WebAssembly

Introduction, History, Instruction Set, Security and live Demonstration

A SEMINAR REPORT

Submitted by

R Midhun Suresh

MBT17CS095

In partial fulfilment of the requirements

for the award of the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

Of the A.P. J. Abdul Kalam Technological University

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MAR BASELIOS COLLEGE OF ENGINEERING AND TECHNOLOGY

MAR IVANIOS VIDYANAGAR, NALANCHIRA

THIRUVANANTHAPURAM – 695 015 November 2020

Mar Baselios College of Engineering & Technology

Mar Ivanious Vidyanagar, Nalanchira, Thiruvananthapuram - 695 015

(Affiliated to A. P. J. Abdul Kalam Technological University)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Seminar Report titled '*WebAssembly: Introduction, History, Instruction Set, Security and live Demonstration*' is a bonafide record of seminar presented by R Midhun Suresh (MBT17CS095).

Mrs. Gayathri KS	**Mr. Shibu V S**	**Dr. Tessy Mathew**
Seminar Co-ordinator	Seminar Guide	Head of the Department

Place: Thiruvananthapuram
Date: 8th September 2020

ACKNOWLEDGEMENT

I would like to take this opportunity to extend my sincere gratitude to **Mr. Shibu VS** for guiding me through the process of this seminar. I would also like to thank the seminar coordinator, **Mrs. Gayathri KS**, for providing me with this opportunity which has allowed me to explore a modern technology which would otherwise be beyond the confines of my academic course.

R Midhun Suresh

CONTENTS

List of Figures

Abstract

With rapid advances in web-development, the need to run native code written in a statically typed language within the confines of a web-browser is at a all time high. Since the Web is by design an open environment, it must be ensured that any such technology is safe and that the consequences of malicious code is limited by design.

WebAssembly is a binary format for the web that is designed with safety, performance and portability in mind. Furthermore, it is implemented in all popular browsers and is readily available as WebAssembly 1.0 to be used in a production setting.

Here we present WebAssembly as a suitable candidate for the aforementioned task and argue that it is the gradual and expected culmination of several previous unsuccessful attempts at running native code on the Web. We consider the language features that make JavaScript slow and explore how the LLVM compiler toolkit can be used to minimize the effort needed to create a general compiler that outputs WASM binary from any other language. We describe the architecture and working of WebAssembly and chronicle the developer experience by compiling modern C++ code to WASM.

1. INTRODUCTION

The internet was originally visualized as a network capable of exchanging static documents among a few isolated nodes. In the several decades that followed its invention in 1960, the internet has metamorphized into a vast and ubiquitous inter-connection of computers that serve billions of dynamic requests every second. Today, the internet is heavily utilized in everything from high-level trading of stocks to the editing of documents in the cloud.

Modern consumers expect web applications to run as fast as the native applications on their smartphones and computers. WebAssembly meets this demand by providing a way to run native code on the web. Native code, in this context, means code that is written in languages like C/C++/Rust that are often statically typed. These languages also have no garbage collection and instead pass the burden of managing memory to the developer.

WebAssembly, abbreviated as WASM, is a binary format that runs on a stack machine. This means that it is more compact than JavaScript because instead of serving text code to the client, WASM serves binary code and therefore require less number of bytes to encode the same sequence of operations.
Since WASM is executed on a stack machine, the Instruction Set Architecture is also minimal, making use of zero-addressing. Operands are popped from the stack and the resultant of the computation is pushed back. This makes the adoption of WASM more likely since stack machines are much easier to implement than register machines.

It must be stressed that WASM is not a language that developers are expected to learn and program in. Instead it is designed to act as a compilation target to which other languages like C++ can be compiled. Further it is not intended to compete with or replace JavaScript. It is rather designed to tightly interact with JavaScript and aid in the development highly-performant web applications.

WebAssembly has already been used to port sophisticated applications such as Unity and Unreal Engine to the web. Initial benchmarks show that programs ported to WASM can run with up to 90% of the speeds of their native counterparts (running on android Firefox).

2. Previous Technologies

Running native code on the web is nothing new. It has been attempted multiple times over the evolution of the web but have always failed to catch on. Almost all of the prominent methods that were previously used to run native code on the web are now depreciated. We will consider some of them here and will explore how WebAssembly aims to combat the obstacles previously faced by these technologies.

a) Microsoft ActiveX

ActiveX was the Microsoft brainchild that brought Component Object Model to the web. COM is an ABI (Application Binary Interface) that Microsoft created to enable developers to make platform-independent programs that could be expected to run anywhere. The fundamental idea behind COM was to necessitate a fixed interface for all programs through which other programs could communicate to it. ActiveX allowed websites to install COM components to a user's system and control its usage from within the website. It was built as a competitive product that would provide an alternative to Java Applets.

