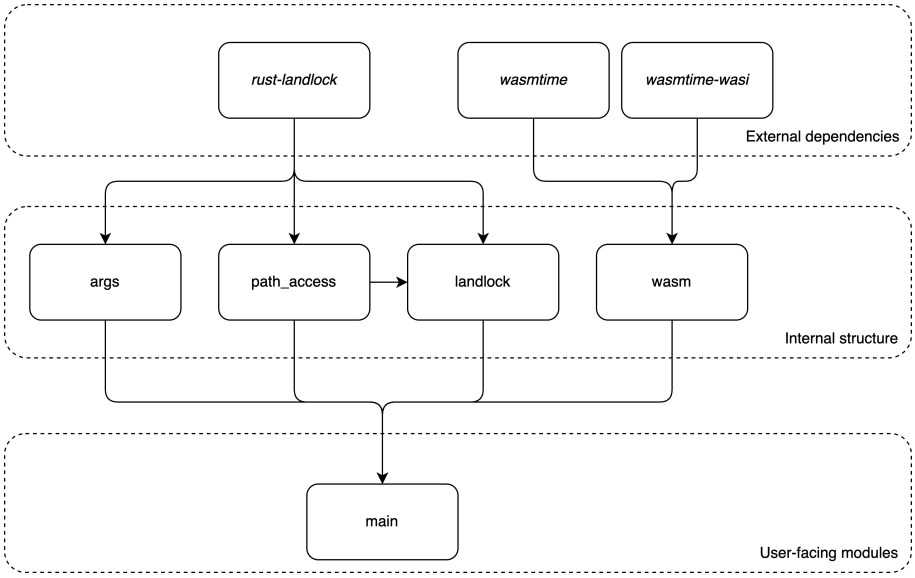


Figure 2.1: The code's architecture.

```rust
pub struct Args {
  // The module to execute
  pub wasm_module: String,

  // The preopened dir(s) to pass to wasmtime
  pub dirs: Vec<String>,

  // The preopened mapped dir(s) to pass to wasmtime
  pub mapdirs: Vec<(String, String)>,

  // A list of the allowed privileges on a
  // particular folder/file
  pub fs_allows: Vec<(String, BitFlags<AccessFs>)>,

  // Disable landlock (test only)
  pub no_landlock: bool,
}
```

Listing 2.1: The `Args` struct.

Argument	*Required*	*Description*
WASM module	Yes	The path to the WASM binary to run
`--dir`	No	All the directories to preopen
`--mapdir`	No	Eventual mappings between the directories
`--fs-allow`	No	All the permitted actions on a single path
`--no-landlock`	No	Disable the self restriction done by Landlock

Table 2.1: All the available command line arguments.

The architecture diagram is shown in Figure 2.1, where are outlined the most important external crates and all dependencies between the modules, represented by directed arrows.

The `args` module

The main definition of this module is the `Args` struct, visible in Listing 2.1, used to represent the command line interface that allows to specify which permissions to enable for one or more specific paths.

Every single field is mapped to one command line argument thanks to the *clap* crate which handles the creation and parsing of the CLI arguments structure. Their usage and meaning is described in Table 2.1.

The first argument is positional, and it is the path where the WebAssembly module is located.

The `dir` and `mapdir` arguments can appear multiple times, and they are used to list the directories to preopen so that the WebAssembly module can access them and their contents. When mapping a directory, in a way similar to the one provided by the *Wasmtime* command line tool, one can make a directory appear as "having a different path" when accessed by the WebAssembly module.

```rust
fn parse_flag_str(s: &str)
    -> Result<(String, BitFlags<AccessFs>)> {
  let parts: Vec<_> = s.splitn(2, ':').collect();
  if parts.len() != 2 {
    bail!("must be in the form 'PATH:FLAGS'");
  }

  let path = parts[0].to_string();
  let args = parts[1].split(',');

  let mut flags: BitFlags<AccessFs> = BitFlags::EMPTY;
  for arg in args {
    let (head, tail) = arg.split_at(1);
    if head == "-" {
      let f = parse_flag(tail)?;
      flags &= !f;
    } else {
      let f = parse_flag(s)?;
      flags |= f;
    }
  }

  Ok((path, flags))
}
```

Listing 2.2: Parsing the `fs-allow` command line option.

The `fs-allow` argument is made up of two parts, separated by a colon – a path to apply the restrictions to, and a series of enabled permissions on that specific path, separated by a comma. Each permission is mapped one-to-one to the flags provided by Landlock in a manner listed in Table 2.2. Moreover, there are three useful "shortcuts" used to represent common situations – "`read`", "`write`" and "`*`", for enabling the common reading operations, write operations and all operations.

```rust
fn parse_flag(s: &str) -> Result<BitFlags<AccessFs>> {
  match s {
    "*" => Ok(BitFlags::all()),
    "read" => Ok(ACCESS_FS_ROUGHLY_READ),
    "write" => Ok(ACCESS_FS_ROUGHLY_WRITE),
    _ => parse_single_flag(s).map(BitFlags::from),
  }
}

fn parse_single_flag(s: &str) -> Result<AccessFs> {
  let f = match s {
    "X" => Execute,
    "W" => WriteFile,
    "R" => ReadFile,
    "RDir" => ReadDir,
    "DDir" => RemoveDir,
    "D" => RemoveFile,
    "MChar" => MakeChar,
    "MDir" => MakeDir,
    "MReg" => MakeReg,
    "MSock" => MakeSock,
    "MFifo" => MakeFifo,
    "MBlock" => MakeBlock,
    "MSym" => MakeSym,
    _ => bail!(format!("invalid flag provided: {}", s)),
  };

  Ok(f)
}
```

Listing 2.3: Parsing a single Landlock flag.

Another utility provided is the ability to specify *exceptions* by prepending a single flag with "–" (a dash). For example, to specify that a program has all access rights on a folder *except* the execution one, then it is quicker to write "`*,-X`" instead of spelling out all the enabled flags. A full example of how these arguments work is explained in Figure 2.2, while the code to parse the `fs-allow` command line

argument is displayed in Listings 2.2 and 2.3.

Lastly, the `no-landlock` argument is only for testing purposes – it is disabled by default, so that Landlock is enabled and self restriction is applied unless explicitly stated otherwise.

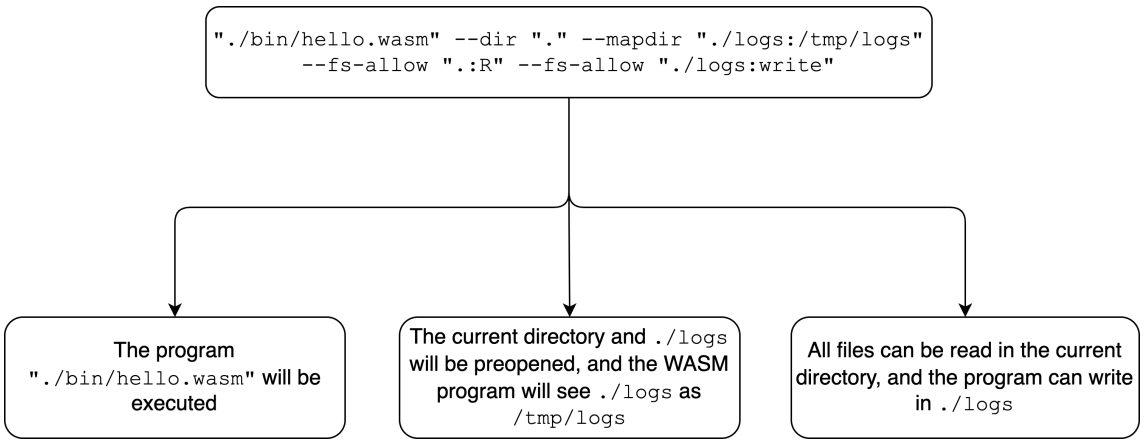


Figure 2.2: A full example of how the arguments are seen by the program.

The `wasm` module

This module manages a single WebAssembly binary module and forms a bridge to the *wasmtime* and the *wasmtime-wasi* crates. In this project, *Wasmtime* bindings are used instead of the *Wasmer* ones, but both provide the same level of functionality, albeit with a different interface. A high level view of the usual sequence in which the steps would be done when loading and running a module is shown in Figure 2.3. An outline of the module is visible in Listing 2.4, while the specific code required to run a WASM binary is described in Listing 2.5.

The *WasmModule* structure represents a WASM binary as a vector of bytes, and it also handles the creation and construction of a *WASI Context*, another structure defined by the *wasmtime-wasi* crate used to store all preopened directories, potential imports, and more data that will be useful to run the WebAssembly module.

Moreover, this module defines various helpers to preopen directories, both unmapped and mapped, and also defines how to run a WebAssembly binary – it must create both an *engine* and a *linker*, as well as a *store*, which represents the memory

of a WebAssembly module described in the previous chapter.

