MITRE

MTR210290
MITRE TECHNICAL REPORT

WebAssembly: Current and Future Applications

Dept. No.: L181
Project No.: 01MSRF21

Montgomery, AL

Authors:
David Bryson
Justin Brunelle

August 2021

This page intentionally left blank.

Abstract

This report introduces WebAssembly (Wasm) technology, with a goal of educating potential users and motivating further exploration and use. To that end, it includes an introduction to Wasm and how it works. A survey of the current state of the art highlights a broad spectrum of Wasm capabilities, and its growing ecosystem of tools and support. It covers details of the Wasm security model and outlines important security considerations as well as recommendations for further research. Finally, the report provides several real-world examples of Wasm use in both client and server-side environments.

Overall, due to Wasm's open nature, broad industry support and use, and growing ecosystem of tools and language support, we envision Wasm as playing an increasingly important role in a wide range of applications. Therefore, we highly recommend exploring its potential use as well as continued monitoring of developments in the technology.

This page intentionally left blank.

Executive Summary

Wasm is a compact binary format and open specification designed to execute code in a fast, safe, and portable manner. Originally designed to improve the performance of client-side web applications, Wasm has attracted growing interest and broad application across several different domains, from in-browser graphics to server-side blockchain smart contracts.

The open nature of the Wasm specification has led to unprecedented collaboration on the development of the Wasm runtime. Every major web browser now includes a Wasm runtime enabled by default. CPU-intensive applications such as games and CAD tools that were previously impractical for use in a traditional JavaScript browser environment have been successfully ported to the web for use via Wasm.

Outside the browser, Wasm is often employed in the emerging blockchain space to implement sophisticated smart contracts. Cloud providers are exploring Wasm use as a flexible approach for developing microservices as well as running untrusted code. In addition, open-source communities are working to advance the state of the art, as many envision that Wasm will play a growing role beyond the browser in server-side computing. In fact, Solomon Hykes, the founder of Docker, tweeted: *"If WASM+WASI existed in 2008, we wouldn't have needed to create Docker"* [31].

The Wasm security model provides a sandboxed execution environment that can reduce the impact of malicious and/or faulty applications on a host environment. However, Wasm cannot make unsafe code safe. Therefore, developers should follow best practices to minimize unsafe code. They should consider using languages such as Rust with Wasm when security and performance are important.

Overall, due to Wasm's open nature, broad industry support and use, and growing ecosystem of tools and language support, we envision Wasm as playing an increasingly important role in a wide range of applications. Therefore, we highly recommend exploring its potential use as well as continued monitoring of developments in the technology.

Table of Contents

List of Figures

This page intentionally left blank.

1 What Is WebAssembly?

WebAssembly (Wasm) is a compact binary format and open specification that enables portable, high-performance applications and executes code in a fast, safe, and portable manner. First introduced in 2015, it was originally designed to run within a web browser to improve the performance and safety of client-side browser applications. Today, Wasm can be used in both browser and non-browser environments and has attracted growing interest and broad application across several different domains, from in-browser graphics to server-side blockchain smart contracts.

The overall design of Wasm must meet several goals outlined in the Wasm specification [1], which state that Wasm must be:

- Fast: can often run as fast as native code compiled for a specific platform
- Safe: code is validated and executes in a memory-safe, sandboxed environment to minimize and isolate faulty or malicious code.
- Hardware-independent/portable: can be compiled on all modern architectures such as desktop, mobile devices, and embedded systems alike.
- Language-independent: supports several popular programming languages such as C, C++, Rust, Go, Java, and more
- Platform-independent: can be embedded in browsers, run as a stand-alone virtual machine (VM), or integrated with other languages and environments
- Compact: uses a binary format that is fast to transmit by being smaller than typical text or native code formats
- Open: The World Wide Web Consortium (W3C) defines and maintains the WebAssembly core specification.

1.1 What Problems Does It Solve?

Performance demands on web-based client-side applications have increased. Applications such as games, data analytics, music streaming, image/video editing, and more can benefit by taking advantage of modern desktop and mobile architectures. However, the native browser language, JavaScript, is often not ideal (in either performance or security) for these types of applications.

Wasm is designed to provide a more secure, performant environment for client-side applications. To be fair, other technology stacks previously attempted to solve the same problems, namely Flash and Java Applets. Wasm improves over other approaches thanks to the characteristics listed below:

- Broad industry support: Previous solutions were owned and controlled by a single company. This left the technology under the control of the company, and often forced the government to continue its contractual relationship with the existing vendor, regardless of that vendor's performance. Wasm is based on an open specification developed under theW3C and is supported by every major browser vendor.

