Summary

1. Introduction

Background

Software technologies are constantly evolving, with the development of new programming languages, libraries, and frameworks. Specifically, the world of the Web had a peculiar history, along which various aspects were focused.

The approaching end of Moore's Law[1] because of the physical limitations of technology put particular emphasis on the optimization of resources over an increment of raw performances, which are hard to enhance at the state of art.

The Web

In traditional Web development, client logic is managed through Javascript[2] code.

Javascript was born in 1995 as a dynamic language by nature. It was intended to be a simple scripting language to animate web pages and turn the browser into an application platform. This quest for simplicity drives Javascript's simple syntax and structure, making it one of the easiest languages to learn.

Its evolution along the decades lead Javascript to be one of the most popular programming languages all over the world, bringing it outside the browser to servers and native applications.

Taking a step back, for web applications to be executed, JS code needs to be transferred from the server to the client machine, just-in-time compiled, optimized, then executed.

This specifical sequence has been made necessary by the plethora of Javascript engines distributed with different web browsers and is not likely to be changed.

---

[1] Moore's law is the observation that the number of transistors in a dense integrated circuit doubles about every two years. Moore's law is an observation and projection of a historical trend. Rather than a law of physics, it is an empirical relationship linked to gains from experience in production.

[2] Often abbreviated as JS

WebAssembly

In this context, in 2015, WebAssembly was announced as a solution to these performance limitations of Javascript.

WebAssembly, often abbreviated as WASM, is a binary instruction format for a stack-based virtual machine. Wasm is designed to be a portable compilation target for programming languages, enabling deployment on the web for client and server applications.

Wasm is by nature faster to process and to analyse than Javascript, not as a substitution, but as an integration to Javascript: they're meant to work together inside the Web ecosystem. WebAssembly is particularly suitable to improve the performance of critical and computationally heavy parts of a Web application.

Since 2019, WebAssembly has been a W3C standard, maintained in collaboration with Mozilla, Microsoft, Google, and Apple, and is currently supported by the main modern browsers such as Chrome, Edge, Firefox, and Safari.

It's possible to write WebAssembly applications directly in a text format called WAT (WebAssembly Text), which can be compiled and translated to binary code.

Although it's not a usual practice, as this format is very close to Assembly language and lacks any sort of abstraction, resulting in an impractical way of writing code.

Being no more than a set of binary instructions, it's also possible to compile applications written in several programming languages into Wasm. Among these, C/C++, C#, and Rust stand out; expressly for them, some tools have been developed to compile code to Wasm. For instance, Emscripten was one of the first and most popular ones: it allows to compile even complex C/C++ applications to Wasm.

As an alternative, a new programming language called AssemblyScript was developed. The name recalls TypeScript, and the syntax does as well, but with

some key differences, for example AssemblyScript has strict static typing and allows no dynamic objects, which must be replaced by maps.

WebAssembly is executed directly inside Javascript Runtime thanks to some API’s which are developed ad hoc and incorporated into it.

Wasm was designed with a specific goal in mind: the efficient and high-performance execution of browser-based applications, without compromising compatibility and security.

The web platform can be thought of as having two parts:

- A virtual machine (VM) that runs the Web app's code.
- A set of Web APIs that the Web app can call to control web browser/device functionality and make things happen (DOM, CSSOM, WebGL, IndexedDB, Web Audio API, etc.).

WebAssembly is a different language from JavaScript, designed to complement and work alongside it, allowing web developers to take advantage of both languages' strong points:

- JavaScript is a high-level language, flexible and expressive enough to write web applications. It has many advantages — it is dynamically typed, requires no compile step, and has a huge ecosystem that provides powerful frameworks, libraries, and other tools.
- WebAssembly is a low-level assembly-like language with a compact binary format that runs with near-native performance and provides languages with low-level memory models such as C++ and Rust with a compilation target so that they can run on the web. (Note that WebAssembly has the high-level goal of supporting languages with garbage-collected memory models in the future.)