Note that the WebAssembly module must have a *default exported function* to be run – when compiling with a WASI target, this is usually represented by the `_start` function, which corresponds to the `main` function in languages such as C and Rust.

```rust
pub struct WasmModule {
  bytes: Vec<u8>, ctx_builder: WasiCtxBuilder,
}

impl WasmModule {
  // Reads the WASM module, initialises WasiCtxBuilder
  pub fn new(path: &str) -> Result<Self> {...}

  // Make the WASM module inherit stdio
  pub fn use_stdio(mut self) -> Self {...}

  // Preopen all given directories
  pub fn preopen_all(
    mut self,
    dirs: &Vec<String>) -> Result<Self> {...}

  // Preopen and map all given directories
  pub fn preopen_all_map(
    mut self,
    mapdirs: &Vec<(String, String)>) -> Result<Self> {...}

  // Preopen (and map) a single directory
  pub fn preopen(
    mut self,
    dir: &str, guest_path: &str) -> Result<Self> {...}

  // Run the module
  pub fn run(self) -> Result<()> {...}
}
```

Listing 2.4: The outline of the `wasm` module.

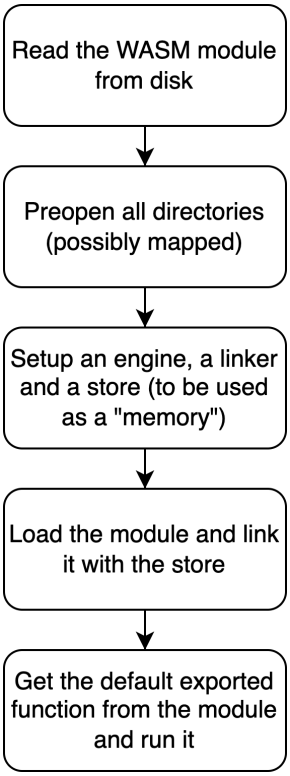


Figure 2.3: A high level view diagram of how the `wasm` module operates.

```rust
pub fn run(self) -> Result<()> {
  let wasi_ctx = self.ctx_builder.build();
  let engine = Engine::default();

  let mut linker = Linker::new(&engine);
  wasmtime_wasi::add_to_linker(&mut linker, |s| s)?;

  let mut store = Store::new(&engine, wasi_ctx);
  let module = Module::new(&engine, &self.bytes)?;
  linker.module(&mut store, "", &module)?;
  linker
    .get_default(&mut store, "")?
    .typed::<(), (), _>(&store)?
    .call(&mut store, ())?;

  Ok(())
}
```

Listing 2.5: The implementation of `WasmModule`'s `run` function.

```rust
pub struct Landlock {
  ruleset: RulesetCreated,
}

impl Landlock {
  // Creates a new ruleset with flags from
  // Landlock ABI version 1
  pub fn new() -> Result<Self, RulesetError> {...}

  // Add a set of rules to the ruleset
  pub fn add_rules(
    mut self,
    rules: impl Iterator<Item = PathAccess>)
    -> Result<Self>
  {...}

  // Add a single rule to the ruleset
  pub fn add_rule(
    mut self,
    path_access: PathAccess) -> Result<Self>
  {...}

  // Self restrict the process and checks whether Landlock
  // is supported or not by the running kernel
  pub fn enforce(self) -> Result<RestrictionStatus>
  {...}
}
```

Listing 2.6: The outline of the `landlock` module.

The `landlock` module

The `landlock` module is a thin wrapper around the API made available by the *rust-landlock* crate. It handles mostly ruleset creation, rule insertion, and enforces the policies before the desired WASM module is run. The main outline of the code is presented in Listing 2.6.

The `path_access` module

The `landlock` module makes use of another thin wrapper, the `PathAccess` struct defined in the `path_access` module. Here, the `PathAccess` struct tracks which flags, as defined in Table 2.2, are applied to a single path. The flags list starts empty, and multiple flags can be added at any time. Moreover, this module also takes care of conversions between types required by the Landlock ABI to make the code compile.

2.2.2 Available permission settings

The program allows to specify all access right flags from the first version of the Landlock ABI[1], which are listed in Table 2.2. The following examples illustrate how to combine some of them:

- "`file:R`" allows only to read from *file*;
- the combination of "`file:X`" and "`.:RDir`" allows the execution of *file* and the listing of the current directory;
- "`.:MReg`" only allows the creation, rename or linking of regular files in the current directory;
- "`.:R,W`" and "`./subdir:*`" allow only reads and writes on files in the current directory, while no restrictions are made for *subdir*;
- "`.:*,-X,-W`" allows every action on the current directory except for the reading and writing of files.

Note that the directories containing the files of interest must always be preopened, even if their content is then restricted by Landlock. This is because the preopen operation is required by the WebAssembly runtimes, while specific permissions are handled by Landlock.

[1] At the moment it is not possible to specify the `LANDLOCK_ACCESS_FS_REFERER` in order to reparent a file hierarchy.

The Landlock self-restriction is applied after the necessary preopening of the directories, otherwise the running process would have to always have the permissions for listing directories even if not needed by the WASM binary.

Flag	*Enabled permission*
`X`	Execute a file
`W`	Write to a file
`R`	Read a file
`RDir`	Open a directory or list its content
`DDir`	Delete an empty directory or rename one
`D`	Unlink or rename a file
`MChar`	Create, rename or link a character device
`MDir`	Create or rename a directory
`MReg`	Create, rename or link a regular file
`MSock`	Create, rename or link a socket
`MFifo`	Create, rename or link a named pipe
`MBlock`	Create, rename or link a block device
`MSym`	Create, rename or link a symbolic link
`read`	Combination of `X`, `R` and `RDir`
`write`	Combination of all but `X`, `R` and `RDir`
`*`	Enable all flags

Table 2.2: Provided flags and corresponding Landlock permissions.

2.2.3 Advantages and disadvantages

The main advantage that Landlock brings is simplicity. For the developer, the API is easy enough to use, and it can be accessed in a variety of languages – the main three languages are C (directly with the kernel libraries), Rust (as in this project), and Go[2]. From the user's point of view, the provided permission flags are clear enough that it is required to have only a basic understanding of Linux's file system permissions. Hence, it can be used to provide a clear interface that is not so different from the one already given by the existing runtimes. Moreover, Landlock allows *unprivileged* access control – it is not necessary to have particular privileges when running a binary, so the user can run a WebAssembly module in a protected environment even if they are not in the *sudoers* group.

The biggest drawback with this approach is the lack of advanced customisation options. As highlighted in Section 1.3.1, there is no way to define exceptions – this restricts the possibilities given to the user, and makes some situations impossible to describe. For example, if a `LANDLOCK_ACCESS_FS_REMOVE_FILE` right is given to a directory, all files in that directory can be renamed and there is no way to prevent renaming for a single file. Or similarly, if we have given the `LANDLOCK_ACCESS_FS_READ_FILE` and `LANDLOCK_ACCESS_FS_WRITE_FILE` permissions on a certain directory, then all files that are children of that directory can be read or written, even if they are inside subdirectories. If we then try to "restrict" this behaviour by giving only read permissions to certain subdirectories, they would still have the write permissions from the parent directory. In addition to this, the set of capabilities is somewhat limited, and mostly related to the file system.

[2] `https://github.com/landlock-lsm/go-landlock`

2.3 Restricting the WASI sandbox with eBPF

BPFContain [6] is a container security daemon for Linux written in Rust that offers interoperability with *eBPF* [7]. Since *BPFContain* is an external tool and not a library such as *rust-landlock*, there was no need to develop a project from scratch, since it is possible to combine it with the WebAssembly runtime of choice.

It should be noted that *BPFContain* is still in active development and not yet feature-complete. Finally, for this tool to work, BPF should be enabled when compiling the Linux Kernel.

2.3.1 Using *BPFContain*

The usage is pretty straightforward:

1. run the daemon once in the foreground in order to create the work files and directories – these are mostly log files and a directory used for the policies in `/var/lib/bpfcontain`, which permissions are set so that its content is readable, writable and executable by the daemon;
2. create a policy in `/var/lib/bpfcontain/policy`;
3. start the daemon with super user privileges;
4. run confined program by referring the policy name, and the policy itself can be written in YAML.

2.3.2 Available permission settings

The number of permission options that is possible to specify is way greater than the one offered by Landlock. A policy is hence separated into multiple parts:

- a *name* for the policy file and a *command* to run, both required;
- an optional set of *rights* to grant; a right can appear in different forms, such as:

  - a right on specific devices, such as `terminal` or `/dev/null`;
  - a grant on a particular folder or file, which is usually a combination of the classic `rwxa` permissions;
  - access to the kernel's capabilities (e.g., DAC override);
  - a list of process names with which communication is allowed;
  - a list of primitives that the program can use on the network (such as `client`, `send`, `recv`).

- an optional set of *restrictions* to explicitly revoke some rights, which follows the same structure as the granted rights;
- and finally an optional set of *taint rules*, which again follow the same structure described above.

Specifically, BPFContain offers, through the taint rules, a way to selectively choose when to start restricting a process. This means that a process monitored by BPFContain can start its life without any restriction, but as soon as it does a particular action (e.g., access the network or send a message to another process) BPFContain will begin to enforce the policy. In this case, the policy becomes *tainted* and starts being enforced. This however must be explicitly declared in the policy by setting `defaultTaint` to `false`, since all policies are tainted, hence enforced, by default.