- Language agnostic: Flash required the use of ActionScript while Java Applets used the Java programming language. Wasm applications can be written in many different languages.
- No plugin required: Early approaches required users to add heavyweight/slow plugins to the browser. Wasm is included as part of the browser platform, and has a smaller overall runtime footprint.
- True to the Web: Wasm provides controlled access to the browser environment around it. It can call and be called by JavaScript, interact with the Document Object Model (DOM), and make use of Cascading Style Sheets (CSSs) to style the application. This represents a big improvement over previous approaches that often operated in isolated environments and did not integrate well with the rest of the application.
- Safety: Wasm improves safety by piggybacking on the browser's security model, while providing additional safety mechanisms such as memory isolation and control-flow integrity.

Wasm represents the next step in the evolution of secure, performant client-side web applications. We expect Wasm to improve security and performance over its predecessors while staying true to the web environment within which it operates.

1.2 How Does It Work?

The Wasm specification defines a standardized, portable binary code format. Programming languages such as C++ or Rust can be translated into the Wasm format, resulting in a binary file that host environments know how to decode, validate, and execute. The list below shows a typical flow of generating a Wasm application:

1. A developer writes an application in a traditional programming language that enable the generation of Wasm code. A growing number of languages support Wasm output, including C++, Rust, Java, .NET, and Typescript, to name a few.
2. The programming language designates Wasm as the target output during the compilation step. For example, in the Rust programming language WebAssembly is specified as the build target. This results in a Wasm-compliant binary file.
3. The resulting binary file is distributed and executed in any Wasm-compliant host (container) environment.

Given a Wasm binary, a compliant host environment runs through the following phases to execute the code:

1. First, the host environment decodes the binary and transforms it into machine language for the target architecture.
2. Next, the code is validated to ensure it is well formed and safe. This includes type checking as well as verifying that the inputs and outputs of operations are consistent.
3. Finally, the code begins the execution phase. First it instantiates the code, creating the application's local memory, and invokes the code's internal start function if specified. At this point the calling environment can execute any exported function.

At a high level, it is easier to think of Wasm as a conceptual machine, as it is not bound to a specific architecture. This is important because when running applications across the internet, the application does not know ahead of time which architecture the browser is running on (x86, ARM). Wasm solves this problem by removing the specifics of the architecture from its binary.

In essence, Wasm is a form of assembly language. The binary format serves as an intermediate representation (IR) of the original code (see Figure 1), which can then be translated into a specific machine's language. For example, when a programmer writes an application in the Rust language, the Wasm compiler translates the code to a Wasm-compliant binary format. The resulting binary can then be distributed with the application and downloaded across the internet. Once downloaded, the Wasm runtime container compiles the code into the target machine's assembly code.

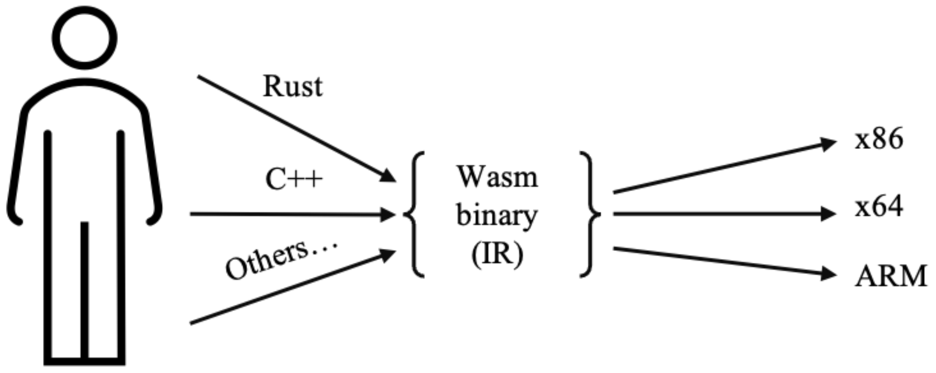

Figure 1. WASM Intermediate Representation (IR)

To readers already familiar with compiler toolchains, the Wasm approach may seem very similar to the way LLVM [2] works. LLVM is a popular toolchain used by many programming languages (such as Rust) to generate an IR format that can then be compiled to different target architectures. Thus, a natural question arises: why not just use LLVM "bitcode" format versus a custom Wasm binary format? The reason is that in essence Wasm has several goals and requirements that differ enough from LLVM [3] to justify its own format. However, many Wasm compilers make use of LLVM in the compilation chain to generate code. For example, Emscripten [4] and Rust use LLVM to compile code to the Wasm format.

1.3 Compiling to Wasm

To give a better idea of how the application flow described above works, this section walks the reader through a simple example of creating a Wasm module and compiling it for use in a non-browser JavaScript environment (i.e., Node.js). Wasm can be used in other environments as well. In fact, a growing number of tools for many languages automate the process of generating a Wasm binary. For example, the Rust programming language has a built-in tool chain for writing code in Rust and compiling to a Wasm binary that can be ran in several different environments: most major web browsers, Node.js, Python, Java, Go, and more.

