Bachelor’s thesis

Information and Communications Technology

2020

Nguyen Hoang Minh Le
Ba Hiep Phung

WEBASSEMBLY: PORTABILITY AND RISKS

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR’S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communications Technology

2020 | 47

Nguyen Hoang Minh Le
Ba Hiep Phung

WEBASSEMBLY: PORTABILITY AND RISKS

WebAssembly (or Wasm in short) is a new programming language which is supported by several browsers nowadays and can be run with near-native performance. Web developers can utilize JavaScript libraries that are using WebAssembly under the hood without writing low level language such as C/C++, Rust, etc.

The purpose of the thesis was to analyze the portability and the performance of WebAssembly by researching WebAssembly Ecosystem with Rust, compiling a task from Rust and TypeScript into WebAssembly and comparing the runtime of this task to the same task which was written in JavaScript. The authors also aimed to research how WebAssembly can be abused in the real world to perform illegal activities by analyzing a cryptominer sample and a keylogger sample which were written in WebAssembly.

The results show that WebAssembly performs the tasks faster than other languages (Rust, JavaScript and TypeScript). Moreover, the cryptominer and keylogger in WebAssembly are easy to create and use in any website without any detection.

This thesis provides a basic introduction to WebAssembly, compares the performance of WebAssembly with JavaScript, explores the usage of Emscripten and discovers how WebAssembly can be used in cryptomining and keylogging. The thesis aims to help web developers and security researchers in developing the web environment more efficiently and safely.

KEYWORDS:

webassembly, malware, cryptomining, rust, c/c++, javascript, keylogger, performance, portability, risks

CONTENTS

APPENDICES

FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
AST	Abstract Syntax Tree
BSON	Binary JavaScript Object Notation
CL	Command line
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DOM	Document Object Model
GC	Garbage Collector
HTML	HyperText Markup Language
JIT	Just-In-Time
JSON	JavaScript Object Notation
LLVM	Low Level Virtual Machine
LTS	Long-Term Support
MDN	Mozilla Developer Network
MVP	Minimum Viable Product
SIMD	Single instruction, multiple data
SQL	Structured Query Language
UI	User Interface
W3C	World Wide Web Consortium
WASM	WebAssembly Binary format
WAT	WebAssembly Text format

XSS	Cross-site Scripting

1 INTRODUCTION

The objectives of the thesis are to analyze the portability and the performance of WebAssembly. The thesis also aims to research how WebAssembly can be abused in the real world to perform malicious deeds. To analyze the portability and the performance of WebAssembly, we first research the WebAssembly Ecosystem with Rust, then we use Rust and AssemblyScript to perform some tasks and compile them to WebAssembly binary, then we compare the runtime of the same task which was written in JavaScript to have an overview of WebAssembly performance. The abuse of WebAssembly in the real world will be researched by collecting, analyzing real cryptominer sample and making a simple keylogger which can be used directly in a website.

The thesis begins with a brief introduction of WebAssembly (Chapter 2). The first two sections of Chapter 2 explain what WebAssembly is and the structure of WebAssembly and they are written by Minh Le. The last two sections of Chapter 2 provide the goals of WebAssemby and how to write the very basic ”Hello World!” program in WebAssembly which are written by Hiep Phung. Chapter 3 discusses the portability and performance of WebAssembly compared to other programming languages (Rust, JavaScript, TypeScript) and it is introduced by Hiep Phung. IChapter 4 discusses the risks of using WebAssembly in the wild, how the Emscripten toolchain works and presents a demo of a WebAssembly keylogger.

2 INTRODUCTION TO WEBASSEMBLY

2.1 What is WebAssembly?

When we talk about web development, HTML, CSS and JavaScript are the top languages that a usual developer will think of. JavaScript is great for creating rich and interactive user interface. However, it is not capable of handling highly computed tasks for example image editing, video editing, low-latency augmented reality. To tackle this problem, WebAssembly was first introduced on March 2017. With the high demand for more efficient web applicattion and the rapid development of web technology, WebAssembly is now supported in 4 major browsers (Firefox, Chrome, Safari and Edge) and has received the recommendation from W3C on December 2019 [1].

WebAssembly brings more performancing to web application due to its capability to utilize computer hardware such as the processor and memory which JavaScript cannot do. Not only is WebAssembly being used by developer to improve web application but is also being absused by hackers to perform malicious deeds. There are several cases that have been found where Webassembly was used to perform crytomining, hide malicious code and log keystroke without user consent.

WebAssembly was on the initial public announcement in 2015 on Github and recently has been become a recommendation by World Wide Web Consortium on September 2019. It is designed to compile code from another low-level languages such as C/C++ or Rust then run in browser environment but it is not considered as a replacement for Javascript. [2]

There are two types of format in WebAssembly's specification: WebAssembly binary (WASM extension) and WebAssembly Text Format (WAT extension). The WebAssembly binary format is fetched by web application and converted into a Module. On the other hand, the WebAssembly Text Format is not designed to run in browser but rather for humans to read. In this thesis, we will use the WebAssembly Text Format for the explanations and demonstrations.