An example of this kind of "untainted" policy is shown in Listing 2.7, which defines how to run *bash* in a confined environment. In this policy, *bash* is allowed to access the terminal and `/dev/null`, as well as reading and executing everything in `/` and reading everything in `/proc`. An important point is the `deny` section – here *bash* is prevented to writing and appending anything in all of the file system. Lastly, this policy is not active by default, but the process becomes "tainted" and the policy is enforced when *bash* tries to write or append anywhere in the file system.

A more complex example is presented in Listing 2.9, which shows a policy used to run *httpd*, allowing reading the files to serve, reading and appending to log files, and allowing inter-process communication only with running *mysqld* processes.

In the specific case of WebAssembly, it is enough to simply set the `cmd` option as a call to the WASI runtime with the desired binary, *Wasmtime* or *Wasmer*, making sure that necessary directories are preopened and eventual environment variables are imported. An example of a policy is visible in Listing 2.8, where *copy-file.wasm* simply copies the content of `input.txt` to `output.txt`.

```
# Name and command of the policy
name: bash
cmd: /bin/bash
defaultTaint: false

allow:
  # Allow access to terminal and /dev/null
  - device: terminal
  - device: null

  # Allow read and execute access on the file system
  - fs: {pathname: /, access: rxm}
  - fs: {pathname: /proc, access: r}

deny:
  # Prevent writes
  - fs: {path: /, access: wa}

taint:
  - fs: {path: /, access: wa}
```

Listing 2.7: A policy for running bash.

```
# Name and command of the policy
name: copy-file
cmd: wasmtime --dir=. copy-file.wasm

allow:
  - fs: {pathname: ., access: rwxm}
```

Listing 2.8: Running WASM with BPFContain.

```
# Name and command of the policy
name: httpd
cmd: httpd

allow:
  # Allow access to terminal and /dev/null
  - dev: random
  - dev: null

  # Access to log files
  - file: {pathname: /var/log/httpd, access: rw}
  - file: {pathname: /var/log/httpd/*.log, access: ra}

  # Read configuration
  - file: {pathname: /etc/httpd, access: r}

  # Serve files
  - file: {pathname: /srv, access: r}

  # Read hostname information
  - file: {pathname: /etc/host*, access: r}

  # Shared libraries loaded at runtime
  - file: {pathname: /usr/lib/httpd/modules/*.so,
           access: mr}
  - file: {pathname: /usr/lib/libnss*.so.*, access: mr}
  - file: {pathname: /usr/lib/libgcc_s.so.*, access: mr}

  # Allow ipc with MySQL
  - ipc: mysqld

  # Allow sending signals to existing httpd instances
  - signal: {to: httpd, signals: [check, superFatal]}

  # Use networking
  - net: [server, send, recv]
```

Listing 2.9: A policy for HTTPD.

2.3.3 Advantages and disadvantages

The main advantage of using *eBPF* is the extreme customisation in defining all the permissions and capabilities that a process can have. Most notably, when compared with Landlock, there is support for exceptions and restrictions, so that more cases can be covered.

A point of note is the ability to restrict communication with other processes, be it by allowing or disallowing IPC, or specifying what signals can be sent and to whom. Moreover, the restrictions and permissions applied to files and directories are more specific and do not necessarily transfer to subdirectories. As seen in the examples in the previous sections, one can also use *wildcards* when describing paths, such as the `/var/log/httpd/*.log` in Listing 2.9.

Finally, other desirable permissions are covered, such as network and devices. Network was also present in Landlock, but only regarding the creation and management of sockets. Similarly, Landlock offered creating, renaming and linking devices, but eBPF allows to specify which particular devices can be accessed and which not.

However, eBPF brings also some disadvantages that Landlock does not have. The biggest downside is regarding usability – while Landlock can be used by anyone able to use the terminal and understand the basics of passing parameters, BPFContain requires the writing of a policy in a specific language, albeit a simple one like YAML, and this policy must be in a very specific location[3]. This obviously decreases the level of simplicity and usability for a user that wants to contain and restrict some binaries.

Another point regarding the specific usage with *Wasmtime*[4] and WebAssembly is the necessity of having executable privileges – since *Wasmtime* offers a command line binary, then in order to run a WebAssembly module the confined process must be able to execute the necessary commands. This however can transfer this permission to the WebAssembly module itself – execution rights can be restricted only to a specific file system location, but it is still something that has to be taken into

[3]To be fair, this is a limitation of *BPFContain* and not of *eBPF* itself.

[4]Or, more generally, any WASM runtime that offers a command line interface.

consideration.

Lastly, with this method it is necessary to run a *separate* daemon that does the actual confinement, and this daemon must be run as a super user. Hence, a user that is not able to use the `sudo` command is not able to leverage eBPF expressive power. This is in stark contrast with Landlock, which sets out to offer *unprivileged* access control, so that super user privileges are not required to enforce the sandbox.

2.4 Final product

The final product of this development process is a *proof-of-concept* WebAssembly runtime that is natively integrated with LSM, in particular with Landlock as eBPF is enforced externally through the server managed by BPFContain. As highlighted before, the project is open source and available at `https://github.com/micheleberetta98/rust-wasm-landlock`.

In order to run a WASM binary, it is necessary to provide its name and directories to preopen (in case there are any) through the `--dir` and `--mapdir` command line arguments.

Landlock enforcement is done through the `--fs-allow` argument, which accepts a string in the format `path:restrictions`, where `path` is any folder or file path that the binary should be able to access and `restrictions` is a comma-separated list of flags as outlined in Table 2.2. This argument can be repeated to set different access restrictions on different paths. As an example, if `print.wasm` is a WASM module that prints the content of a certain `file.txt`, then a possible command to run it with Landlock could be `./rust-wasm-landlock print.wasm --dir . --fs-allow "file.txt:R"`, where `rust-wasm-landlock` is the name of the WASM runtime binary.

On the other hand, eBPF enforcement is done externally through BPFContain, and as such the WASM runtime is treated as any other native process. In this case, the desired restrictions must be listed in a policy as described in Section 2.3 and then the BPFContain server must be running. Note that the relevant directories have to be preopened in the `cmd` section of the policy through the `--dir` and `--mapdir`

arguments, but Landlock has to be disabled through `--no-landlock`, otherwise the process would be restricted by Landlock and by eBPF at the same time.

Chapter 3

Performance

3.1 Testing plan

The main goals of this performance testing plan are:

- to find out whether the usage of LSM implies some sort of performance penalty;
- to identify possible improvements when embedding WebAssembly in another language.

For a given program written in Rust, the following configurations are tested in order to compare overall execution speed in both restricted and unrestricted scenarios:

- a native binary, compiled by `rustc` and run:
  - without a sandbox;
  - sandboxed with Landlock;
  - sandboxed with eBPF.
- a WebAssembly binary obtained by the same Rust program and run:
  - directly by the *Wasmtime* command line tool;
  - by the WASM runtime developed in Section 2.2 and restricted with Landlock;

– by the WASM runtime developed in Section 2.2 and restricted with eBPF[1].

When testing the native binary with Landlock, the program has to be modified. That is, some instructions are added to the code that setup Landlock in a way compatible with the particular test. An example of the required instructions is visible in Listing 3.1. On the other hand, for the other methods (i.e., without a sandbox and with eBPF) the program does not need any change, but in the case of eBPF it is necessary to use the `bpfcontain run` command with the name of the chosen policy to actually apply the sandbox.

Moreover, the internal performance of the WASM runtime developed in Section 2.2 and the performance of the sandboxing methods will be tested in Section 3.3.

```
1  use anyhow::Result;
2  use landlock::*;
3
4  fn main() -> Result<()> {
5      let path_fd = PathFd::new("input-file.txt")?;
6      let access = make_bitflags!(AccessFs::{ ReadFile });
7
8      let path_beneath = PathBeneath::new(path_fd)
9        .allow_access(access);
10     let _ = Ruleset::new()
11         .handle_access(AccessFs::from_all(ABI::V1))?
12         .create()?
13         .add_rule(path_beneath)?
14         .restrict_self()?;
15
16     // The program's code goes here...
17
18     Ok(())
19 }
```

Listing 3.1: An example of a program restricted with Landlock.

[1]In this case, Landlock is disabled.

3.1.1 System and hardware

The system and hardware used for both the development and test of the project described in Section 2.2 is as follows:

- Arch Linux [4] as the operating system, more specifically the 2022.05.01 version;
- an Intel Core i5-7200U quad-core with a clock rate of 2.5 GHz;
- 8 GB of RAM;
- a 120 GB solid state disk.

The choice of the operating system is mainly dictated by the fact that Arch Linux has both Landlock and eBPF active out of the box, removing the need to compile the Linux Kernel with the necessary flags to enable these functionalities.

3.1.2 Test scenarios

The performed tests are:

- a purely computational program, given by the sorting of 10000 random numbers as in Listing 3.2, in order to measure only the computational impact of the various methods;
- a simple reading of files of various sizes as in Listing 3.3, with only the required permissions enabled on a case-by-case base, in order to test the sandboxes provided and how they fare against native binaries.