In addition to requiring the ability to generate a Wasm binary from many popular traditional languages, the Wasm specification also outlines a text format language [5]. The Wasm text format, often referred to as 'wat,' is a textual representation of the Wasm binary format. It functions as a human-readable intermediate format useful for experimenting and gaining a deeper knowledge of how Wasm works. Wat is based on the S-Expression [6] language. Developers can write a complete Wasm application in the wat format. However, most developers will likely use a traditional programming language that supports Wasm generation as part of its tool chain.

Figure 2 shows an example of a function for adding two numbers and returning the result defined in the wat format.

```
(module
  (func $add (param $lhs i32) (param $rhs i32) (result i32)
   local.get $lhs
   local.get $rhs
   i32.add)
  (export "add" (func $add))
)
```

Figure 2. Example Function Written in the Wasm Text Format

The figure illustrates that the unit of code in Wasm is a module. Each module may have many functions. In the wat format a function signature starts with 'func' and then the name followed by the input parameters. In this example, the function 'add' can accept two 32-bit integers and will return a 32-bit integer.

The body of the function includes the logic to retrieve the input parameters, and call i32.add, which is built-in logic available to add 32-bit integers within Wasm. The 'local.get' calls push the input parameters onto the stack for use by the program. Wasm is a stack-based VM, meaning values are pushed on and popped off the stack for use. In this example, the two input parameters are pushed on the stack, and the i32.add call pops them off the stack, adds them together, and returns the result. Finally, the code exports the function with the name 'add'. Functions must be exported for use outside the module.

Next, the user must transform the code to a valid Wasm binary. Doing that requires use of the 'wat2wasm' tool. It takes the text format and compiles it into a valid binary that can be loaded and used in any Wasm-compliant runtime environment. Since the text (wat) format is often used for learning and experimentation, there are many online interactive tools that will automate the process. This example uses the wat2wasm demo tool from the WebAssembly group [7]. Running wat code through the tool results in the annotated Wasm binary shown in Figure 3:

```
0000000: 0061 736d                ; WASM_BINARY_MAGIC
0000004: 0100 0000                ; WASM_BINARY_VERSION
; section "Type" (1)
0000008: 01                       ; section code
0000009: 00                       ; section size (guess)
000000a: 01                       ; num types
; func type 0
000000b: 60                       ; func
000000c: 02                       ; num params
000000d: 7f                       ; i32
0000010: 7f                       ; i32
...
```

Figure 3. Snippet of an Annotated WASM Binary

Note that the binary is laid out in sections (section 1.4.2 contains more information on their contents) and is independent of the host architecture. The output file for the Wasm binary uses the file extension '.wasm'.

With the code in the Wasm binary format users can now load and execute the 'add' function in any Wasm runtime environment (browser or non-browser host). Figure 4 shows an example of the code (with comments) needed to load and execute the add function in a non-browser, Node.js environment:

```
// Load the wasm binary from file
const wasm = await readFile('./add.wasm');
// parses binary into module structure
const module = await WebAssembly.compile(wasm);
// Load the module creating an instance of the Wasm runtime
const instance = await WebAssembly.instantiate(module);
// Call the exported function
const result = instance.exports.add(3, 2);
console.log(`The answer is: ${result}`);
```

Figure 4. JavaScript Example of Loading and Running a WASM Binary

1.4 Binary Format

Wasm programs are organized into *modules*, which are the unit of deployment, loading, and compilation [8]. A module contains several sections that contain the definitions that the runtime will use to determine how to execute code. When a runtime module loads a binary it is parsed/decoded into these sections to create an instance of the module for the execution environment. The binary itself is simply a linear sequence of bytes in the little-endian format.

1.4.1 Types

Wasm uses four primitive types:[1]

- 32-bit integer
- 64-bit integer
- 32-bit floating point (IEEE 754-2008)
- 64-bit floating point (IEEE 754-2008).

Complex types such as strings or objects are built from the four basic types, but the programming language's compiler used to create the Wasm module almost always performs this task. In other words, mapping basic types to more complex types occurs automatically.

1.4.2 Sections

The binary specification [9] has a total of 12 sections that organize the modules' content:

[1] Technically, Wasm also has a "reference" type used to reference objects in the runtime table. However, they are mostly opaque.