However, ActiveX was plagued with several complications and is infamous for its extremely poor security. Although it was originally intended to be cross platform, it did not work on the Linux operating system. Further it had severe incompatibilities with other browsers such as Firefox. To make things worse, ActiveX used a trust model for its security in which developers signed a contract with Microsoft promising not to develop malicious applications. This meant that any security vulnerability that creeped into the applications due to bugs in the code could have dire consequences for its users.

Unlike other methods of running native code on the web, ActiveX did not sandbox the native code. This meant that any ActiveX component had full access to the Operating System. Worse still, initial releases allowed developers to install components silently without explicit initiation from the user.

Microsoft officially depreciated ActiveX in 2015. Microsoft does not support it on the latest Edge browser.

b) Java Applets

Java Applets was the most preferred method of running native code on the web. It has been the de facto standard for safe and robust execution of Java programs on the web. Java Applets were embedded into webpages with the applet (now object) tag. The applet ran Java bytecode by utilizing the Java Runtime Environment. It was used to power everything from games (like the MMORPG Runescape) to complex and impressive applications like the NASA WorldWind (a virtual globe).

Java Applets were more secure than ActiveX components since the bytecode was executed within a sandbox. Additionally, it required the user to explicitly grant permission before running. That being said, applets too had issues which led to it eventually being depreciated in Java 9.

Firstly, Java Applets, like other Java applications, were dependent directly on the presence of the Java Runtime Environment which by itself was a considerably large application that had to be separately installed by the user. Additionally, most browsers required specialized plugins in order to run Java Applets.

However, the actual reason behind the downfall of Java Applets can be attributed to the increase in execution speed of JavaScript – primarily due to JIT compilation in recent JavaScript engines – and modern Web API's that allow hardware accelerated graphics. In other words, JavaScript can do today most of what Java Applets were traditionally used for without the need to install any other additional software (like JRE).

Additionally, an estimated 51 percent of total internet users are on smartphones running android or IOS, neither of which supports a JVM.

Java applets also left a lot to be desired for in terms of security since the average user lacked the ability to discern legitimate applets from malicious applets. This led to many users being infected by attack vectors such as drive-by downloads.

<u>c) Google Native Client</u>

Native client, more commonly known as NaCl, was an attempt at running native code developed by engineers at Google. The idea was to run C/C++ code on the chrome browser by compiling it to 'nexe'. NaCl code runs within a strict sandbox in order to limit the damages that malicious code can cause. It relied on static validation of code to ensure safety. This involves scanning the code to detect patterns that are generally not found in legitimate programs

Code compiled by using NaCl will work on any Operating System however it is not portable because nexe code is generated for a particular architecture. Google released a second, improved version of NaCl called PNaCl (Portable Native Client) which instead of generating nexe code would generate pexe code. The browser would recompile the pexe code to machine code depending on the system architecture. Thus, PNaCl was portable.

However, NaCl and PNaCl only worked on Chrome and no support was provided by Firefox. This is partly because native client was perceived as slightly better clone of Microsoft ActiveX.

Native client was also not interoperable with JavaScript code. This is a severe limitation because most use cases required JavaScript for basic interaction and native code for heavy computation.

Despite its limitations and problems, native client was used to port graphic heavy games like Bastion. It has since been depreciated and is now only used in Chromebooks.

<u>d) Asm.js</u>

An academic discussion of WASM cannot proceed without the mention of asm.js. This is because asm.js is the direct predecessor of WebAssembly. It was a compilation target developed by engineers at Mozilla (Firefox) for faster computation.

To understand the rationale behind asm.js it is necessary to briefly explain what makes JavaScript slow. JavaScript is a dynamically typed language. This means that the compiler needs to perform multiple steps to figure out what type a variable belongs to.

Asm.js is a strict subset of JS that includes additional information about the typing of variables. This means a compiler can optimize the JavaScript code for performance by eliminating the extra computation needed to figure out the typing details.

Since Asm.js is a subset of JavaScript, it is guaranteed to be backward compatible with browsers that do not support an optimizing compiler for it. That is browsers that do not support asm.js will still run the code albeit without the faster performance.

Asm.js was depreciated in favour of WebAssembly.

e) WASM in retrospect

Here we state how WebAssembly deals with the problems and limitations that haunted the previous technologies it aims to replace.