The reading file test is repeated with different file sizes[2], which are 100 KB, 1 MB, 10 MB and 100 MB. By doing this, it is possible to see how performance varies when dealing with progressively larger files. Moreover, an additional empty file is

[2]Randomly generated by using `/dev/urandom`.

used to test the overhead on the pure opening of a file.

```rust
use rand::Rng;

fn main() {
  let mut rng = rand::thread_rng();
  let mut vec: Vec<i32> = Vec::new();

  for _ in 0..10000 {
    vec.push(rng.get::<i32>());
  }

  vec.sort();
}
```

Listing 3.2: The “sorting program”.

```rust
fn main() {
  let content = std::fs::read("input-file.txt");
  match content {
      Ok(c) => println!("{}", c.len()),
      Err(_e) => std::process::exit(1),
  }
}
```

Listing 3.3: The “reading program”.

3.1.3 Performance indicators

The main performance indicators will be the mean execution time, measured in milliseconds, together with its standard deviation in order to compensate for variability. These measures are always obtained from a sample of 100 runs, executed and measured by `hyperfine`, a command-line benchmarking tools [15], and finally saved to a JSON file.

In this chapter, we define for brevity μ_x to be the mean of the method x and σ_x to be the standard deviation of the method x.

Test type	μ_{Native}	σ_{Native}	μ_{Landlock}	σ_{Landlock}	μ_{eBPF}	σ_{eBPF}
Sorting 10000 numbers	0.82	0.07	1.03	0.25	2.30	0.10
Opening a file	0.49	0.07	0.48	0.06	1.75	0.09
Reading a 100 KB file	0.53	0.15	0.56	0.10	1.79	0.07
Reading a 1 MB file	1.06	0.79	0.99	0.07	2.21	0.07
Reading a 10 MB file	5.51	2.57	5.26	0.13	6.44	0.09
Reading a 100 MB file	47.32	17.71	45.57	0.60	46.86	0.70

Table 3.1: Execution times of a native binary under different restrictions (in ms).

Test type	μ_{Wasmtime}	σ_{Wasmtime}	μ_{WL}	σ_{WL}	μ_{WeBPF}	σ_{WeBPF}
Sorting 10000 numbers	18.00	1.35	44.20	2.05	45.72	2.14
Opening a file	17.12	0.24	45.83	1.73	47.40	2.12
Reading a 100 KB file	17.47	1.62	45.90	1.71	47.50	2.06
Reading a 1 MB file	17.97	1.88	46.44	2.08	47.91	2.17
Reading a 10 MB file	23.21	0.76	51.93	2.16	53.41	2.00
Reading a 100 MB file	75.68	1.31	104.91	3.25	105.92	1.90

Table 3.2: Execution times of a WASM binary under different restrictions (in ms).

3.2 Comparison between different sandboxes

In this section we will look at what impact different LSMs bring on a program, be it a native binary or a WebAssembly binary run through some kind of runtime.

Table 3.1 reports all execution times for a native binary, compiled directly with `rustc` and run both unrestricted (*Native*) and restricted with different LSMs (*Landlock* and *eBPF*).

Furthermore, Table 3.2 lists the runs of a WebAssembly binary through *Wasmtime* without any restriction, or through the WASM runtime developed in Section 2.2 and restricted with Landlock (*WL*) or eBPF (*WeBPF*).

3.2.1 A computational-heavy program

This section analyses how restricting a purely computational-heavy program, namely the sorting of 10000 random numbers shown in Listing 3.2, affects its performance. In order to test the eBPF sandbox, the policy outlined in Listing 3.4 will be used.

Comparing the first lines of Table 3.1 and Table 3.2, it is immediately apparent that using a LSM to restrict a binary, be it native or WebAssembly, worsen performance.

When dealing with a native binary, *Landlock* has a lower overhead than *eBPF* (an average of 1.03 ms for Landlock against an average of 2.30 ms for eBPF), as it is visible in Figure 3.1a. This could be due to how the two sandboxes are enforced:

- Landlock is directly available as a C library in the Linux Kernel, so the communication between the program and the relative LSM can be as simple as multiple function calls;
- on the other hand, eBPF needs to have an active server[3], so the communication has to undergo a message exchange between processes.

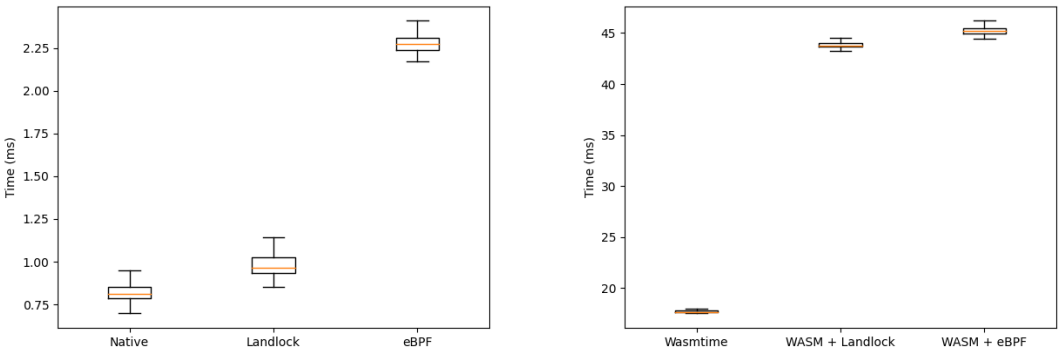
By following this line of reasoning, it is expected that Landlock should be a little faster than eBPF. Furthermore, eBPF is more complex than Landlock since it allows more control on what can and cannot be done by a running process, hence it has to deal with a higher complexity. The overhead of eBPF, however, becomes smaller and smaller when measured in percentage as the execution times get higher. This means that for small and fast programs the slow-down could become apparent if they get called many times in a short period of time, while for slower program called with less frequency this should be negligible.

```
1 name: test-sorting
2 cmd: # The command to be executed
3 allow:
4   - fs: {pathname: ., access: rxm}
```

Listing 3.4: The outline of the policy used for testing the sorting program.

[3]As highlighted in Section 2.3.

When dealing with a WebAssembly binary, the computational-heavy program is noticeably slower compared to the native binary execution times, either unrestricted or restricted, as shown in Figure 3.1b[4]. The same reasoning applied before is valid here too – eBPF is a little slower than Landlock, with a overhead of about 1 ms. However, restricting a WASM binary with the runtime developed in Section 2.2 introduces a constant overhead of around 30 ms with respect to *Wasmtime*. Since it is shown in Table 3.1 that Landlock is quite close to unrestricted performance on average, and because Landlock was disabled when running the tests combining WASM and eBPF, this extra time could be due to how the WASM module gets interpreted by the library provided by *Wasmtime*[5].



(a) Native

(b) WebAssembly

Figure 3.1: Execution times when sorting 10000 numbers.

[4]Note that Figures 3.1a and 3.1b have different scales.

[5]This will be analysed more accurately in Section 3.3.

3.2.2 Opening a file

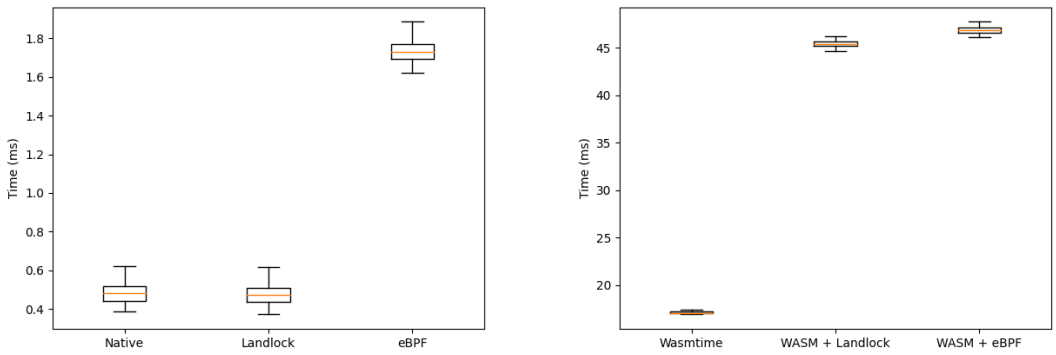
This set of tests is meant to measure how the sandboxes perform when the restricted program opens a file. In this case, the performance is measured by reading an empty file with the program described in Listing 3.3. For the policy used with eBPF, its outline is highlighted in Listing 3.5 – in this case, it is necessary to give permissions to access the terminal since the size of the read content is printed.

As Tables 3.1 and 3.2 show, the same considerations made for a purely computational program are valid in this case too – Landlock, on average, is much closer to native speed than eBPF. This is also shown in Figure 3.2a, where the eBPF average is shifted towards higher execution times.

Similarly, the WASM tests present the same behaviour remarked in the previous section – even when unrestricted, WASM is much slower than a native binary, and the developed runtime still has the constant overhead of around 30 *ms* in both the tests within the Landlock and eBPF sandboxes. Again, Figure 3.2b shows the box plot of execution times.

```
1 name: test-reading-file
2
3 cmd: # The command to be executed
4
5 allow:
6   - device: terminal
7   - fs: {pathname: ., access: rxm}
```

Listing 3.5: The outline of the policy used for testing the reading program.