The different code types can call each other as required — the WebAssembly JavaScript API wraps exported WebAssembly code with JavaScript functions that can be called normally, and WebAssembly code can import and synchronously call normal JavaScript functions. In fact, the basic unit of

WebAssembly code is called a module and WebAssembly modules are symmetric in many ways to ES modules[3].

There are several key concepts needed to understand how WebAssembly runs in the browser. All these concepts are reflected 1:1 in the WebAssembly JavaScript API.

- Module: Represents a WebAssembly binary that has been compiled by the browser into executable machine code. A Module is stateless and thus, like a Blob, can be explicitly shared between windows and workers (via postMessage()). A Module declares imports and exports just like an ES module.
- Memory: A resizable ArrayBuffer that contains the linear array of bytes read and written by WebAssembly's low-level memory access instructions.
- Table: A resizable typed array of references (e.g., to functions) that could not otherwise be stored as raw bytes in Memory (for safety and portability reasons).
- Instance: A Module paired with all the state it uses at runtime including a Memory, Table, and set of imported values. An Instance is like an ES module that has been loaded into a particular global with a particular set of imports.

The JavaScript API provides developers with the ability to create modules, memories, tables, and instances. Given a WebAssembly instance, JavaScript code can synchronously call its exports, which are exposed as normal JavaScript functions. Arbitrary JavaScript functions can also be synchronously called by WebAssembly code by passing in those JavaScript functions as the imports to a WebAssembly instance.

---

[3] Short for ECMAScript module, it is a standard for organizing and sharing JavaScript code. It is a way to include and reuse code in JavaScript applications, providing better structure and maintainability.
An ES module is defined using the "**export**" keyword to export functions, objects, or values, and the "**import**" keyword to import them into another module. The modules are stored in separate files and have their own scope, which means the variables defined inside an ES module are not visible in other modules unless they are explicitly exported.
This provides a clear boundary between the different parts of a codebase, making it easier to manage and reuse code in a scalable way. ES modules are supported natively in modern web browsers and in Node.js, the popular JavaScript runtime environment.

Since JavaScript has complete control over how WebAssembly code is downloaded, compiled, and run, JavaScript developers could even think of WebAssembly as just a JavaScript feature for efficiently generating high-performance functions.

In the future, WebAssembly modules will be loadable just like ES modules (using <script type='module'>), meaning that JavaScript will be able to fetch, compile, and import a WebAssembly module as easily as an ES module.

A comparison between WebAssembly and JavaScript

Among the pros of the usage of Wasm over Javascript we have:

- Performance: thanks to static typing and ahead of time compilation. Code is highly optimized before reaching the browser, where it executes at near-native speed. Its binary files are considerably smaller than JavaScript's, resulting in significantly faster loading times.
- Cross-Platform Support: As stated earlier, one of the biggest drivers for WASM adoption is that developers can write code for the web in languages other than JavaScript and port existing applications over the web. Portability is a prominent feature of WebAssembly from the beginning, and this makes it worthwhile outside the browser powering efficient and performant applications on various operating systems (Windows, Linux, OSX, …) and architectures (Arm32/64, x64, RISC-V, …).
- Top-Notch Security**:** WebAssembly was built with security in mind. Its goal is to protect users from potential web insecurities while empowering developers to produce secure applications. WebAssembly provides a secure application experience by isolating module execution in a sandboxed environment while enforcing known browser security policies.

We also have some cons for the adoption of WebAssembly for browser applications:

- WASM is still in its early stages of development, and it will take some time before it builds the rich environment that JS had 20 years to create. WebAssembly, for example, currently lacks document object model (DOM) and garbage collection features, and it still relies on JavaScript for full platform access.
- Imperfect Security**:** Although WASM was built with security in mind, some features make it useful for malicious attackers. Furthermore, while the sandbox feature was designed to contain exploits, recent findings have proven this to be not entirely accurate. These concerns might be linked to the teething problems of a new language. WebAssembly may prove to be the solution that minimizes browser-based attacks as more features are developed.

Why WASI?

Since WebAssembly provides a new fast, scalable, secure way to run the same code across all machines, Mozilla developers decided to push Wasm beyond the browser.