- WebAssembly is designed to work across all architecture, browsers and operating systems. It is truly portable.
- WebAssembly enables security through technological implementation as opposed to using a trust model.
- It requires no additional software or tooling and comes packaged with browsers.
- It can be polyfilled using asm.js in browsers that do not support it.
- WebAssembly works consistently with JavaScript APIs.

3. Working

We will now take an in-depth look into the functioning of WebAssembly.

a) <u>Components of a WASM program</u>

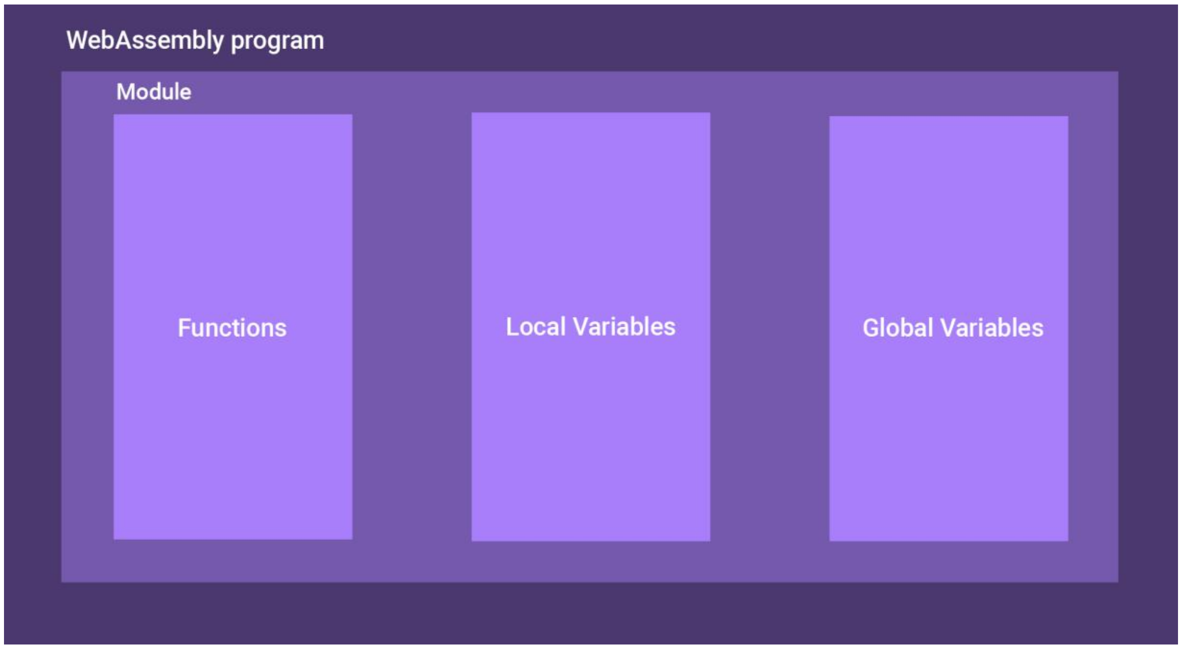


Figure 3.1. Hierarchy of Components

A WebAssembly program is composed of different units. A brief description of them ensue:

Modules

Every WASM program is made up of a module. The module contains the definitions for functions and variables (local and global). The functions are exported and imported within modules. Modules are run as instances in which the execution environment (e.g. JavaScript environment) assigns a mutable memory and an execution stack to it.

Functions

These components take in WASM values as input (parameters) and output a single WASM value. Function in WASM are different in nature from functions in JavaScript. Functions in WebAssembly are not first-class citizens which is to say that they cannot be assigned to variables or passed as arguments. This also implies that WebAssembly does not support nested functions. However, recursion and calling other functions are allowed.

Variables

WebAssembly contains only four fundamental data types. These are the 32-bit and 64-bit flavours of integers and IEE 754 floating point numbers. Most instructions in WASM have different versions for each of these data-types.

Local variables are accessed using **get_local** and **set_local** instructions. We will consider their syntax and operation in further sections. WebAssembly also supports global variables that can be either set to be mutable or immutable.

Like local variables, global variables are also controlled using **get_global** and **set_global** instructions. Global variables must be initialized using constant expressions which are expressions that satisfy the constraint that they are evaluable at compile time.

b) <u>Compilation to WASM</u>

Now we concern ourselves with the challenge of transforming code written in C/C++ to WASM binary. The process is generalized in figure 3.2.