WebAssembly has a significant impact on website development because it allows a program which is written in another language (for example C/C++ or Rust) to run on a website which was impossible previously. Along with EmScripten, which will be introduced in Chapter 4, researchers and developers have carried out several interesting experiments on different aspects such as computer vision, language detector, and cryptography.

2.2 Structure of WebAssembly

There are two types of format in WebAssembly's specification: WebAssembly binary (WASM extension) and WebAssembly Text Format (WAT extension). WebAssembly binary format is a series of operation codes, which is then fetched by the web page and converted into a Module. On the other hand, WebAssembly is a human readable format of WebAssembly binary, it is not designed to be run on the browser.

A module is the basic unit of code in both the binary and textual formats. ”In the text format, a module is represented as one large S-expression. S-expressions are a very old and very simple textual format for representing trees, and thus we can think of a module as a tree of nodes that describe the module's structure and its code” (MDN Web docs) [3]. It is clear, therefore, that we can understand and describe the text format of WebAssembly easily in a tree.

2.3 Goals of WebAssembly

Web application has been evolved and developed rapidly in the past decade. There are a massive number of JavaScript libraries and frameworks that helps to create rich and performance web application; however, due to the current user demand for universal access, web applications have to be more complex and heavier, for example, in video games, graphic design, and audio processing. To make this possible JavaScript libraries and frameworks use WebAssembly to reduce the loading and compilation process. There are some other approaches such as using plugin to achieve near-native performance in the browser, but they tend to be unstable and insecure. On the other hand WebAssembly runs entirely within the Web Platform where developers utilized JavaScript libraries that use WebAssembly handle CPU-intensive calculations [4].

2.4 “Hello, world!” in WebAssembly

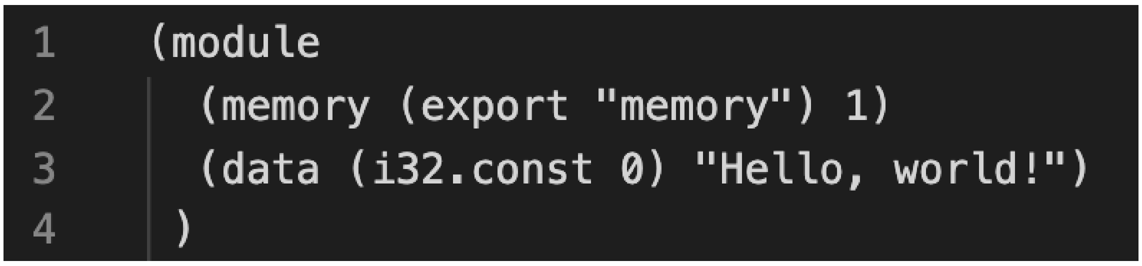
To start with programming in WebAssembly, firstly, we install The WebAssembly Binary Toolkit (wabt) [5]. It is a tool to compile WebAssembly text code into Webassembly binary to run the browser and chose the compile to compile WebAssembly text in this case is Clang [6].

```
$ git clone --recursive https://github.com/WebAssembly/wabt
```

```
$ cd wabt
```

```
$ make clang-release
```

After that, we create a file name hello-world.wat



```
(module
  (memory (export "memory") 1)
  (data (i32.const 0) "Hello, world!")
)
```

Figure 1. WebAssembly text format.

Since WebAssembly is using linear memory [7], it must be specified how many memory blocks it needs to be allocated. The second line show that 1 memory block of 64 kilo bytes is allocated . The memory block is exported from the module under the name “memory”.

We want to assign the string “hello, world” to the memory block, therefore, a starting index is specified. In this case, it is the index zero and this memory block is allocated by calling (data (i32.const 0) "Hello, world!").

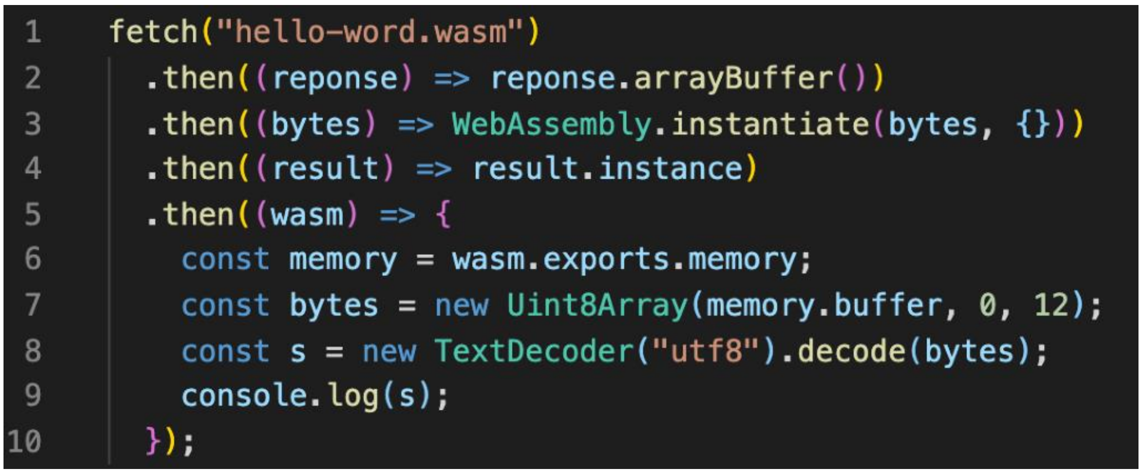
We next compile WebAssembly text to WebAssembly binary using wat2wasm CL tool which is install earlier.

```
$ ./bin/wat2wasm hello-word.wat -o hello-world.wasm
```

Wat2wasm accepts the hello-word.wat as an input file then the flag “o” is provided to indicate that the output is in binary format name hello-world.wasm.