(a) Native

(b) WebAssembly

Figure 3.2: Execution times when opening a file.

3.2.3 Accessing the file system

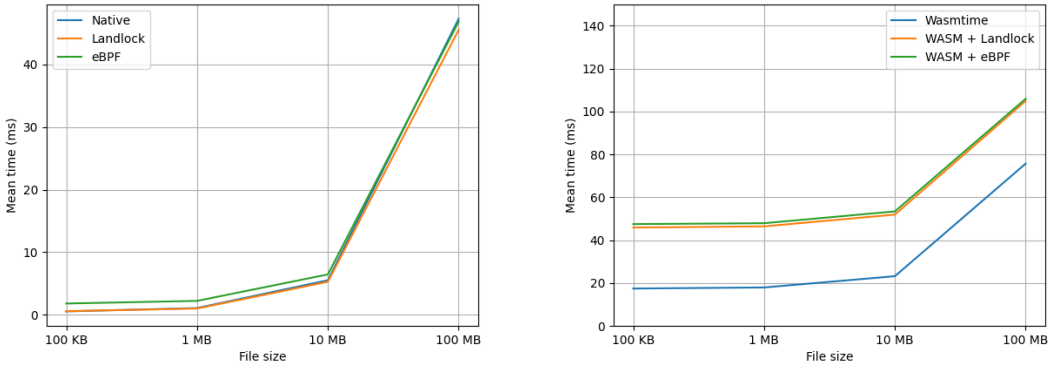
This last batch of tests is meant to show how sandboxed binaries, both native and WASM, perform when dealing with progressively larger files. The following figures, namely Figure 3.3a and Figure 3.3b, represent how the average execution time of a native binary and a WASM binary, respectively, changes when the read file gets bigger.

The first notable feature is that performance drops as the file gets larger, starting to get worse when reading a file in the order of tens of MBs, and with a significant spike in execution times when reaching the order of hundreds of MBs. This slowdown is clearly visible in the native curves as well as in the WASM curves, hence it has to be independent of the method used to run the program and only due to external circumstances, such as the operating system or the disk reading speed. For small files, however, there is not a notable change in speed.

Another point of note is that in Figure 3.3a the gap between eBPF and Landlock, identified in the previous sections, is clearly visible at all file sizes. This can be seen also in Figure 3.3b, albeit less clearly as the file reaches 100 MB. Note that, while the WASM runtime used for restricting a WASM binary with Landlock is the same used when testing eBPF, in this last case Landlock was disabled altogether, so any performance difference is only due to eBPF's internals.

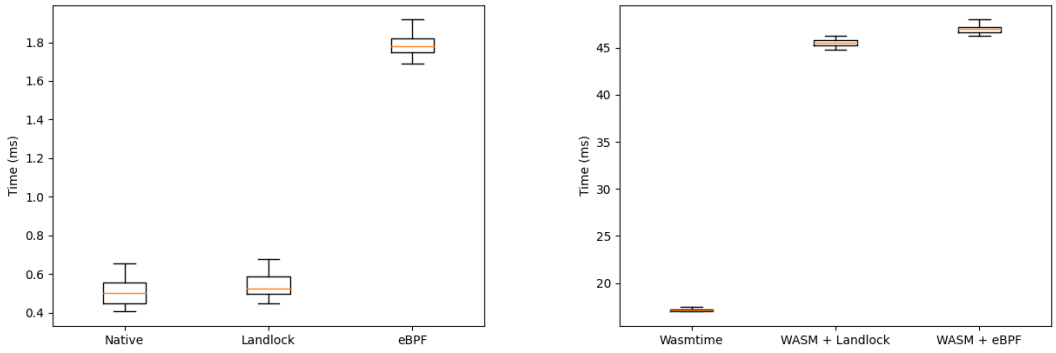
Again, the WASM curves in Figure 3.3b clearly show the overhead of about 30 ms identified in the previous sections – the *WASM+Landlock* and *WASM+eBPF* curves exhibit the same trend as the *Wasmtime* one, but appear translated by a constant factor.

Finally, Figures 3.4, 3.5, 3.6, and 3.7 collect all box plots relative to the different readings for native binaries and WASM binaries.



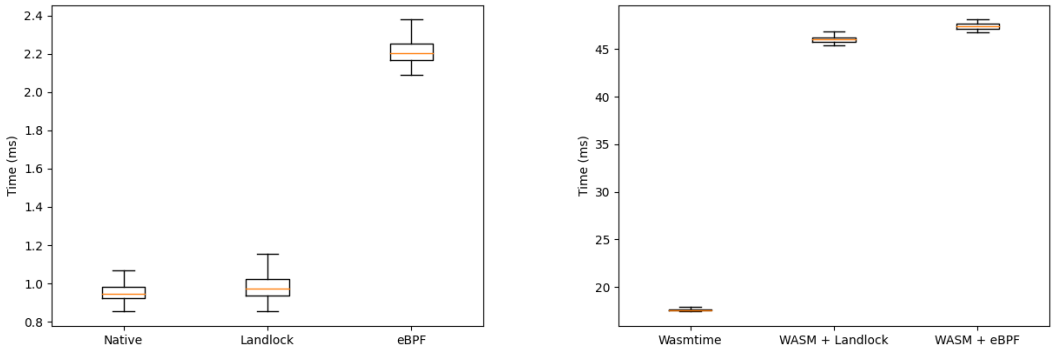
(a) Native (b) WebAssembly

Figure 3.3: A comparison of average speeds when reading a file.



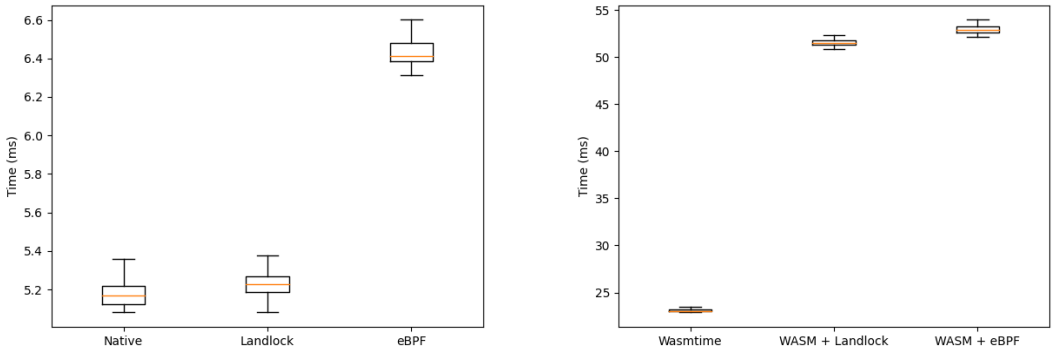
(a) Native (b) WASM

Figure 3.4: Execution times when reading a 100 KB file.



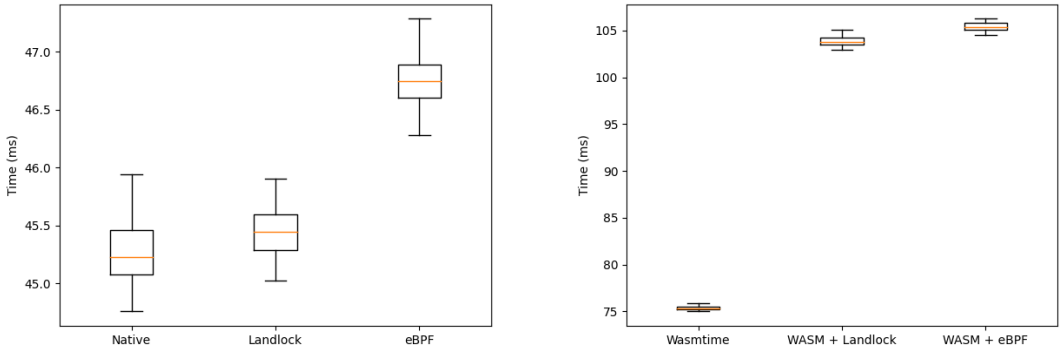
(a) Native (b) WASM

Figure 3.5: Execution times when reading a 1 MB file.



(a) Native (b) WASM

Figure 3.6: Execution times when reading a 10 MB file.



(a) Native (b) WASM

Figure 3.7: Execution times when reading a 100 MB file.

3.3 Internal analysis of the various sandboxes

The goal of this series of tests is to measure how the developed runtime, described in Section 2.2, behaves internally as well as how Landlock and eBPF scale with the complexity of the rules to implement.

The various measures are taken by using the standard Rust library `std::time`, in particular the `Instant` struct, and then averaged on a total of 100 runs. The programs used as reference are:

- a trivial one described in Listing 3.6, dealing only with printing a simple string to the terminal;
- a more complex one shown in Listing 3.7, reading and writing some files, so that specific permissions are required (in particular, reading and writing files).

```
1 fn main() {
2   println!("Hello World");
3 }
```

Listing 3.6: The trivial program.

```
1 use std::fs::{read_to_string, write};
2
3 fn main() -> {
4   read_to_string("file1.txt").expect("Read error");
5   write("file1.txt", "Content 1").expect("Write error");
6   read_to_string("file2.txt").expect("Read error");
7   write("file2.txt", "Content 2").expect("Write error");
8 }
```

Listing 3.7: The more complex program.