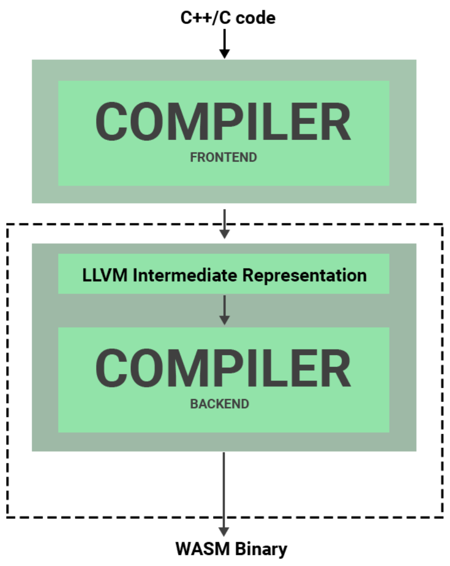


Figure 3.2. Compilation process

The Compiler is visualized as being composed of two parts: the frontend and the backend. The frontend is concerned with the transformation of C/C++ code into LLVM bytecode (LLVM IR). The backend is concerned with further transforming the output bytecode into WASM bytecode.

The rationale for this separation can only be understood by briefly describing what LLVM is. LLVM is a compiler toolchain that is intended to be reusable. Additionally, LLVM gives extreme importance to optimizable code. When compiler technologies can be reused, the time required to create new compilers can be reduced dramatically.

From figure 3.2, it is clear that we have a compiler that takes as input LLVM IR and outputs a WASM binary. Now if we wanted to create a compiler that outputs WASM from Rust code, we do not have to reinvent the wheel and come up with a new compiler. We only need to create a new frontend compiler that takes Rust code as input and outputs LLVM IR. In other words, we can reuse the backend compiler with many different frontend compilers.

c) <u>WASM pipeline on the browser</u>

Now that we have a WASM binary on our hand, we can examine what the browser does to execute it. It is important at this point to contrast the execution pipeline of WebAssembly with that of JavaScript.

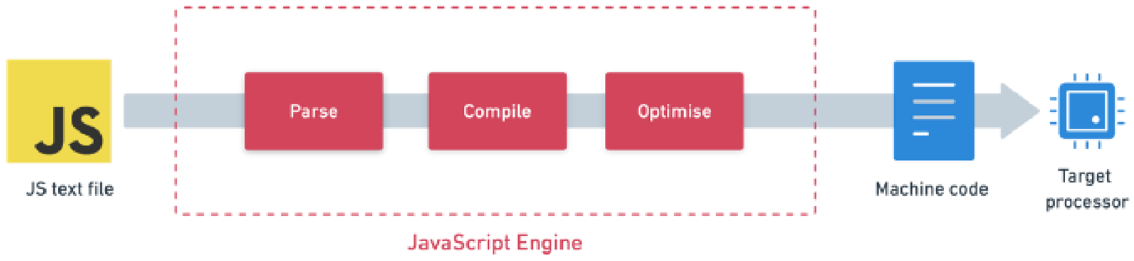


Figure 3.3.a. JavaScript execution pipeline

Figure 3.3.a illustrates the actions that a modern browser takes when it needs to execute JS code. The first point of consideration is that the input is a text-file with a *.js* extension. There are two compilation phases. The box labelled ‘Compile’ on the figure is an un-optimizing baseline compiler that immediately starts JIT compiling JavaScript source to machine code. The box labelled ‘Optimise’ is a second and better compiler that produces more efficient and faster code by consuming more memory and utilizing extra processor cycles.

This complex arrangement is necessitated by the fact that JavaScript with an interpreter is very slow. The purpose of the first baseline compiler is to immediately produce some runnable machine code while the second optimizing compiler is busy working on producing faster code. Without the first compiler, the user would experience a delay having to wait for the optimization to complete.

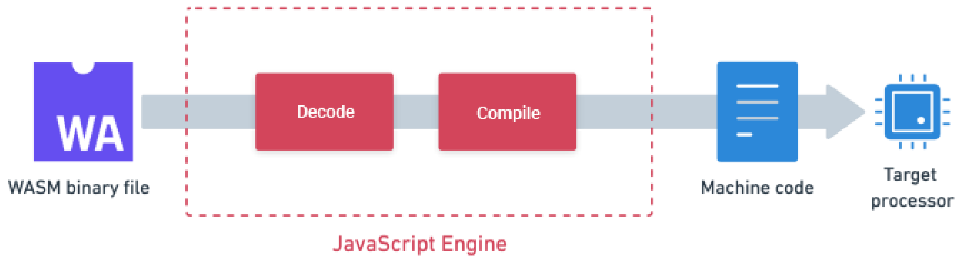


Figure 3.3.b. WebAssembly execution pipeline

Figure 3.3.b shows the WASM pipeline. It is immediately clear that the process of running WASM code is much more simplistic compared to that of running JavaScript code.