The output hello-world.wasm is the WebAssembly binary. It can be imported to and run in any browsers that support WebAssembly.

We will follow MDN Web docs to setup and use WebAssembly in JavaScript project [8].



```
fetch("hello-word.wasm")
  .then((reponse) => reponse.arrayBuffer())
  .then((bytes) => WebAssembly.instantiate(bytes, {}))
  .then((result) => result.instance)
  .then((wasm) => {
    const memory = wasm.exports.memory;
    const bytes = new Uint8Array(memory.buffer, 0, 12);
    const s = new TextDecoder("utf8").decode(bytes);
    console.log(s);
  });
```

Figure 2. Import WebAssembly to webpage.

The instance return from wasm.exports.memory is the WebAssembly linear memory object. The bytes array that stored the string “hello world” in that memory block can be accessed by selecting index 0 to 12 in that memory buffer. The final step is to decode those bytes array to text format by using the TextDecoder object of the browser [9].

3 WEBASSEMBLY PORTABILITY AND PERFORMANCE

3.1 WebAssembly Ecosystem with Rust

WebAssembly is not very practical to be hand-written but most likely to be compiled from other languages such as Rust, C, C++, Kotlin, etc. Also WebAssembly is not designed to replace JavaScript. They are expected to work together with some configuration such as a web application using HTML/CSS/JavaScript to create a UI with WebAssembly controlling the center canvas, a web application which imports WebAssembly modules to handle the heavy looped logic, etc.

Rust and toolchains

Rust is an incredible high-performance programming language. It utilizes a new concept called ownership to handle memory which get rid of runtime and garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages. It is also the one of the most supported language when it comes to compiling into WebAssembly.

Rustup

Rustup [10] is a Rust installation tool which is recommended by the Rust team for Rust to compile and run in different operating systems such as Window, Linux and MacOS

Cargo

Cargo [11] is the Rust package manager. Cargo downloads Rust package’s dependencies, compiles packages, makes distributable packages, and uploads them to crates.io, the Rust community’s package registry.

Wasm-pack

Wasm-pack [12] is a CL library to generate WebAssembly from Rust that integrates with JavaScript, in the browser environment or with Node.js in native environment. It can be used alongside any JavaScript packages in workflows that already used, such as webpack.

Wasm-bindgen

JavaScript and Rust have complex types. However, WebAssembly specification only defines four types: two integer types and two floating-point types. Due to those limitation, come wasm-bindgen [13]. It allows JavaScript and Rust to interactive with each other API and provides a bridge between JavaScript, Rust and WebAssembly without any verbose boilerplate.

The js-system crate

The js-sys crate [14] provides Rust the raw bindings to all the global APIs guaranteed to exist in every JavaScript environment by the ECMAScript standard, and its source lives at wasm-bindgen/crates/js-sys. With the js-sys crate provides Rust can get access to Objects, Arrays, Functions, Maps, Sets, etc.

The web-system crate

The web-sys crate [15] provides Rust the raw bindings to all the Web's APIs such as window object, WebGL, WebAudio, etc.

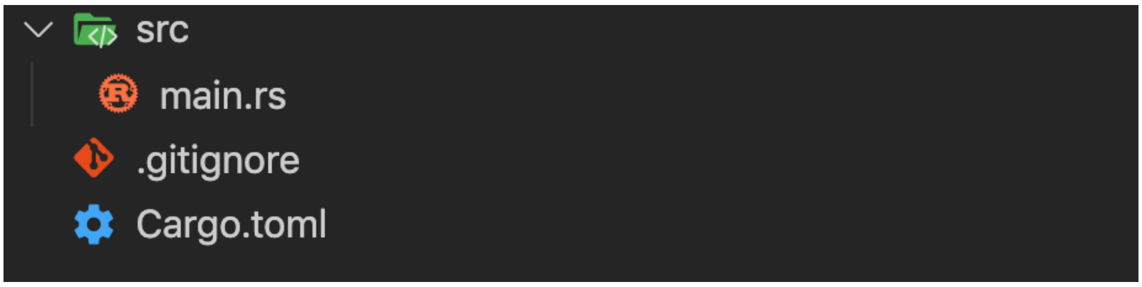
The web-sys crate and js-system crate are the essential tools for Rust to interact with JavaScript and the DOM.