3.3.1 Execution times of the developed runtime

In order to measure the execution times of the developed runtime, various timestamps were taken and combined directly in the code in order to have some data on the execution time of important program parts, which are the following:

- command line argument parsing;
- WASM module instantiation;
- the preopening of all directories by the Wasmtime library;
- collecting and applying all the Landlock rules;
- and finally, compiling and running the WASM module.

Note that in this case only Landlock is considered – eBPF is enforced externally by a different process, while Landlock system calls are embedded in the code of the runtime. Hence, only the impact of Landlock can be measured *internally* in the code.

Program section	μ_I	σ_I	μ_C	σ_C
Argument parsing	0.056	0.006	0.056	0.006
Module initialisation	0.801	0.051	0.857	0.054
Preopen	0.005	0.001	0.863	0.054
Landlock enforcement	0.012	0.003	0.874	0.056
Running WASM binary	37.144	0.882	38.018	0.921

Table 3.3: Execution times in ms when running the simple program (Listing 3.6).

Tables 3.3 and 3.4 contain all measured data in milliseconds for, respectively, the trivial and complex program. The recorded timestamps are both the average time spent by a section, given by μ_I, and the cumulative time up until that section's completion, given by μ_C. Standard deviations σ_I and σ_C are also given for any respective program part. For example, if the part "Landlock enforcement" is such that $\mu_I = 20\ ms$ and $\mu_C = 100\ ms$, this means that on average it takes 20 ms to

Program section	μ_I	σ_I	μ_C	σ_C
Argument parsing	0.067	0.009	0.067	0.009
Module initialisation	0.832	0.094	0.899	0.101
Preopen	0.012	0.005	0.911	0.105
Landlock enforcement	0.018	0.004	0.930	0.108
Running WASM binary	46.731	9.697	47.661	9.760

Table 3.4: Execution times in *ms* when running the complex program (Listing 3.7).

apply Landlock to the running process, and on average this part is fully completed after 100 *ms* from the beginning of the program.

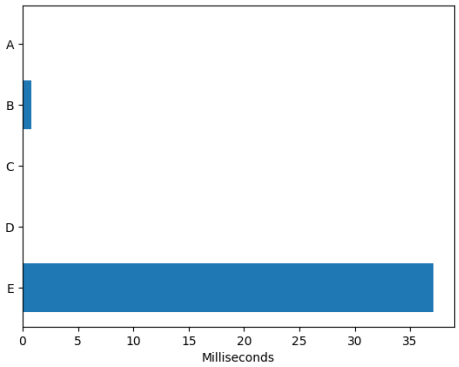
Figure 3.8 shows how each program section occupies the overall time spent running by the program. The various labels are as follows – `A` is for "Argument parsing", `B` is for "Module initialisation", `C` is for "Preopen", `D` is for "Landlock enforcement" and `E` is for "Running WASM binary".

As it is immediately noticeable by the collected data, the biggest bottleneck on performance is given by *running the WebAssembly binary* with the library provided by *Wasmtime*. This section comprises of instantiating the required data structures, interpreting and linking the WASM module, and finally running it, as already highlighted in Section 2.2.1. Running the WebAssembly binary alone makes up 97.7% of the time spent running the trivial program, and 98.1% of the time spent running the complex program.

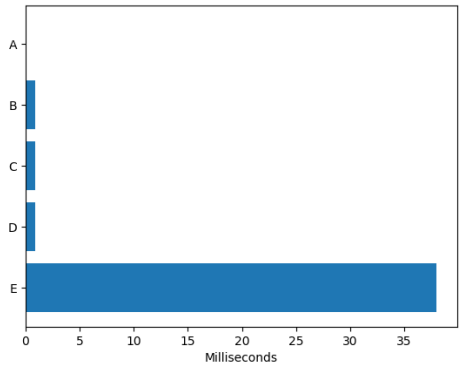
The next biggest chunk of time is spent into *module initialisation*, which is comprised of the reading of the WASM binary file into memory and the instantiation of specific context structures used to keep track of preopened directories. For both programs, this is around 800 μs in duration.

On the other hand, enforcing Landlock does not have a substantial impact on performance, and it is even faster than the argument parsing algorithm.

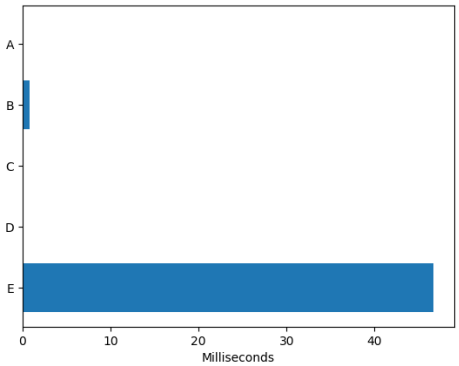
These findings further support the data shown in Section 3.2 – it was found that the developed WASM runtime was slower than calling *Wasmtime* from the command line, but because Landlock had little overhead when dealing with native



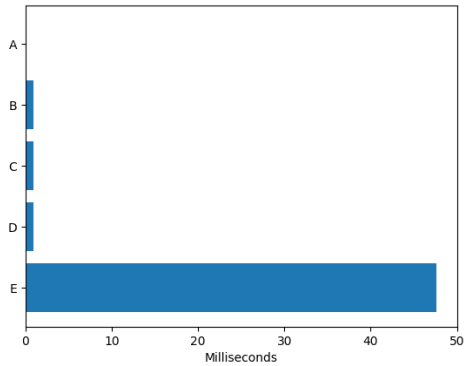
(a) Simple program.



(b) Simple program (cumulative).



(c) Complex program.



(d) Complex program (cumulative).

Figure 3.8: A graphic comparison of all the execution times.

binaries this implied that the "culprit" for the overhead could be the *Wasmtime* library. Moreover, in this case it can also be seen than the time took for running the WASM binary is, in both cases, about the same as the constant overhead.

3.3.2 Landlock performance scaling

In this section we will focus on how Landlock impacts performance and how it scales against the complexity of the specified rules. In particular, performance is measured against two "modes" of complexity scaling – adding permissions to a single folder, and allowing access to multiple folders with the same permissions. In order to do this, the code of the developed WASM runtime will be used, and the program tested will be the one described in Listing 3.7 – Landlock enforcement happens *before* running the WASM program of choice, so how much time the restriction takes should not depend on the program that has to be sandboxed.

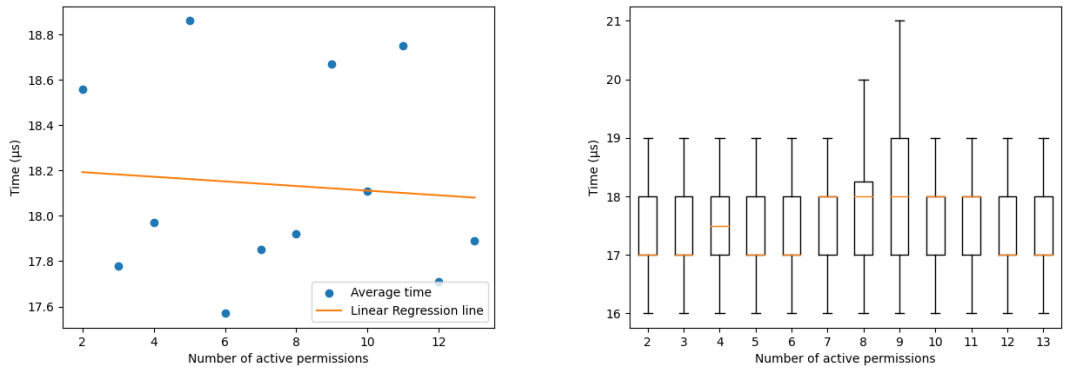


Figure 3.9: Relation between Landlock enforcement and the number of permissions.

For the first series of tests, specific permissions will be added one at a time to a single folder, starting from `R,W` (i.e., reading and writing files) and up to enabling all the permissions listed in Table 2.2. Figure 3.9 shows how the average time varies against the number of permissions. As it is immediately visible, execution time does not depend on the number of permissions on a single folder – the average time is always between 17 μs and 19 μs, without great variance. There are some outliers, but it is safe to conclude that the number of permissions does not affect performance when it comes to enforcing Landlock on the running WASM runtime. This is to be expected – the code treats all permissions as numbers, and when enforcing a rule on a single folder, all permissions are *xor*-ed together. This means that the cost of applying one rule is the same whatever the number of allowed actions is.