In 2019 therefore they announced WASI, the WebAssembly System Interface, the begin of a new standardization effort.

WebAssembly was powerful but, at the time, it didn’t have a way to communicate with the underlying system, hence it needed a solid system interface to run outside the browser.

Just as WebAssembly is an assembly language for a conceptual machine, WebAssembly needs a system interface for a conceptual operating system, not any single operating system. This way, it can be run across all different OSs.

This is what WASI is: a system interface for the WebAssembly platform.

WASI: history, state of art and future

WASI started as a different name in 2016, CloudABI, just one year after the announcement of Wasm, as an attempt to create a portable system API.

After the release of WebAssembly MVP[4], developers started thinking more systematically about this concept and CloudABI was deprecated.

At that point, a lot of effort was driven to modularization, which was really important because several environments needed to use WebAssembly, but they may not be able to expose all of the interfaces that could be available under WASI. Thus, WASI specification was divided into multiple different modules in a way that platforms would support only the API's they needed.

When WASI first snapshot was released in 2020, it contained a bunch of the core interfaces, like clocks, filesystems, networking, and arguments, it was definitely insufficient, as it was non-modular.

At the time of this writing, in February 2023, WASI is still in development and Networking, being a critical aspect[5], lacks a few functionalities. Future versions will change based on experience and feedback with the first version, and add features to address new use cases. They may also see significant architectural changes.

What is exactly a system interface?

Before jumping right into the WebAssembly System Interface, it's appropriate to explain in a detailed manner what a System Inter

Every programming language, even low-level ones like the C language, cannot have direct access to system resources, like opening, creating files or accessing memory; these are too important for stability and security. If one program unintentionally messes up the resources of another, then it could crash the

[4] Minimum Viable Product
[5] We are going to see why in the next chapter

program. Even worse, if a program (or user) intentionally messes with the resources of another, it could steal sensitive data.

So, we need a way to control which programs and users can access which resources. People figured this out early on and produced a way to provide this control: protection ring security.

With protection ring security, the operating system basically puts a protective barrier around the system's resources. This is the kernel. The kernel is the only thing that gets to do operations like creating a new file or opening a file or opening a network connection.

The user's programs run outside of this kernel in something called user mode. If a program wants to do anything like open a file, it must ask the kernel to open the file for it. This is where the concept of the system call comes in. When a program needs to ask the kernel to do one of these things, it asks using a system call. This gives the kernel a chance to figure out which user is asking. Then it can see if that user has access to the file before opening it.

On most devices, this is the only way that your code can access the system's resources, through system calls. The operating system makes the system calls available. But if each operating system has its own system calls, wouldn't you need a different version of the code for each operating system? Fortunately, you don't.

How is this problem solved? Abstraction.

Most languages provide a standard library. While coding, the programmer doesn't need to know what system they are targeting. They just use the interface.

Then, when compiling, your toolchain picks which implementation of the interface to use based on what system you're targeting. This implementation uses functions from the operating system's API, so it's specific to the system.

This is where the system interface comes in. For example, printf being compiled for a Windows machine could use the Windows API to interact with

the machine. If it's being compiled for Mac or Linux, it will use POSIX[6] instead. This poses a problem for WebAssembly, though.

With WebAssembly, you don't know what kind of operating system you're targeting even when you're compiling. So, you can't use any single OS's system interface inside the WebAssembly implementation of the standard library. WebAssembly needs a system interface for a conceptual operating system, not a real operating system. But there are already runtimes that can run WebAssembly outside the browser, even without having this system interface in place.

The first tool for producing WebAssembly was Emscripten. It emulates a particular OS system interface, POSIX, on the web. This means that the programmer can use functions from the C standard library (libc).

To do this, Emscripten created its own implementation of libc. This implementation was split in two: part was compiled into the WebAssembly module, and the other part was implemented in JS glue code. This JS glue would then call into the browser, which would then talk to the OS.

The Goals

This document aims to analyse WASI in all its aspects and to provide the reader a full overview of the technology.