3.2 Portal application

This is example of #[wasm_bindgen] showing how using native browser API such as Video API. We export a function to JavaScript, call it from the browser, and then execute the functionality from Rust code.

We first initialize project named video-api with Cargo:

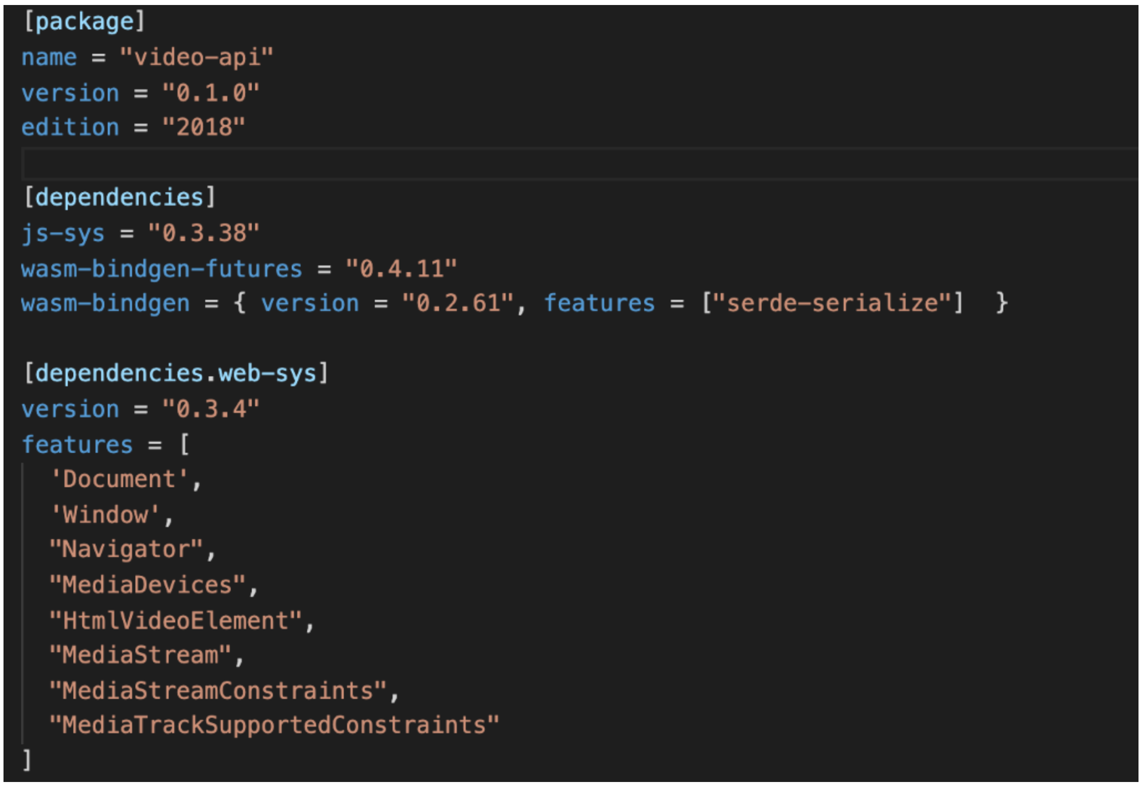
```
$ cargo new video-api
```



```
src
    main.rs
.gitignore
Cargo.toml
```

Figure 3. Initial rust project file structure.

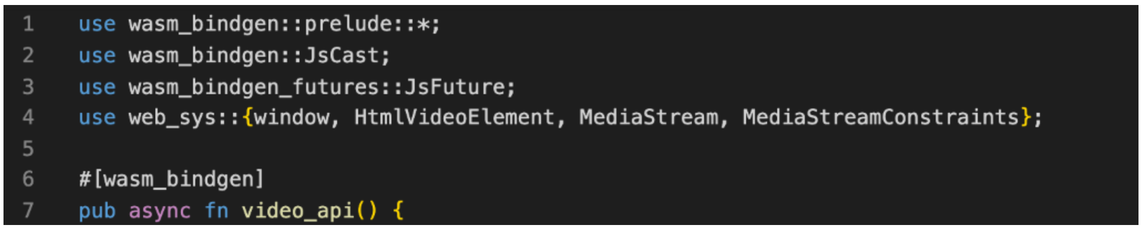
We then configure Cargo.toml file as showed in figure 4.



```toml
[package]
name = "video-api"
version = "0.1.0"
edition = "2018"

[dependencies]
js-sys = "0.3.38"
wasm-bindgen-futures = "0.4.11"
wasm-bindgen = { version = "0.2.61", features = ["serde-serialize"]  }

[dependencies.web-sys]
version = "0.3.4"
features = [
  'Document',
  'Window',
  "Navigator",
  "MediaDevices",
  "HtmlVideoElement",
  "MediaStream",
  "MediaStreamConstraints",
  "MediaTrackSupportedConstraints"
]
```

Figure 4. Cargo.toml is a manifest file in which can specify dependencies which the project required.