In contrast to the JavaScript pipeline, WebAssembly is in binary format and therefore no parsing is necessary. Secondly, no extra time or hardware has to be utilized in optimizing WASM code since it is already optimized during compilation. The only operation that needs to be done is to decode and compile the WASM code to machine code.

Therefore, WebAssembly gains improvements in performance by eliminating the paint points of JavaScript compilation. The minimal decode and compile process also ensures that all browser vendors will support WebAssembly since it is relatively easy to implement without breaking existing processes.

d) WASM Memory Model

Perhaps the greatest security feature of WebAssembly is its unique memory model.
The memory allocated for a WASM program is in the form of a liner array of bytes.

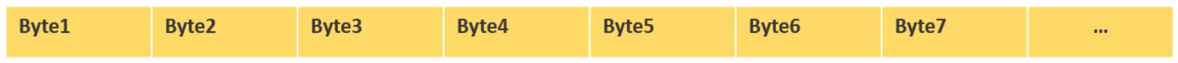


Figure 3.4. Memory model of WASM

For WebAssembly code executed within a JavaScript runtime, the memory is simply an **ArrayBuffer** object. This linear array of bytes exists independently from the code space and WebAssembly internals.

There are two advantages to this memory model:

- The worst a malicious WebAssembly code can do is corrupt its own memory without affecting any other parts.
- Since the memory is a JavaScript object and since JavaScript objects are garbage collected, there is no possibility of WASM programs creating memory leaks

Memory in WebAssembly is dynamic i.e. it can be resized by calling the **grow_memory** instruction. Memory is increased in units of 64KB – which is the average of popularly used page sizes in different operating systems. The amount of memory currently allocated can be found by calling **current_memory** instruction.

Data is inserted and deleted into the memory with load and store instructions respectively.

The WebAssembly specification mandates that little endian byte order be followed. This is because most memory controllers have the option to convert between both endian formats. Additionally, most modern hardware support bi-endian encoding.

4. Instruction Set

Instructions in WebAssembly do not explicitly mention their operands. Instead the operands are popped from the stack and the resultant of the execution is pushed back in. We will consider some of the important instructions here.

a) <u>Numeric Instructions</u>

The following arithmetic instructions are supported:

- add
- sub
- mul
- div_*sx*
- rem_*sx*

The first three are the common addition, subtraction and multiplication operations. The last two are division and modulo operation respectively. These last two operations come in two flavours – signed and unsigned – which is selected with the *sx* part of the instruction. The *sx* part can be either *u* (unsigned) or *s* (signed).

The following commonly used numerical functions are available:

- abs
- neg
- sqrt
- ceil
- floor

These are the absolute, negation, square root, ceiling and floor operators respectively.

The following comparison operators are available:

- eq
- ne
- lt
- gt
- le
- ge

These stand for *equal to*, *less than*, *greater than*, *less than or equal to* and *greater than or equal to* comparison operators. They consume two operands from the stack and pushes a Boolean output into the stack.

Two instructions are available to push constant values into the stack:

- **inn**.const
- **fnn**.const

Here the prefixes tell the compiler whether the constant is an integer or a floating-point number and their size.

For e.g., ***i32.const 51*** would push the 32-bit integer constant 51 into the operand stack.

b) Variable Instructions

These instructions are used to manage local and global variables. They are:

- **local**.get, **global**.get
- **local**.set, **global**.set
- **local**.tee

The get instruction is used to fetch the values of local or global variable (as specified in the prefix). Similarly, set instruction is used to assign values.
The tee instruction does exactly what the set instruction does but it does not pop a value off the stack.

c) Memory Instructions

These instructions are used to manipulate the linear array of memory. The following instructions are available:

- **inn**.load, **fnn**.load
- **inn**.store, **fnn**.store
- **memory**.size
- **memory**.grow

The load instruction pushes data from the memory into the stack whereas the store instruction pops a value off the stack and stores in the memory. The instructions are available for all datatypes i.e. ***i32***, ***i64***, ***f32*** and ***f64***.