1. **Custom**: used for debugging information or third-party extensions and ignored by the Wasm semantics.
2. **Type**: contains a list of the unique function signatures used by the module. For example, (i32, i32 -> i32) is the signature for the 'add' function created earlier and depicted in Figure 2. The position of the signature in the list becomes the function's unique index within the module.
3. **Import**: declares any external modules or functions that are imported for use in the current module. Wasm code can reuse functions and code from other Wasm modules.
4. **Function**: declares the list of indexes for each function. The index is used to look up the actual function code defined in a separate section of the binary (see 11 below).
5. **Table**: lists references used to access indirect callable code via an index. For example, function calls cannot be directly addressed. Instead, they are indexed in the table and accessed via the index. The target index must be a valid entry (index) in the table.
6. **Memory**: defines the memory used by the module. Memory in Wasm is a linear array of raw uninterpreted bytes. Each module has its own isolated memory, as Wasm does not have access to the memory in its host's environment. Memory is allocated in chunks of 64kb (page size).
7. **Global**: lists mutable or immutable global variables used by the module. A global are like a static variable in other languages.
8. **Export**: contains a list of the functions in the module that will be available to the host. For example, the 'add' function was exported from the example above. which enabled the ability to call it from the JavaScript environment.
9. **Start**: contains an optional index of a function that may be ran when the module is started
10. **Element**: can be used to initialize the contents of Tables imported from the host or defined in the Table section.
11. **Code**: represents the bulk of the content in a module. This is where the logic for the functions is defined. The bodies are laid out in the same order as their indexes in the Function section.
12. **Data**: stores a list of data segments that can be used to initialize memory. By default, memory is initialized to zero bytes.

A module does not have to include all the sections listed above. For example, the minimal entries required for the 'add' module would be the Type, Function, and Code sections.

The calling application's programming interface (API) parses a binary and transforms it into a module instance. The instance contains a "store" that is a runtime representation of the sections described in the binary. Access to the store is restricted to the instance of the owning module.

1.5 What Makes Wasm Faster Than JavaScript?

Wasm evolved due to the need to improve the performance and safety of browser-based applications. Importantly, Wasm was designed to work *alongside* JavaScript in the browser, not to serve as a complete replacement. JavaScript is the lingua franca of the internet and will remain so for the foreseeable future. Therefore, one of the primary goals of Wasm is to improve performance-critical parts of traditional JavaScript applications. To give readers a better idea of how it does this, the list below summarizes the process by which JavaScript runs in the browser as compared to Wasm [10].

- **Fetching code**: When a browser page containing application logic is loaded over the network, the first thing it must do is download the associated code. The compact code format of Wasm makes this part of the process faster compared to the more verbose JavaScript format. Even when JavaScript uses compaction, the Wasm code remains smaller.
- **Parsing**: Once the code is downloaded, JavaScript code must first be parsed into an Abstract Syntax Tree and then transformed into the IR format. Wasm does not require this transformation, as it is already in a highly compact and efficient binary IR format, so parsing and validation are much faster and more efficient.
- **Compile/Optimization**: Once parsed, many modern JavaScript runtime instances use a just-in-time compiler (JIT) to compile and optimize the JavaScript code to improve performance. Wasm does not have to do this, as it is already close to machine code and makes use of explicit types. Therefore, this phase of preparing to execute the code happens much faster with Wasm.
- **Execution**: Since Wasm is designed as a compiler target, the instruction set is closer to what the machine requires. Therefore, depending on the specific logic,[2] the code can execute much faster than the comparable JavaScript code generated by the JavaScript JIT.
- **Garbage collection**: Developers do not need to actively manage memory in JavaScript. The JavaScript runtime automatically monitors memory allocations and releases memory based on the garbage collection algorithm. Wasm does not yet have full support for garbage collection. Therefore, developers are responsible for memory management, which can make performance more predictable and consistent – especially in larger applications.

[2] Not all WebAssembly code is naturally faster than JavaScript. Again, WebAssembly is designed for performant critical parts of the code.

2 State of the Art

The popularity and application of Wasm has grown considerably over the past few years. There are several ways to run a Wasm application along with a growing ecosystem of libraries. A Wasm application requires an execution environment (host) to run code. This environment may be a browser, a standalone runtime environment, or a runtime embedded in an existing application.

2.1 Browser

All major browsers currently support Wasm, thanks to unprecedented collaboration among browser makers to bring high-performance applications to the web [11]. As of 2021, the following browsers support the Wasm specification [12] and provide a runtime environment enabled by default, with no additional plugins or configuration needed:

- Microsoft Edge
- Firefox
- Chrome
- Safari
- Opera
- Brave.

Browsers supporting Wasm provide a default JavaScript API that can load and instantiate Wasm binaries. This API also allows Wasm to interact with the host environment: JavaScript can call Wasm functions, and Wasm can interact with JavaScript. However, these interactions are constrained by both the hosting environment as well as the imported and exported functions defined by the Wasm code. Additionally, each Wasm application has its own isolated memory for safety.