Being in its initial stages, it will be difficult to be accurate on the potentialities and future developments, but this review will attempt to be as accurate and as scientific as possible.

Particularly, it will try to consider the advantages and the trade-offs of adapting WASI for future native cross-platform applications development.

[6] **P**ortable **O**perating **S**ystem **I**nterface for Uni**x** is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

2. WASI

Design principles

There are two important principles that are baked into WebAssembly: portability and security. These key principles need to be maintained as we move to outside-the-browser use cases. Portability makes it easier to develop and distribute software, and providing the tools for hosts to secure themselves or their users is an absolute must.

POSIX provides source code portability. You can compile the same source code with different versions of libc to target different machines. But WebAssembly needs to go one step beyond this. We need to be able to compile once and run across a whole bunch of different machines. We need portable binaries. This kind of portability makes it much easier to distribute code to users.

Now let's jump into what concerns security. When a line of code asks the operating system to do some input or output, the OS needs to determine if it is safe to do what the code asks.

Operating systems typically handle this with access control that is based on ownership and groups. For example, the program might ask the OS to open a file. A user has a certain set of files that they have access to.

When the user starts the program, the program runs on behalf of that user. If the user has access to the file, then the program has that same access, too. This protects users from each other in a multi-user environment. Systems now are usually single user, but they are running code that pulls in lots of other, third-party code, sometimes of unknown trustworthiness. Now the biggest threat is that the code that you yourself are running will turn against you.

If a library has access to do anything on your system - for example, open any of your files and send them over the network - then a code that uses this library can cause a lot of damage.

WebAssembly's way of doing security is different. WebAssembly is sandboxed.

This means that code can’t talk directly to the OS. But then how does it do anything with system resources? The host (which might be a browser or might be a wasm runtime) puts functions in the sandbox that the code can use.

WASI is built using capability-based security principles. Access to external resources is always represented by handles, which are special values that are unforgeable[7].

WASI is also aiming to have no ambient authorities, meaning that there should be no way to request a handle purely by providing a string or other user-controlled identifier providing the name of a resource. With these two properties, the only ways to obtain access to resources are to be explicitly given handles, or to perform operations on handles which return new handles.

Note that this is a different sense of "capability" than Linux capabilities or the withdrawn POSIX capabilities, which are per-process rather than per-resource.

The simplest representation of handles are values of reference type. References in wasm are inherently unforgeable, so they can represent handles directly.

Some programming languages operate primarily within linear memory, such as C, C++, and Rust, and there currently is no effortless way for these languages to use references in normal code. And even if it does become possible, it's likely that source code will still require annotations to fully opt into references, so it won't always be feasible to use. For these languages, references are stored in a table called a c-list. Integer indices into the table then identify resources, which can be easily passed around or stored in memory. In some contexts, these indices are called file descriptors since they're similar to what POSIX uses that term for. There are even some tools, which make this fairly easy.

---

[7] One of the key words that describes capabilities is *unforgeable*.
A pointer in C is forgeable, because untrusted code could cast an integer to a pointer, thus *forging* access to whatever that pointer value points to.
MVP WebAssembly doesn't have unforgeable references, but what we can do instead is just use integer values which are indices into a table that's held outside the reach of untrusted code. The indices themselves are forgeable, but ultimately the table is the thing which holds the actual capabilities, and its elements are unforgeable. There's no way to gain access to a new resource by making up a new index.
When the reference-types proposal lands, references will be unforgeable, and will likely subsume the current integer-based APIs, at the WASI API layer.

Integer indices are themselves forgeable, however a program can only access handles within the c-list it has access to, so isolation can still be achieved, even between libraries which internally use integer indices, by withholding access to each library's c-list to the other libraries. Instances can be given access to some c-lists and not others, or even no c-lists at all, so it's still possible to establish isolation between instances.

There are two levels of capabilities that we can describe: static and dynamic.

The static capabilities of a wasm module are its imports. These essentially declare the set of "rights" the module itself will be able to request. An important caveat though is that this doesn't consider capabilities which may be passed into an instance at runtime.