The size instruction is used to find out the current memory size. The grow instruction is used to acquire more memory. If the amount of memory requested is greater than the amount of memory available in the system, the grow instruction returns -1.

d) <u>Control Instructions</u>

Unlike other assembly languages, WebAssembly supports higher-level control structures such as *if* and *else*.
The following control instructions are available:

- nop, unreachable
- block, loop, if, else, end
- br, br_if, br_table
- call, call_indirect

The **nop** instruction signifies to do nothing. The **unreachable** instruction, on the other hand, defaults to a trap. For a JavaScript execution environment, this trap is an exception.

The **block**, **loop**, **if**, **else** and **end** instructions give the developer the ability to utilize structured flow. The structured flow in WASM is much richer than those found in other assembly-based languages which predominantly use branches to named labels.

The following WASM text format (readable format of WASM binary) illustrates the usage of if else block.

```
(if (result i32)
  (i32.lt_s
    (get_local $x)
    (get_local $y)
  )
  (then
    (i32.const 10)
  )
  (else
    (i32.const 20)
  )
)
```

Figure 4.4.a. Example if – else construct in WASM

Figure 4.4.a shows a WASM code snippet that examines if ***$x <$y*** where **$x** and **$y** are local variables. If the condition holds, the 32-bit integer 10 is pushed onto the stack. Otherwise the constant 20 is pushed onto the stack.

The **loop** instruction behaves like an infinite *while(1*) loop in C except it has a default break at its end. The loop can be continued by branching to it before the end.

```
    loop
        call myFunc
        if
            br 1
        end
    end
```

Figure 4.4.b. Example looping

Figure 4.4.b shows an example loop in which *myFunc* is called repeatedly.

Branching in WebAssembly is based on labels but the labels do not signify a named position in the code instead they specify a block based on relative nesting.

For example, in the code shown in figure 4.4.b, the ***br 1*** instruction will branch to the label 1. Label 0 would be the ***if … end*** block. Naturally, label 1 is the ***loop … end*** block.

The ***block … end*** instructions can be used to created more nested blocks that act as labels.

The ***br_if*** instruction branches based on a conditional and the ***br_table*** instruction branches indirectly through table index.

Functions are called using the **call** instruction. Since WebAssembly acts as a compilation target for languages like C/C++. Since these languages support features such as function pointers, WebAssembly provides the **call_indirect** instruction which can be used to call functions by indexing into a table.

The tables mentioned during the description of the **br_table** and the **call_indirect** instructions have the following format:

Table index	Function reference

Figure 4.4.c. Table format in WebAssembly

Storing the references in a separate table instead of storing them on the WASM memory adds security by ensuring that these references cannot be manipulated by malicious code.

5. Security

Unlike previous technologies that let developers run native code on the web, WebAssembly aims to provide security through technological implementation rather than through something akin to a trust-based model

All browsers implement a sandbox environment that acts as a black-box within which code can run securely. This ensures that the code can access resources only though well-defined API endpoints. WebAssembly code runs within such a sandbox.

Additionally since WASM code runs within the confines of a web-browser, it is restricted by all the security polices of the browser. This means that WebAssembly code cannot override, say for example, the Same-origin Policy. The memory model engineered for WASM ensures that a malicious binary cannot damage the data-structures used by the WebAssembly internals or the internal stack used by the stack machine. Since the memory is after all a JavaScript object, the possibility of memory leaks is also eliminated.

Moreover, WebAssembly has strict Control Flow Integrity which acts as an obstacle to many popular code injection attacks. For example, unlike other assembly languages, WebAssembly does not have ***goto*** instructions. Instead branching is limited to scoped labels.