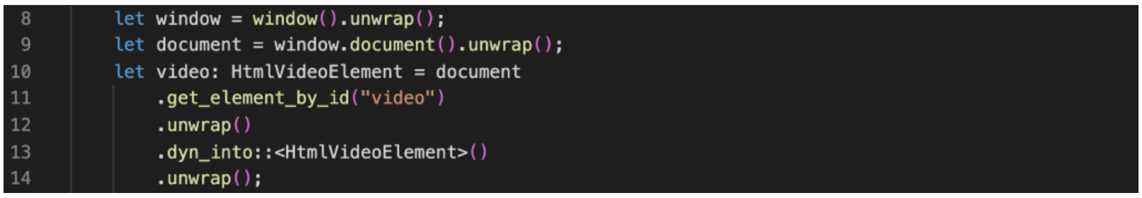
Figure 4 describes the project configuration. The package section shows the project information such as name, version, author, etc. The dependencies section specifies all the of required library for JavaScript binding and DOM interaction in Rust. Due to the DOM specification is massive so that in dependencies. In the web-sys section must specify which features are required for accessing browser video API.



```
1  use wasm_bindgen::prelude::*;
2  use wasm_bindgen::JsCast;
3  use wasm_bindgen_futures::JsFuture;
4  use web_sys::{window, HtmlVideoElement, MediaStream, MediaStreamConstraints};
5
6  #[wasm_bindgen]
7  pub async fn video_api() {
```

Figure 5. Import library and DOM objects.

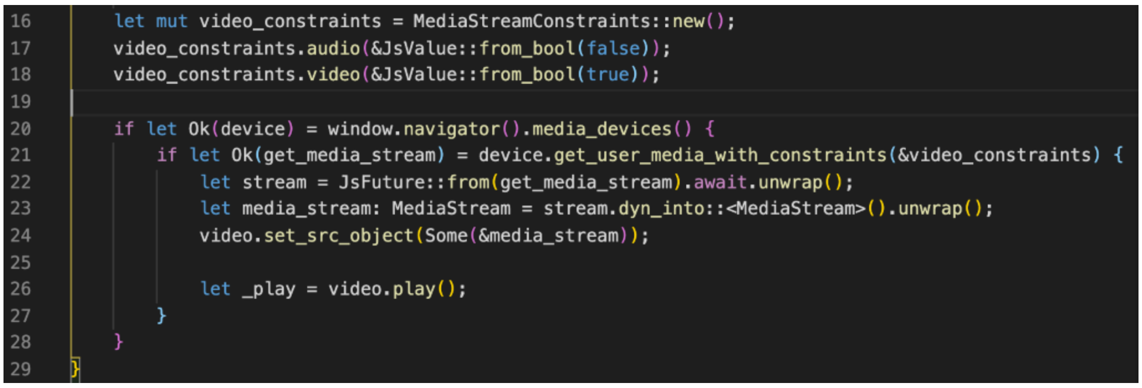
Required libraries are imported on the first 4 line. Since videp_api function interact with DOM API, therefore it needs #[wasm_bindge] on line 6.



```
8      let window = window().unwrap();
9      let document = window.document().unwrap();
10     let video: HtmlVideoElement = document
11         .get_element_by_id("video")
12         .unwrap()
13         .dyn_into::<HtmlVideoElement>()
14         .unwrap();
```

Figure 6. Select DOM element.

Function video_api initialized the window by invoking window function. After we invoke window function on line 8, Rust does not return the window object immediately but rather than an Option<Window> due to Rust is a statically typed safe language, it needs to call unwrap function on Option<Window> to get the window. If window object is unavailable unwrap function stop will the program. After that we select an element has id named “video”, since get_element_by_id returns a general HtmlElement type, it needs a type cast to HtmlVideoElement on line 13.



```
16     let mut video_constraints = MediaStreamConstraints::new();
17     video_constraints.audio(&JsValue::from_bool(false));
18     video_constraints.video(&JsValue::from_bool(true));
19
20     if let Ok(device) = window.navigator().media_devices() {
21         if let Ok(get_media_stream) = device.get_user_media_with_constraints(&video_constraints) {
22             let stream = JsFuture::from(get_media_stream).await.unwrap();
23             let media_stream: MediaStream = stream.dyn_into::<MediaStream>().unwrap();
24             video.set_src_object(Some(&media_stream));
25
26             let _play = video.play();
27         }
28     }
29 }
```

Figure 7. Setup Navigator and video’s constraints.

It setups a video's constraints with 2 properties which audio is set to false and video is set to true since we only interested in the video. It gets the user media device by calling method get_user_media_with_constraints which prompts the user for permission to use a media input which produces a MediaStream [16] object with tracks containing the requested types of media. That stream can include, for example, a video track such as a camera, video recording device, etc. Since the result from calling unwrap function from get_media_stream is a Rust Promise type. This type is different from JavaScript Promise. Fortunately, there is a JsFuture crate [17] which is imported at the beginning to convert Rust Promise to JavaScript Promise the result is store in a variable named media_stream. It then streams the image data which the array the of bytes to the video object. The image data collected from user camera to the video element. _play variable indicates that the call play method on video object is used for side effect.