Once Wasm code is initialized, any functions exported by the binary can be called through browser-based JavaScript. This allows performance-critical parts of a client-side application to be implemented in Wasm and work seamlessly alongside the rest of the JavaScript application.

Most applications use a combination of Wasm functions and traditional JavaScript code to implement client-side applications. However, developers can create an entire web application with just Wasm. In fact, some of the first successful examples resulted from porting traditional C++ games to Wasm for use in the browser. Among the growing ecosystem of tools for converting and compiling code to Wasm for use in the browser, some of the most popular are:

- **Emscripten:** A compiler toolchain that can convert any language that supports LLVM into Wasm for use on the Web, or a standalone Wasm runtime. Emscripten supports many popular APIs such as SDL and pthreads and can convert native OpenGL into WebGL for use in the browser. Many large codebases such as the Unity and Unreal game engines have used Emscripten to convert existing C/C++ code to Wasm [4].
- **Binaryen:** Another C++ compiler for Wasm. It is often integrated with Emscripten to provide a complete compiler toolchain. Several open-source projects use Binaryen to compile code from a broad range of programming languages into Wasm.

- **Wasm-bindgen:** A Rust library that facilitates interaction between JavaScript and Wasm modules. With wasm-bindgen a programmer can write performance-critical code in Rust and call it from JavaScript. The list below provides some features of the library [13]:
    - Export Rust code for use in the browser via JavaScript
    - Import JavaScript functionality to Rust, such as DOM manipulation, console logging, and more
    - Automatically generate JavaScript and TypeScript bindings for Rust code.
- **Yew:** A pure Rust framework for creating browser-based web applications entirely in the Rust programming language. It offers a component-based framework for creating interactive user interfaces and supports interoperability with existing JavaScript code [14].
- **Blazor:** A Microsoft .NET framework for building interactive client-side user interfaces. It is a feature of ASP.net that extends .NET for building web applications. Blazor runs .NET in the browser using Wasm [15].

2.2 Standalone/Embedded

While Wasm was originally designed for client-side web applications, interest in using it outside the browser has increased. Open-source communities such as the Bytecode Alliance [16], as well as others, are working to establish a platform that can run Wasm on any infrastructure.

A standalone runtime environment provides a sandboxed container that can run Wasm code outside the browser across different platforms/operating systems (OSs). By default, the runtime has no access to host resources such as files, network, etc.; the container focuses solely on pure computation of the Wasm code. This is an ideal environment for running untrusted code. However, in some cases an application may need access to host resources; therefore many standalone runtimes are adding support for the emerging WebAssembly System Interface specification (WASI) [17].

WASI is a set of standardized APIs being designed through the W3C Wasm community. WASI will provide portable system APIs that can give users access to host resources such as files and networking in a controlled, sandboxed manner. WASI's core design follows the concept of capability-based security, in which access to external sources is only granted when the caller possesses the appropriate capability.

Many open-source Wasm runtimes also provide the ability (via an API) to directly embed the runtime within a supported programming language. This means Wasm functionality can be incorporated into an existing codebase. For example, many emerging blockchain platforms use embedded Wasm runtimes to provide domain specific language support for authoring business logic (e.g., Smart Contracts) on a the blockchain.

Some of the more popular open-source standalone runtimes and APIs for Wasm include:

- **Wasmtime**: A small, efficient, standalone Wasm runtime built in Rust. Developed by the Bytecode Alliance community, with members such as Intel and Google, it offers several tools for working with Wasm and a runtime that can be run completely in standalone mode or embedded within an existing application [18].

- **Wasmer**: Another Rust based open-source runtime [19] with support and APIs for integrating and generating Wasm from several programming languages to include Python, Go, Java, PHP, Ruby and more.
- **WebAssembly Micro Runtime** (WAMR): A standalone runtime designed for embedding in constrained environments such as those often used in an Internet of Things (IoT) environment. WAMR boasts features such as ahead of time (AoT) and just in time (JIT) compilation and has a small runtime footprint of less than 100k bytes [20].

3 Security

One of the main goals of Wasm is improved safety. Wasm is designed to help reduce the effects of both faulty and potentially malicious programs. This section examines what makes Wasm safe and some potential Wasm vulnerabilities.

3.1 Safety Features

A Wasm module executes in a sandboxed environment that is under the control of the Wasm VM, not the host operating system. By default, Wasm code does not have access to host resources such as host APIs, file operations, network connections, etc. The developer must explicitly import access to any external host calls. This prevents Wasm code from arbitrarily calling host functions. Programming languages supporting Wasm have their own syntax for specifying these imports. Additionally, any Wasm binary can be converted to the wat format and visually inspected for these imports.