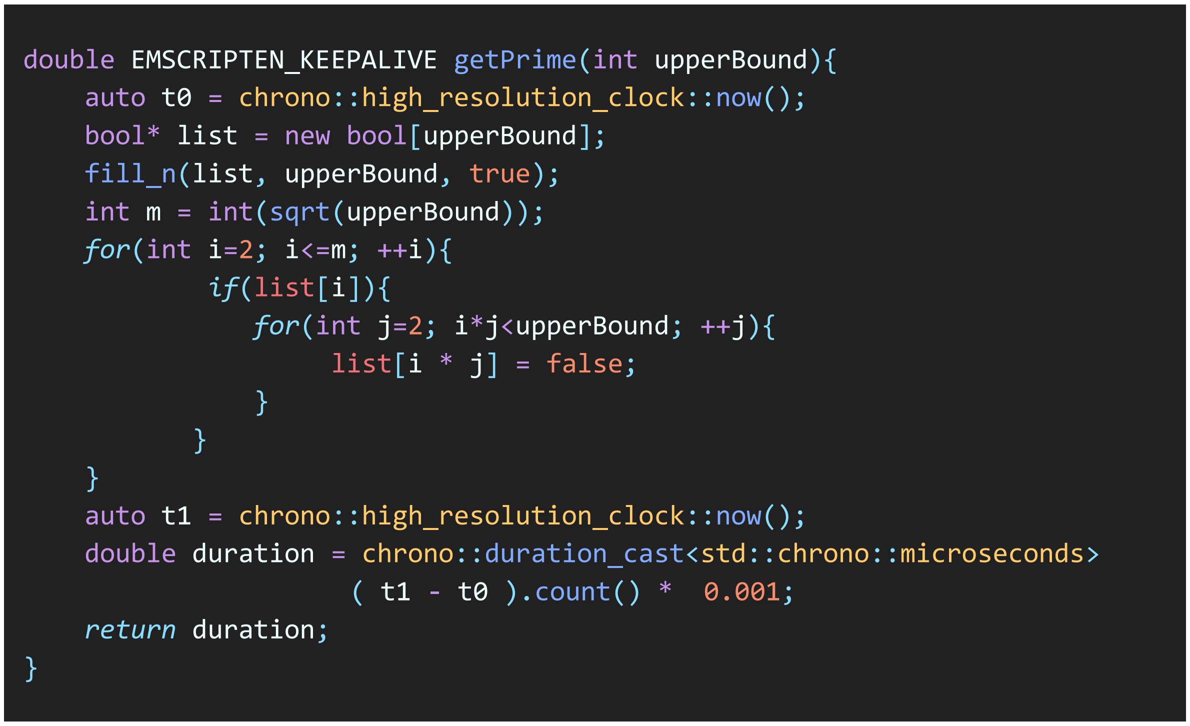
Many of the unforgiving security flaws in C/C++ code usually originates from the heavy usage of pointers. WebAssembly avoids this by not including pointer types. Instead indirect references are allowed by indexing to a table with key values that are integers.

Every direct and indirect function call is validated during compilation and runtime to ensure that their signatures match. If the signatures do not match, a compilation error or a trap state is produced. Finally, the call-stack of a WebAssembly program is hidden from the module. Since memory is a linear array, buffer overflows cannot occur due to the already enforced bound checking.

6. Compiling modern C++ to WASM

Here we detail the process of actually compiling some C++ code to WASM. We will use the **emscripten** compiler for this purpose.

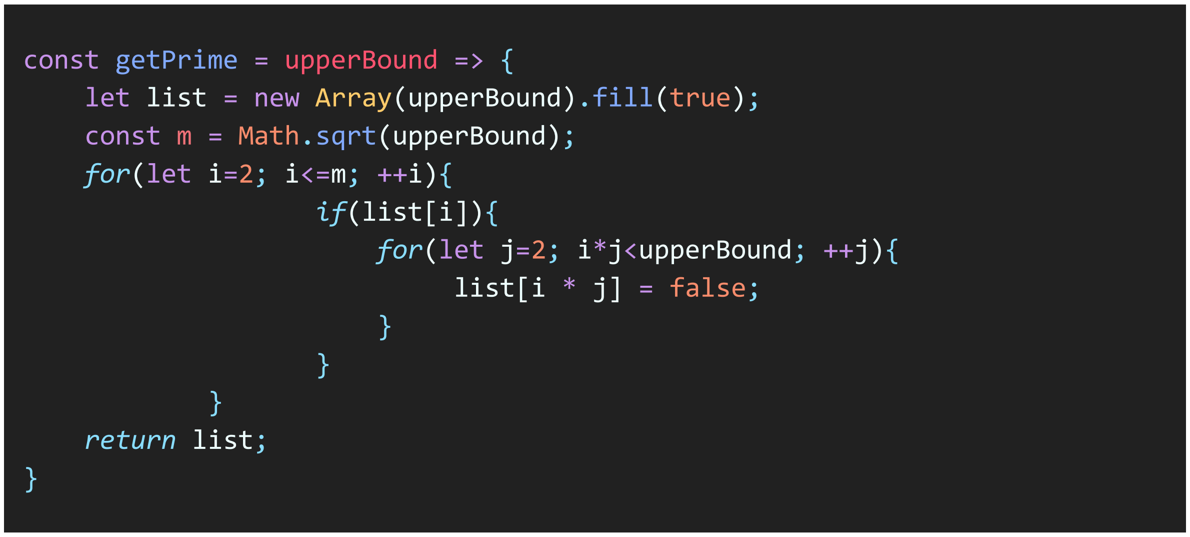
To illustrate the performance difference between pure JavaScript and C++ using WASM, we have chosen the Sieve of Eratosthenes algorithm. This algorithm is used to efficiently compute a list of prime numbers.



```
double EMSCRIPTEN_KEEPALIVE getPrime(int upperBound){
    auto t0 = chrono::high_resolution_clock::now();
    bool* list = new bool[upperBound];
    fill_n(list, upperBound, true);
    int m = int(sqrt(upperBound));
    for(int i=2; i<=m; ++i){
            if(list[i]){
               for(int j=2; i*j<upperBound; ++j){
                    list[i * j] = false;
               }
           }
    }
    auto t1 = chrono::high_resolution_clock::now();
    double duration = chrono::duration_cast<std::chrono::microseconds>
                      ( t1 - t0 ).count() *  0.001;
    return duration;
}
```