Working with Rust is very strict and verbose, but those safety checks guarantee that the compile WebAssembly is run without errors.

We run the CL tool wasm-pack with the build command. It automatically takes the src/lib.rs file as entry file then compile and bundle everything in a folder call pkg.

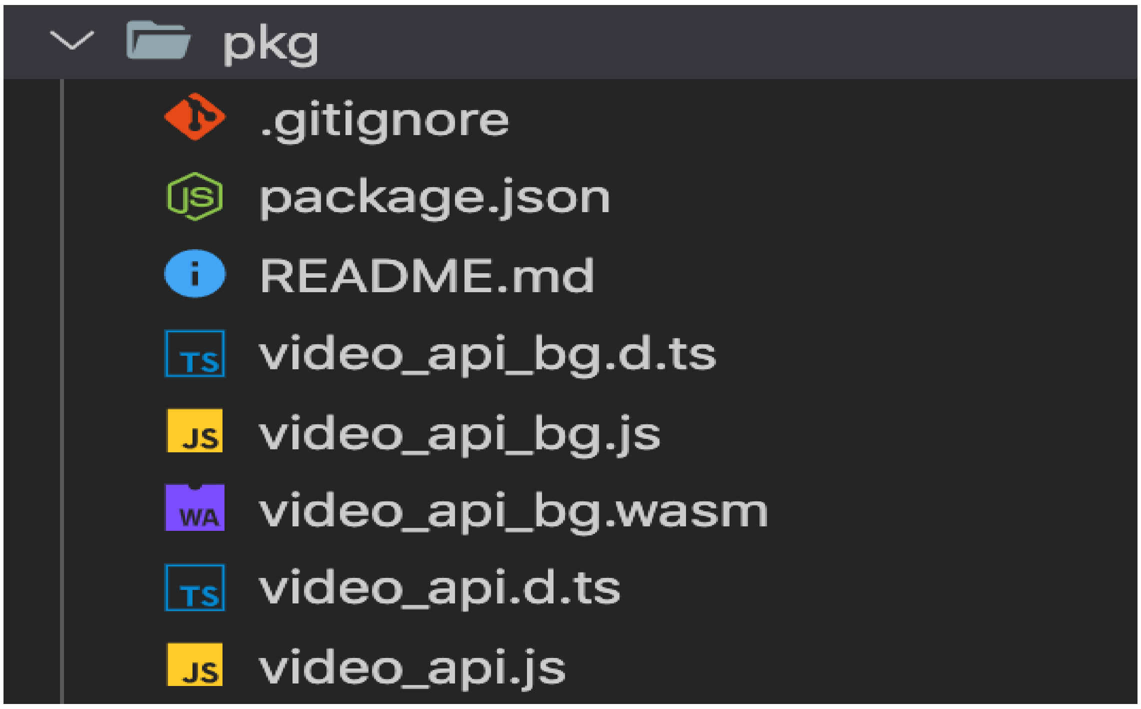
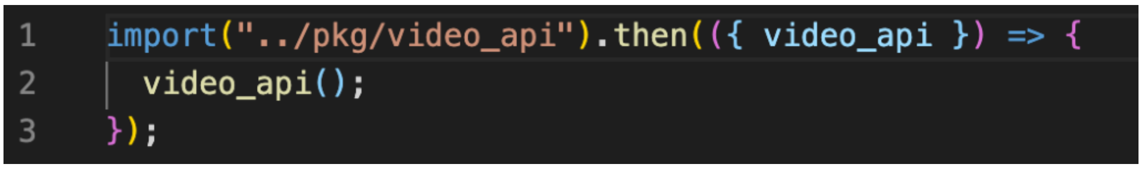

```
pkg
  .gitignore
  package.json
  README.md
  video_api_bg.d.ts
  video_api_bg.js
  video_api_bg.wasm
  video_api.d.ts
  video_api.js
```

Figure 8. pgk folder files structure.

Wasm-pack has compiled the Rust code into WebAssembly and setup the JavaScript binding. It generates two TypeScript for documenting and linting when using this module. There is a video_api_bg.js file which is a bridge code between JavaScript and WebAssembly. Its setup the error handler wrapper and a Shared Array Buffer [18] for WebAssembly to access which is highly efficient and secure.

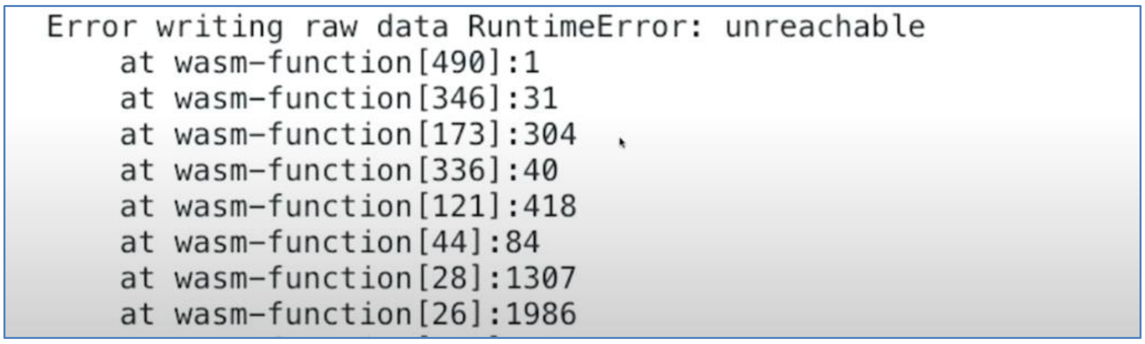
Due the compilation target is to the web. The pkg module can now be export and run in any browser that supports WebAssembly and in difference operating system without any installation.



```
import("../pkg/video_api").then(({ video_api }) => {
  video_api();
});
```

Figure 9. Using WebAssembly module.