The dynamic capabilities of a wasm module are a set of Boolean values associated with a file descriptor, indicating individual "rights". This includes things like the right to read, or to write, using a given file descriptor.

POSIX normally allows processes to request a file descriptor for any file in the entire filesystem hierarchy, which is granted based on whatever security policies are in place. This doesn't violate the capability model, but it doesn't take full advantage of it. Some capability-oriented systems prefer to take advantage of the hierarchical nature of the filesystem and require untrusted code to have a capability for a directory to access things inside that directory.

Sockets aren't naturally hierarchical though. This is an area that isn't yet implemented, but developers will need to decide how sockets capabilities will work.

In CloudABI, users launch programs with the sockets they need already created. That's potentially a starting point, which might be enough for simple cases.

This capability-based security allows the host to limit what a program can do on a program-by-program basis. It doesn’t just let the program act on behalf of the user, calling any system call with the user’s full permissions.

Just having a mechanism for sandboxing doesn’t make a system secure in and of itself - the host can still put all the capabilities into the sandbox, in which

case we’re no better off - but it at least gives hosts the option of creating a more secure system.

Another important design principle behind WASI is its interposition[8]. Interposition in the context of WASI interfaces is the ability for a WebAssembly instance to implement a given WASI interface, and for a consumer WebAssembly instance to be able to use this implementation transparently. This can be used to adapt or attenuate the functionality of a WASI API without changing the code using it.

In WASI, we envision interposition will primarily be configured through the mechanisms in the module linking link-time virtualization. Imports are resolved when a module is instantiated, which may happen during the runtime of a larger logical application, so we can support interposition of WASI APIs without defining them in terms of explicit dynamic dispatch mechanisms.

Compatibility with existing applications and libraries, as well as existing host platforms, is important, but will sometimes conflict with overall API cleanliness, safety, performance, or portability. Where practical, WASI seeks to keep the WASI API itself free of compatibility concerns, and provides compatibility through libraries, such as WASI libc, and tools. This way, applications which don't require compatibility for compatibility' sake aren't burdened by it.

Portability is important to WASI; however, the meaning of portability will be specific to each API.

WASI's modular nature means that engines don't need to implement every API in WASI, so we don't need to exclude APIs just because some host environments can't implement them.

It’s worth spending a little time taking a detailed look at file system. It’s not shared among the different modules, instead a compatibility layer will be used. The host is not the one providing the file system, instead the module itself is the one virtualizing its own file system and the files to be accessed will be in

[8] Interposition is sometimes referred to as "virtualization", however we use "interposition" here because the word "virtualization" has several related meanings.

the linear memory of the wasm module. This means that we don’t have that global shared mutable state problem that the filesystem introduces.

Even though in the source code we have files treated as they would be on a native support, under the hood, they use WASI I/O types that will give them full portability.

However, these virtualizations will introduce some performance inefficiencies including large file sizes for wasm modules. In case you want full portability and efficiency at the same time, you will have a different API in the source code, the WASI I/O API. In this case, some changes to the source code will be necessary: instead of passing files around, you will be passing those I/O types.

With these, the developer no longer needs to think in terms of files, they become a pure I/O stream of bytes. And this means the code can really run anywhere, because every system can represent these basic primitive types. This way, the overhead of the per-module file system and the issues of a global shared mutable state were eliminated.

3. Performance analysis

4. Sitography

Articles

- Clark L., *Standardizing WASI: A system interface to run WebAssembly outside the web*, https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, visited on 09/02/2023.
- Salinas Gardón D., *Webassembly vs. JavaScript: How Do They Compare*, https://snipcart.com/blog/webassembly-vs-javascript, visited on 10/02/2023.

Sites

- https://en.wikipedia.org/wiki/POSIX, visited on 12/02/2023
- https://en.wikipedia.org/wiki/Moore%27s_law, visited on 12/02/2023
- https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts, visited on 12/02/2023
- https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-capabilities.md, visited on 12/02/2023
- https://github.com/webassembly/wasi, visited on 12/02/2023