However, because Landlock rules are related to paths, then Landlock perfor-

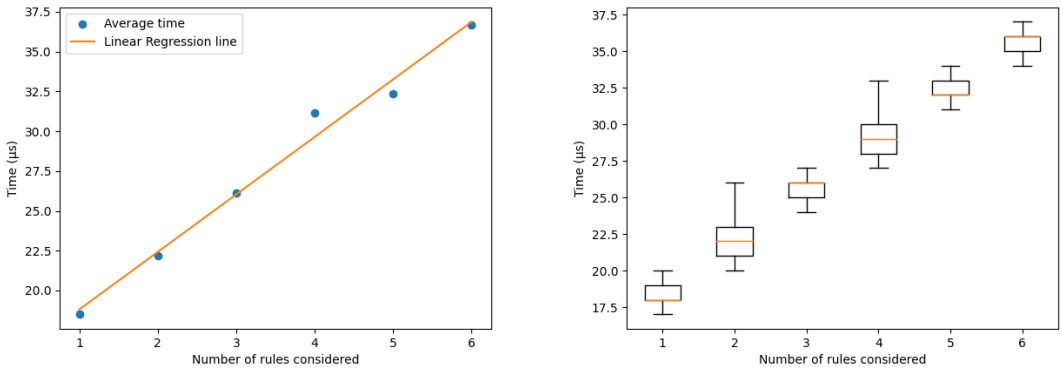


Figure 3.10: Relation between Landlock enforcement and the number of rules.

mance *should* be influenced by the number of rules used when specifying multiple paths (folders or files). In this series of tests, a different number of rules related to multiple folders[6] is taken into account for each performed test.

As can be seen in Figure 3.10, there is a clear correlation between the number of rules used and the time necessary for Landlock to enforce them. More specifically, this relation is linear – this is a reasonable result, since if n rules are given, then Landlock has to analyse them at least once in order to enforce all of them. Hence, an algorithm with linear complexity is the best possible that could be implemented in this case.

3.3.3 eBPF performance scaling

Similarly to the previous section, here we will test how eBPF initial policy enforcement scales with the complexity of a given policy using BPFContain. In order to be able to make a fair comparison with Landlock, only access to the file system will be taken into consideration, and the tests will be divided into scaling with the number of permissions on a single folder and scaling with the number of folders. Again, the program used for the tests will be the one described in Listing 3.7.

Note that BPFContain sandboxing is "split into two" – the policies are parsed and implemented by a *server* process, while a *client* process restricts itself with a chosen policy before running a given program and communicating with the server.

[6]It does not matter if they are related to a folder or a file, since both are considered as paths.

This initial restriction is what will be measured[7], so that a parallel with Landlock can be made.

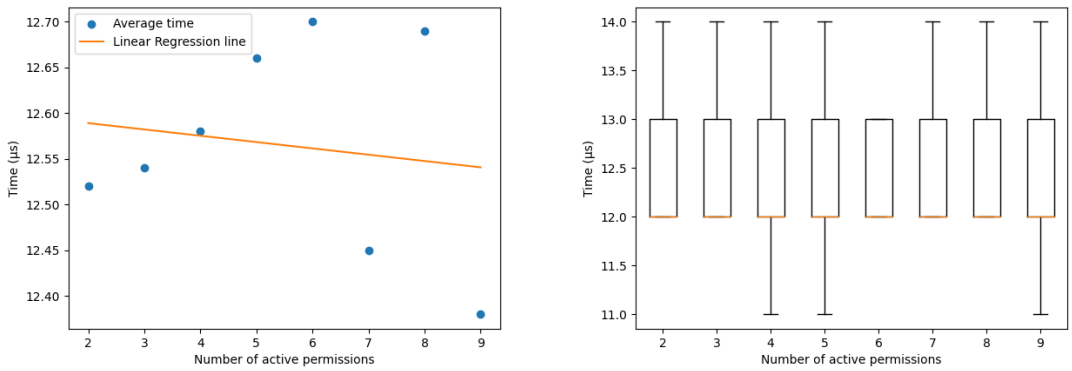


Figure 3.11: Relation between eBPF enforcement and the number of permissions.

As before, for the first series of tests permissions will be added one at a time to the policy which outline is visible in Listing 3.8, starting from `r` (read) and `w` (write) and ending with all the permissions allowed by BPFContain[8]. The results are shown in Figure 3.11 and, as with Landlock, the sandboxing of the running program does not depend by the number of permissions on a single folder.

```
1 name: test-policy
2 cmd: ./wasm-runtime program.wasm --no-landlock
3
4 allow:
5   - dev: terminal
6   - fs: {pathname: ./wasm-runtime, access: rxm}
7   - fs: {pathname: ./dir, access: rw}
8   # Other folders added here
```

Listing 3.8: The outline of the policy used for eBPF tests.

However, when looking at how eBPF scales with the number of paths provided there is a difference with Landlock. As Figure 3.12 shows, in this case there is no

[7]In order to do this, the code of BPFContain has to be modified. The version used for these tests is available at `https://github.com/micheleberetta98/bpfcontain-rs/tree/perf-test`.

[8]These are read, write, execute, append, delete, `chmod`, `mmap`, linking and `ioctl`.

relation between the number of folders specified and the time taken by BPFContain to containerise the running process. The average time is always around 12.35 μs, with little variability.

This could be due to the fact that, as already highlighted, BPFContain sandboxing uses two processes – a server and a client that communicates with each other. In this case, the sandboxing done on the client does not depend on how much the particular policy is complex, since it needs only to know which policy to implement, hence the constant complexity that emerges from the tests.

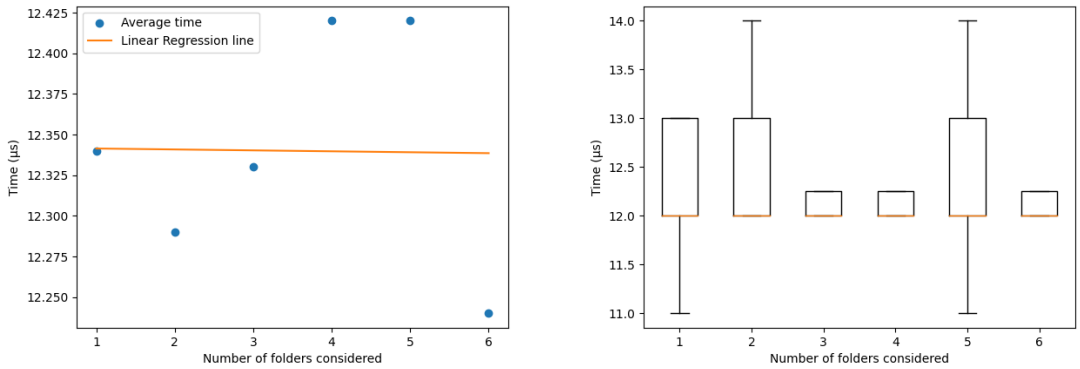


Figure 3.12: Relation between eBPF enforcement and the number of folders.

Lastly, note that the times here are all a bit lower than Landlock, but in Section 3.2 it was shown that BPFContain was a little slower overall. As hinted before, this could be due to the communication between the server and the client, so even with more initial overhead Landlock could come out faster nonetheless.

3.4 Conclusions and observations

One of the key points of this series of tests is that, although perfectly usable, WebAssembly is not still at the level of performance that can be obtained by using a native binary. As seen in Section 3.2, WASM binaries always perform worse than the native counterpart, even when not sandboxed. This result is however to be expected – a WASM binary is made up of WASM instructions and it is similar to a sort of *bytecode*, interpreted by the runtime of one's choosing, and usually a interpreted program[9] is slower than a compiled one. This is true also for other languages that follow this paradigm – for example, Java with its JVM or Python (that can be compiled into a bytecode before being interpreted) are usually regarded as slower and/or more resource-hungry when compared to compiled languages such as C or Rust.

When embedding WebAssembly in another language this performance gap can become higher (although performance itself is still comparable) due to the work that must be done through the provided interpreting libraries, as it has been highlighted in Section 3.3.1.

A positive point that has been shown multiple times is that restricting a binary with a LSM, be it Landlock or eBPF, does not worsen performance a lot. As seen in Section 3.2 a small performance impact is present, but this could be considered negligible for large programs that take some time to run. However, for small programs that have an execution time of the same order of magnitude as the delay introduced this could become problematic if they are called frequently. Furthermore, as found in Section 3.3.2, when using Landlock as a sandbox the overhead introduced is linear with respect to the number of rules (i.e., paths) specified, so programs that have to access few locations on the file system have an advantage against programs that need access to a lot of files and/or directories. However, this is not the case of eBPF, in which the particular policy only translates into a constant overhead when restricting the running process, as found in Section 3.3.3. The two restriction methods, Landlock and eBPF, are hence both viable, each with its own advantages

[9]Despite the program in question being a binary format and hence faster to parse.

and downsides but overall comparable performance.

Chapter 4

Conclusions

4.1 WebAssembly and LSM

WebAssembly, abbreviated to WASM, is a relatively new binary instruction format that strives to bring security, speed, portability, and compactness to the web as a low-level alternative to JavaScript. With the advent of the *WebAssembly System Interface* (WASI), this language can also be treated as a compilation target for languages such as C and Rust, and can be executed directly on various operating systems through the aid of runtimes, such as *Wasmtime* and *Wasmer*.