Some caveat using WebAssembly are WebAssembly need to be fetched and stream chunks by chunks to the web. WebAssembly team and Webpack team are working on the loader to improve this import process. Due to the compact in WebAssembly code, debugging process is not very straight forward. There are a numerous of proposal to implement source map to tackle this problem [19].



```
Error writing raw data RuntimeError: unreachable
    at wasm-function[490]:1
    at wasm-function[346]:31
    at wasm-function[173]:304
    at wasm-function[336]:40
    at wasm-function[121]:418
    at wasm-function[44]:84
    at wasm-function[28]:1307
    at wasm-function[26]:1986
```

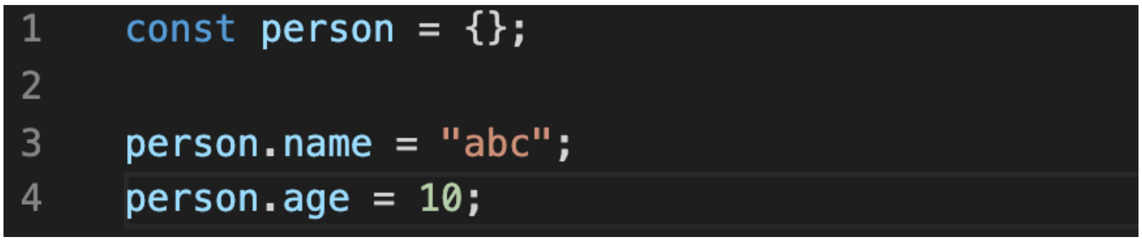
Figure 10. Error message in Using WebAssembly.

3.3 JavaScript performance vs WebAssembly performance

3.3.1 JavaScript performance

Browsers do not execute code directly from compiled executable file. Browsers have to download, parse, interpret, and JIT [20] which compile JavaScript during runtime not a head of time.

The below code snippet is a valid JavaScript code for accessing and assigning values. A dynamic object is created, and values can be access using dot notation and assign using equal sign. Variables and properties can be changed in any time in JavaScript which make it great to deal with JSON files, learn a language and prototyping. However, JavaScript compiler has to a lot of works to anticipate what are the types of the variables since the compiler does not has the blueprint of the object, a single property look has to go to number of operations such as checking for type error, undefined, look up from the prototype chain, value location in memory, etc.



```
const person = {};

person.name = "abc";
person.age = 10;
```

Figure 12. Access and assign value in JavaScript.