Figure 6.a. Sieve of Eratosthenes in C++

Figure 6.a shows our C++ code. We are using the standard C++ 11 std::chrono library to measure time taken to run our program. The function **getPrime** takes an integer input and returns a double that represents the time taken for our function to execute. The function is defined using EMSCRIPTEN_KEEPALIVE to ensure that the compiler does not optimize it away.



```
const getPrime = upperBound => {
    let list = new Array(upperBound).fill(true);
    const m = Math.sqrt(upperBound);
    for(let i=2; i<=m; ++i){
                   if(list[i]){
                       for(let j=2; i*j<upperBound; ++j){
                            list[i * j] = false;
                       }
                   }
            }
    return list;
}
```

Figure 6.b. Sieve of Eratosthenes in JavaScript ES6 syntax

Figure 6.b shows the same algorithm implemented in JavaScript.

We can compiler our C++ file (named native.cpp) by issuing the following command:

emcc -o output.html native.cpp --shell-file html_template/shell_minimal.html -s "EXTRA_EXPORTED_RUNTIME_METHODS=['ccall']" -O3

The command can be broken down as follows:

- *emcc*: This is the emscripten compiler.
- *-o output.html*: Tells emcc that we want to generate the html page to test our code.
- *native.cpp*: This is our C++ file.
- *--shell-file html_template/shell_minimal.html*: Tells emcc to use shell_minimal as html template.
- *-s "EXTRA_EXPORTED_RUNTIME_METHODS=['ccall']"*: Tells the compiler to make ccall available for use. We will use ccall to call our C++ function.
- *-O3*: Tells the compiler to enable optimizations. This will make out code faster.

Now the actual process of calling our C++ function from JavaScript is trivial. It can be done as follows:

```
var result_wasm = Module.ccall(
                'getPrime', // name of C function
                'number',   // return type
                ['number'], // argument types
                [input] // arguments
            );
```

Figure 6.c. Calling WASM function in JavaScript

7. Conclusion

Having taken a broad overview of WebAssembly we can now conclude that it exhibits all the necessary traits that we would expect in an optimal solution to the problem of running native code on the web. It is developed and maintained by engineers at Chrome (Google), Mozilla (Firefox), Webkit (apple) and Blink (Microsoft). This ensures that every modern browser will support WebAssembly code. However, WebAssembly is still in its infancy.

The WebAssembly team has promised a variety of laudable features in future releases. Some of them include:

- Tail Call Optimization
- Threading
- Exception Handling
- SIMD instructions
- Garbage Collection

Some of these features are already available in certain browsers.

Besides running on the browser, WebAssembly can be expected to take off as a way to run portable code across a variety of architectures. This is already available in the form of WASI (WebAssembly System Interface).

Although this report focused on primarily running C/C++ code on the browser, WebAssembly supports about 40 languages with more languages being supported each year. This list includes bleeding edge languages like Rust, Go and Zig.

To summarize, WebAssembly is a modern and safe technique that can be used to run code written in most statically typed languages on the web; without sacrificing an ounce of security.

References:

[1] Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A. and Bastien, J.F., 2017, June. Bringing the web up to speed with WebAssembly. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (pp. 185-200).

[2] Watt, C., Rossberg, A. and Pichon-Pharabod, J., 2019. Weakening WebAssembly. Proceedings of the ACM on Programming Languages, 3(OOPSLA), pp.1-28.

[3] Reiser, M. and Bläser, L., 2017, October. Accelerate JavaScript applications by cross-compiling to WebAssembly. In Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (pp. 10-17).

[4] https://webassembly.github.io/spec/core/ - WASM Specification

[5] https://developer.mozilla.org/en-US/docs/WebAssembly - MDN documentation on WASM