However, these runtimes' CLI interfaces do not provide extensive user control when dealing with the program's access rights to specific file system locations. These interfaces can be extended by using pre-existing access control frameworks, specifically the *Linux Security Modules* framework. Two frameworks in particular are taken into consideration – *Landlock*, to provide unprivileged scoped access control, and *eBPF*, a virtual machine that can run sandboxed program in a privileged context.

4.2 The developed project

This thesis work deals with the development of a proof-of-concept WASM runtime that provides the means to extend a WASM binary's access rights specifiable by the

user. The development of the project was divided into three main phases.

Firstly, an analysis of the existing runtimes, *Wasmtime* and *Wasmer*, was required to understand how they behave in different situations and what specific means they give to the user in order to restrict a WASM binary. As highlighted in Section 1.2.4, when using the default command line interface, there is a lack of a fine granularity on what a program can do – for example, the user cannot describe what a WASM binary is permitted to do on a preopened directory's files.

Secondly, the two LSM frameworks in question, *Landlock* and *eBPF*, were studied and understood so that they could be integrated with arbitrary code. For this phase, having a mainstream Linux distribution is not enough – Landlock and eBPF have to be enabled to be used by programs. This can be done either by manually compiling the kernel with the desired flags enabled, or by using particular distributions that have them enabled by default, such as Arch Linux.

Lastly, WASI libraries and the chosen LSM frameworks had to be integrated in a new runtime, described in Chapter 2, able to run WASM binaries while simultaneously sandboxing them according to the user's choice. When dealing with Landlock, the integration with a given program is provided by C libraries, which can also be wrapped in the language of one's choice – for this project, this language is Rust, in order to have a safer language that could still produce reasonably fast binaries. On the other hand, eBPF has to be integrated externally, as the sandboxing action is done by a server process, acting as a virtual machine. Between the two, Landlock is definitely the easier one, both from a user's point of view, regarding the available access control options, and from a developer's point of view, since the existing libraries are quite straightforward to use. Moreover, its usage does not need the running program to have root privileges. The eBPF sandbox, although being more complex and requiring root privileges, is however the more powerful one, allowing a great deal of fine tuning on multiple areas, such as file system, network and inter-process communication.

Regarding performance, WASM is still not close to native speeds, and the usage of WASI libraries brings a visible overhead against the more traditional runtimes, as shown in the tests in Chapter 3. However, LSMs themselves are not a hindrance

on performance if considered in isolation.

4.3 Possible improvements and future work

The first problem that could be addressed is the performance of the WASI libraries. In the testing runs, a very high percentage of the time is spent into loading and running the WASM module – acting as a bottleneck, these libraries are thus the first and most effective point to improve. This could be done either through optimisations in the original code, or through a more thorough configuration of the library's execution capabilities.

Another possible improvement for the existing WASM runtimes could be the integration of *Landlock* if the development of a custom full-fledged access control framework turns out to be too hard to implement. This could effectively bring a greater deal of control directly available to the user when accessing the file system, while still being relatively simple to use. However, Landlock is only available on Linux, and moreover it has to be enabled in the kernel. This would pose some problems – the runtime code would be less portable, since a custom version for Linux would have to be created, requiring the creation of more OS-specific code and making development more difficult. Furthermore, not all Linux users could take advantage of it, since enabling it can be a challenge that only more advanced users could be able and/or willing to face[1].

[1]This could be eased if Landlock were to be already enabled in most mainstream distributions.

Acknowledgements

I would like to express my deepest thanks to Professor Stefano Paraboschi, without whom this endeavour would have not been possible. Additionally, I am extremely grateful to all the PhD students in the *Seclab* group, who have followed and helped me during all phases of this work, from the research on the existing technologies to the development of the project and redaction of this thesis.

I am also very grateful to my close friends and classmates who stayed by my side for all these years, sharing all hardships and joyful moments. This has been a long journey, and I would not be here as the person I am now without their support. I would also be remiss in not mentioning my family for their aid throughout the years.

Lastly, a thank to all the many people that saw in me much more than what I saw in myself.

Bibliography

[1] *"WebAssembly" | Can I use... Support tables for HTML5, CSS3, etc.* URL: https://caniuse.com/?search=WebAssembly (visited on 05/2022).

[2] *ActiveX Controls.* URL: https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/aa751968(v=vs.85)?redirectedfrom=MSDN (visited on 05/2022).

[3] *AppArmor.* URL: https://www.apparmor.net (visited on 05/2022).

[4] *Arch Linux.* URL: https://archlinux.org.

[5] *asm.js.* URL: http://asmjs.org (visited on 05/2022).

[6] *BPFContain: a container security daemon for GNU/Linux leveraging the power and safety of eBPF and Rust.* URL: https://github.com/willfindlay/bpfcontain-rs.

[7] *eBPF – Introduction, Tutorials & Community Resources.* URL: https://ebpf.io (visited on 05/2022).

[8] *ECMA-262 – Ecma International.* June 2021. URL: https://www.ecma-international.org/publications-and-standards/standards/ecma-262/ (visited on 05/2022).

[9] *Emscripten.* URL: https://emscripten.org (visited on 05/2022).

[10] *FileSystem – Web APIs | MDN.* URL: https://developer.mozilla.org/en-US/docs/Web/API/FileSystem (visited on 05/2022).

[11] *Go Landlock library: a Go library for the Linux Landlock sandboxing feature.* URL: https://github.com/landlock-lsm/go-landlock.

[12] Antonio Gonzalez, Fernando Latorre, and Grigorios Magklis. *Processor Microarchitecture: An Implementation Perspective*. Morgan & Claypool Publishers, 2011.

[13] Andreas Haas et al. "Bringing the web up to speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. June 2017, pp. 185–200. DOI: https://doi.org/10.1145/3062341.3062363.

[14] Bobby Holley. *WebAssembly and Back Again: Fine-Grained Sandboxing in Firefox 95*. Dec. 2021. URL: https://hacks.mozilla.org/2021/12/webassembly-and-back-again-fine-grained-sandboxing-in-firefox-95/ (visited on 05/2022).

[15] *Hyperfine: A command-line benchmarking tool*. URL: https://github.com/sharkdp/hyperfine.

[16] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[17] *Landlock crate: a Rust library for the Linux Landlock sandboxing feature*. URL: https://github.com/landlock-lsm/rust-landlock.

[18] Daniel Lehmann, Johannes Kinder, and Michael Pradel. "Everything Old is New Again: Binary Security of WebAssembly". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 217–234. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

[19] *Linux Security Module Usage*. URL: https://www.kernel.org/doc/html/latest/admin-guide/LSM/index.html (visited on 05/2022).

[20] Salaün Mickaël. *Landlock LSM: kernel documentation*. Mar. 2021. URL: https://docs.kernel.org/security/landlock.html (visited on 05/2022).

[21] Salaün Mickaël. *Landlock: unprivileged access control*. Mar. 2021. URL: https://www.kernel.org/doc/html/latest/userspace-api/landlock.html (visited on 05/2022).

[22] Shravan Narayan et al. “Retrofitting Fine Grain Isolation in the Firefox Renderer”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 699–716. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/narayan.

[23] *Native Client*. URL: https://developer.chrome.com/docs/native-client/ (visited on 05/2022).

[24] *Practical third-party library sandboxing with RLBox*. URL: https://docs.rlbox.dev (visited on 05/2022).

[25] *sandboxing – Wasmtime*. URL: https://docs.wasmtime.dev/security-sandboxing.html (visited on 05/2022).

[26] *SELinux Project*. URL: https://selinuxproject.org (visited on 05/2022).

[27] *Smack Project*. URL: http://schaufler-ca.com (visited on 05/2022).

[28] *WASI – WebAssembly System interface*. URL: https://wasi.dev (visited on 05/2022).

[29] *WASI Web Polyfill*. URL: https://wasi.dev/polyfill/.

[30] *WasiState in `wasmer_wasi` – Rust*. URL: https://docs.rs/wasmer-wasi/latest/wasmer_wasi/struct.WasiState.html.

[31] *Wasmer, The Universal WebAssembly Runtime*. URL: https://wasmer.io (visited on 05/2022).

[32] *Wasmtime – a small and efficient runtime for WebAssembly & WASI*. URL: https://wasmtime.dev (visited on 05/2022).

[33] *WebAssembly*. URL: https://webassembly.org (visited on 05/2022).

[34] *WebAssembly | MDN*. URL: https://developer.mozilla.org/en-US/docs/WebAssembly (visited on 06/2022).

[35] *What is eBPF? An Introduction and Deep Dive into the eBPF Technology*. URL: https://ebpf.io/what-is-ebpf (visited on 05/2022).

[36] *World Wide Web Consortium (W3C) brings a new language to the Web as WebAssembly becomes a W3C Recommendation.* Dec. 2021. URL: https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en (visited on 05/2022).

[37] Chris Wright et al. "Linux Security Modules: General Security Support for the Linux Kernel". In: *11th USENIX Security Symposium*. USENIX Association, Aug. 2002. URL: https://www.usenix.org/legacy/event/sec02/full_papers/wright/wright.pdf.