Wasm protects how it accesses function calls. Unsafe programming languages such as C\C++ or unsafe calls in Rust directly reference compiled functions via a memory address. Programs can technically jump to any address in memory. Address (pointer) arithmetic can pose problems; nothing forces these addresses to be valid or point to the correct position. The ability to jump to any memory address is a frequent source of vulnerabilities.

In Wasm, functions are not represented by a memory address. In fact, the actual memory address of the function is hidden from the Wasm program and controlled and managed by the Wasm VM. Instead, Wasm uses an index to look up the function in the predefined function section (see Section 1.4.2) that maps an index to the function located in the Code section of the binary. This prevents the program from arbitrarily calling a random memory address. Wasm only has to perform a simple check of the index to ensure the call is within bounds of the function table.

The same is true for any function imported from the host. Wasm uses the Table section of the binary to control the actual address of an imported function. Like a local function, the imported function is called via an index located in the Table section that is under control of the VM. The program does not have access to the real memory address of the function in the host's memory.

The way it uses memory represents another security feature of Wasm. Each Wasm program (module) has its own isolated memory that is separate from the hosting environments memory. This memory is represented as a linear, continuous, array of untyped bytes.

Wasm uses linear memory for allocating heaps and moving data back and forth between the application and host. In traditional environments using languages such as C\C++, memory can be directly accessed by address and, as with function calls this can often be a source of vulnerabilities. Wasm minimizes these potential problems by preventing the VM opcodes from having direct access to host memory addresses. Instead, the Wasm runtime references an index or offset into its local linear memory for the heap. This simplifies validating access to memory because Wasm only needs to check the bounds of the array. If that check fails, Wasm generates a trap (or error) resulting in termination of the program. Using this approach, a Wasm program cannot access host memory, which helps reduce the potential attack surface.

Key points made in this section include:

- A Wasm program is isolated from the host operating system. It cannot access host functionality unless the program developer explicitly imports it into the program. Imported host functions can be easily revealed via the text format of a Wasm binary.
- Function calls (direct or indirect) do not have access to host memory addresses. Functions are referenced by indices controlled by the Wasm VM through dedicated internal structures.
- Each Wasm module has its own linear memory that acts like heap memory. As with function calls, a Wasm program cannot directly access memory addresses of the host.

3.2 Potential Vulnerabilities

While the memory model used by Wasm provides some security benefits, it can also lead to a false sense of security. Research [21] has shown the current memory approach can potentially expose traditional vulnerabilities such as stack overflows and overwriting module memory. This occurs primarily because the Wasm memory is both readable and writable; Wasm has no read-only memory like that found in traditional binaries.

For example, Wasm only works with four types of primitives (see Section 1.4.1). When the host and the Wasm program must exchange more complex types such as a string value, they must orchestrate the exchange using linear memory. They often do this by specifying the memory index and length of the string to determine the region of memory of the value so the callee can read and process the bytes. An *unsafe* C\C++ program that does not perform a bounds check when writing a string to a buffer could cause an overflow in Wasm memory; for more details on potential attacks see [21]. Overall, Wasm cannot make unsafe code safe. Like any other environment, bad code can lead to problems. Using memory-safe languages such as Rust (which has first-class support for Wasm) can help mitigate potential problems.

As of 2021, several efforts seek to further harden Wasm's security. A few examples of those efforts include:

- Improving linear memory with traditional approaches such as stack canaries [22] and garbage collection to control heap growth
- Interface Types [23] to provide a standard approach to communicate/translate complex types across languages and between the host and Wasm module
- WASI [17] provides a general-purpose API to support interacting with a host in non-web browser environments.

It is important to point out that Wasm is still evolving and growing in popularity. Wasm runtimes and the associated specifications improve constantly. More research is necessary to track the state of the art and evaluate how well the latest runtime environments address security. Additionally, users need more tools and techniques to detect Wasm code that is specifically designed with malicious intent.

4 Examples of Use

This section of the report highlights the growing interest and broad applicability of Wasm in both browser and non-browser environments.

4.1 Browser

Improving the performance of browser applications was one of the original driving forces behind the creation of Wasm. Tools such Emscripten and Binaryen (Section 2.1) provide the ability to port existing codebases in languages such as C\C++ to the browser. Several stand-alone applications have been successfully ported to Wasm for use in web browser environments, for example:

- AutoCAD is popular design tool used to create 2D and 3D drawings. AutoCAD used Emscripten to port its large, 30+ year-old C/C++ codebase to Wasm to provide a complete in-browser AutoCAD environment.
- Applications that perform graphics through OpenGL APIs are prime candidates for Wasm use, as generating 2D and 3D graphics can be computationally expensive operations. Google ported Earth to Wasm to provide performant, cross-browser support of its 3D application. Unity3D did the same for its game engine, providing the ability to use Wasm as the output format for the Unity WebGL build target.
- Users have a growing need to visualize and manipulate large real-time data sets in the browser. Popular open-source projects such as Perspective [24] use Wasm for fast, streaming in-browser queries of data as well as creating framework-agnostic user interface components. Pyodide [25] brings Python and several of Python's popular scientific libraries, such as numpy, matplotlib, scikit-learn, and more to the browser via Wasm.
- In-browser cryptographic operations, such as digital signatures, encryption, and more, benefit from Wasm. Developers can port existing mature libraries to Wasm to provide more performant and secure functionality. Even more advanced emerging techniques such as Zero Knowledge Proofs (ZKP) [26] have found their way into the browser via Wasm. For example, ZkInterface [27] provides Wasm modules for several different ZKP techniques.

Overall, Wasm has the potential to improve the performance and security of next-generation client-side applications. Its ability to work alongside JavaScript means developers can continue to use popular tools for development of user interfaces, while employing Wasm for the more performance- and security-critical parts of the application.

4.2 Cloud

Wasm has also attracted growing interest related to cloud environments. Using Wasm for microservices allows developers to author and share discrete functions in any language.

Companies such as Fastly and Cloudflare use Wasm for serverless edge applications. This permits cloud providers to increase their security posture by running untrusted code in isolated environments while also improving performance and developer options. Many consider Wasm as the improved next generation of Docker-like containers.

Wasm has also emerged as a safe and secure environment for authoring various platform extensions. For example, Solo.io uses Wasm to provide developers with a more flexible approach to writing filters and transformation extensions for the Envoy proxy. Previously this was done with C++, which limited the developer base. By using Wasm, developers can use a language of their choice. Solo.io also provides a "WebAssembly hub" that makes it easy for the community to share Envoy Wasm-based filters.

4.3 Machine Learning

Machine learning (ML) is a branch of artificial intelligence (AI) used to help automate, detect, and learn to better predict patterns in data. It naturally requires numerical precision and the ability to take advantage of a computer's graphical processing unit (GPU) for tasks such as facial recognition.

Popular ML platforms are turning to Wasm to deliver ML capabilities that can work across browser and non-browser environments. For example, TensorFlow [28] now provides a Wasm backend in TensorFlow.js for both the browser and Node.js. This approach offers an alternative to the company's WebGL backend that improves performance and supports a broad range of devices.

Scailable [29] provides the ability to convert AI and ML models from several different popular frameworks to Wasm modules that will run anywhere. The company cites computational efficiency, small memory footprint, portability, and safety as the driving factors for the move to Wasm.

4.4 Blockchain Smart Contracts

Smart Contracts are the business logic of the blockchain. They allow developers to create and deploy application-specific logic. The best-known smart contract platform is Ethereum. Ethereum uses a custom Turing-complete VM that developers can use to create many types of on-chain applications.

In the past few years several emerging blockchain platforms have turned to Wasm as their Smart Contract environment versus custom-developed VMs such as Ethereum's. With Wasm, developers can create Smart Contracts in any language and deploy them to an on-chain Wasm environment. Blockchain's platforms such as Parity's Substrate [30] use Wasm exclusively for all Smart Contract development. In Substrate, compiled Wasm contracts are stored on-chain in its binary format and can be hot swapped at runtime with updated contracts.

5 Summary

Wasm is a compact binary format and open specification designed to execute code in a fast, safe, and portable manner. Originally designed to improve the performance of client-side web applications, Wasm has attracted growing interest and broad application across several different domains, from in-browser graphics to server-side blockchain smart contracts.

The open nature of the Wasm specification has led to unprecedented collaboration on the development of the Wasm runtime. Every major web browser now includes a Wasm runtime enabled by default. CPU-intensive applications such as games and CAD tools that were previously impractical for use in a traditional JavaScript browser environment have been successfully ported to the web for use via Wasm.

Outside the browser, Wasm is often employed in the emerging blockchain space to implement sophisticated smart contracts. Cloud providers are exploring Wasm use as a flexible approach for developing microservices as well as running untrusted code. In addition, open-source communities are working to advance the state of the art, as many envision that Wasm will play a growing role beyond the browser in server-side computing. In fact, Solomon Hykes, the founder of Docker, tweeted: *"If WASM+WASI existed in 2008, we wouldn't have needed to create Docker"* [31].

The Wasm security model provides a sandboxed execution environment that can reduce the impact of malicious and/or faulty applications on a host environment. However, Wasm cannot make unsafe code safe. Therefore, developers should follow best practices to minimize unsafe code. They should consider using languages such as Rust with Wasm when security and performance are important.

Overall, due to Wasm's open nature, broad industry support and use, and growing ecosystem of tools and language support, we envision Wasm as playing an increasingly important role in a wide range of applications. Therefore, we highly recommend exploring its potential use as well as continued monitoring of developments in the technology.

6 References

[1] "WebAssembly," 2021. [Online]. Available: https://webassembly.org/. [Accessed 2021].

[2] "The LLVM Compiler Infrastructure," [Online]. Available: https://llvm.org/. [Accessed 2021].

[3] "WebAssembly FAQ," [Online]. Available: https://webassembly.org/docs/faq/. [Accessed 2021].

[4] "Building to WebAssembly," [Online]. Available: https://emscripten.org/docs/compiling/WebAssembly.html. [Accessed 2021].

[5] W. Specification, "WebAssembly," 2021. [Online]. Available: https://webassembly.github.io/spec/core/text/index.html. [Accessed 2021].

[6] Wikipedia, "S-Expression," 2021. [Online]. Available: https://en.wikipedia.org/wiki/S-expression. [Accessed 2021].

[7] WebAssembly, "Wat2Wasm demo," 2021. [Online]. Available: https://webassembly.github.io/wabt/demo/wat2wasm/index.html. [Accessed 2021].

[8] W. C. Specification, 2021. [Online]. Available: https://webassembly.github.io/spec/core/syntax/modules.html#. [Accessed 2021].

[9] W. C. Spec, "Binary Format," 2021. [Online]. Available: https://webassembly.github.io/spec/core/binary/modules.html#sections. [Accessed 2021].

[10] L. Clark, "What makes WebAssembly fast?," Mozilla Hacks, 28 Feb 2017. [Online]. Available: https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/. [Accessed 2021].

[11] A. Hass, A. Rossberg, D. Schuff, B. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai and J. Bastien, "Bringing the Web up to Speed with WebAssembly," 2017. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3062341.3062363. [Accessed 2021].

[12] A. Deveria, "Can I Use Wasm," 2021. [Online]. Available: https://caniuse.com/wasm. [Accessed 2021].

[13] "Wasm-bindgen Guide," 2021. [Online]. Available: https://rustwasm.github.io/docs/wasm-bindgen/. [Accessed 2021].

[14] "Yew Docs," 2021. [Online]. Available: https://yew.rs/docs/en/. [Accessed 2021].

[15] Microsoft, "Blazor," Microsoft, 2021. [Online]. Available: https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor. [Accessed 2021].

[16] "Bytecode Alliance," 2021. [Online]. Available: https://bytecodealliance.org/. [Accessed 2021].

[17] "WebAssembly/WASI," 2021. [Online]. Available: https://github.com/WebAssembly/WASI/blob/main/docs/README.md. [Accessed 2021].

[18] B. Alliance, "WasmTime," Bytecode Alliance, 2021. [Online]. Available: https://wasmtime.dev/.

[19] W. IO, "Wasmer," Wasmer, 2021. [Online]. Available: https://wasmer.io/.

[20] B. Alliance, "WAMR," Bytecode Alliance, 2021. [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime.

[21] D. Lehmann, J. Kinder and M. Pradel, "Everything Old is New Again: Binary Security of WebAssembly," Usenix, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann. [Accessed 2021].

[22] Wikipedia, "StackOverflow," 2021. [Online]. Available: https://en.wikipedia.org/wiki/Stack_buffer_overflow#Stack_canaries.

[23] WebAssembly, "Explainer," 2021. [Online]. Available: https://github.com/WebAssembly/interface-types/blob/master/proposals/interface-types/Explainer.md.

[24] finos/perspective, "Perspective," 2021. [Online]. Available: https://github.com/finos/perspective.

[25] Pyodide, "Pyodide," 2021. [Online]. Available: https://github.com/pyodide/pyodide.

[26] Wikipedia, "Zero-Knowledge Proof," 2021. [Online]. Available: https://en.wikipedia.org/wiki/Zero-knowledge_proof.

[27] QED-IT, "Zkinterface," 2021. [Online]. Available: https://github.com/QED-it/zkinterface-wasm.

[28] TensorFlow, "TensorFlow," 2021. [Online]. Available: https://www.tensorflow.org/.

[29] Scailable, "Scailable," 2021. [Online]. Available: https://scailable.net/.

[30] Parity, "Parity Substrate," Parity, 2021. [Online]. Available: https://www.parity.io/substrate/.

[31] S. Hykes, "Twitter," Twitter, 2019. [Online]. Available: https://twitter.com/solomonstre/status/1111004913222324225?lang=en.

[32] M. Mush, C. Wressnegger, M. Johns and K. Rieck, "New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild," TU Braunschweig, 2019.

[33] Wikipedia, "Stack Overflow," 2021. [Online]. Available: https://en.wikipedia.org/wiki/Stack_buffer_overflow#Stack_canaries.