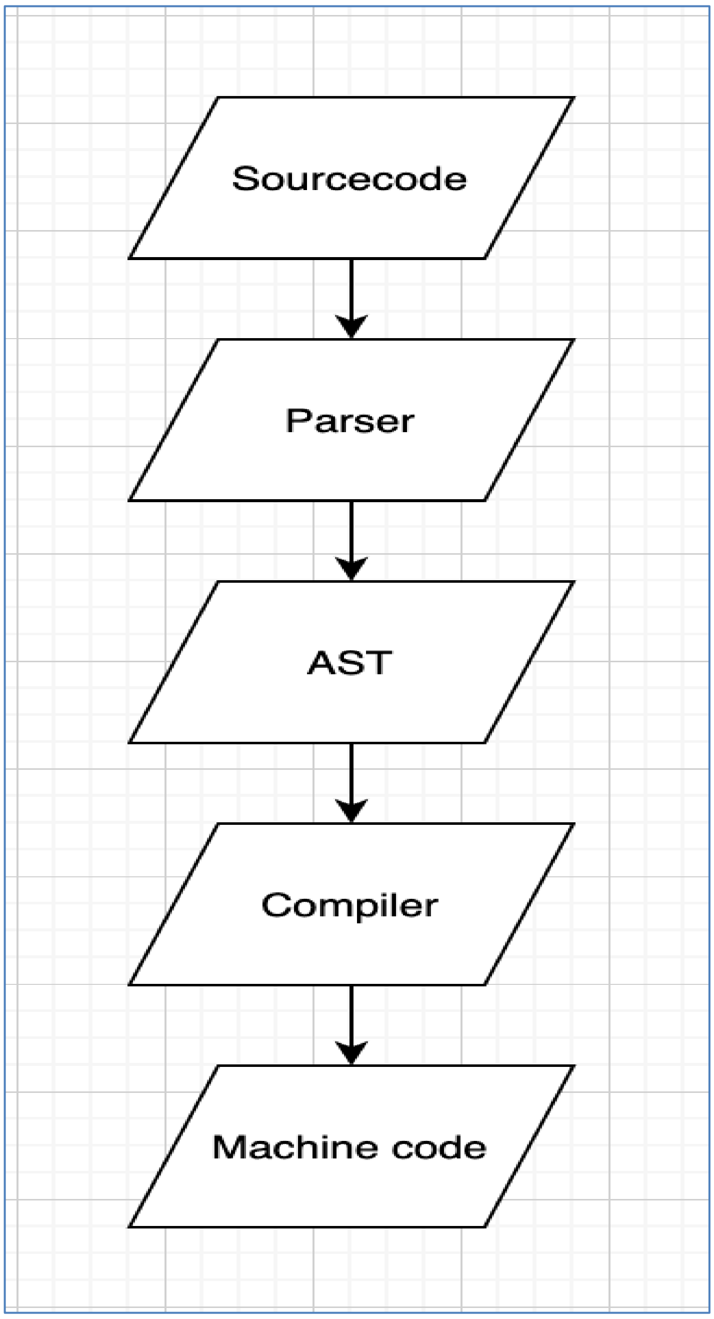


Figure 11. JavaScript compile flow.

JavaScript engine first read the source code as text. The text is run through the engine parser which turn the human readable source code to an AST [21] which is a format design for computer to read then the compiler takes this AST and generate machine code. The performance consumed work is at the compiling process of the compiler.

Compiling process is not one-way flow instead is it a loop that compile and generate a small pieces of machine code at the time and repeat so on and so forth. By compiling in repetition JavaScript can collect the runtime feedback to generate fast machine code.

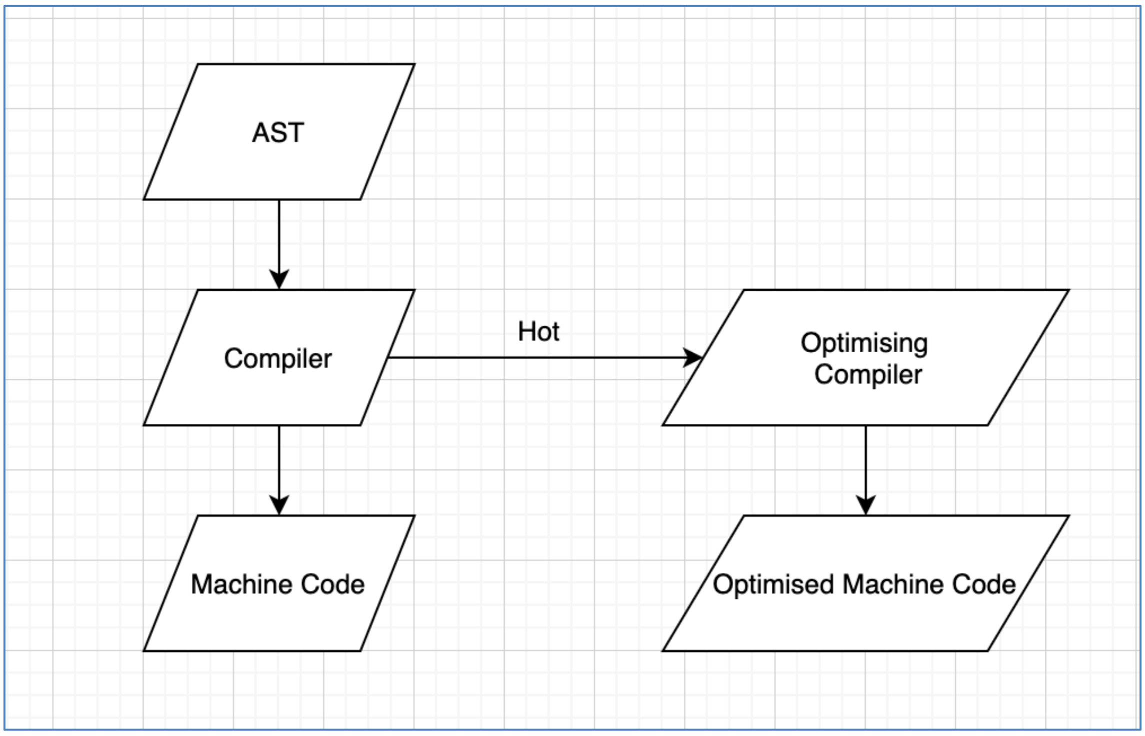


Figure 13. JavaScript (de)optimise flow.

When a function foo is called with a same object type multiple times.

foo({ x: 1 });

foo({ x: 1 });

foo({ x: 1 });

JavaScript engine bases on the feedback from the runtime to label foo function as hot then its forward the compiling process of foo function to the optimizing compiler which the speculation that foo takes in an object which has a property x with a number type. However, at some point in the code foo is called with another object type.

```
foo({ y: 2 });
```

The JavaScript has to decompile and pass the code back to the base compiler. This is where the performance dropped.

3.3.2 WebAssembly performance

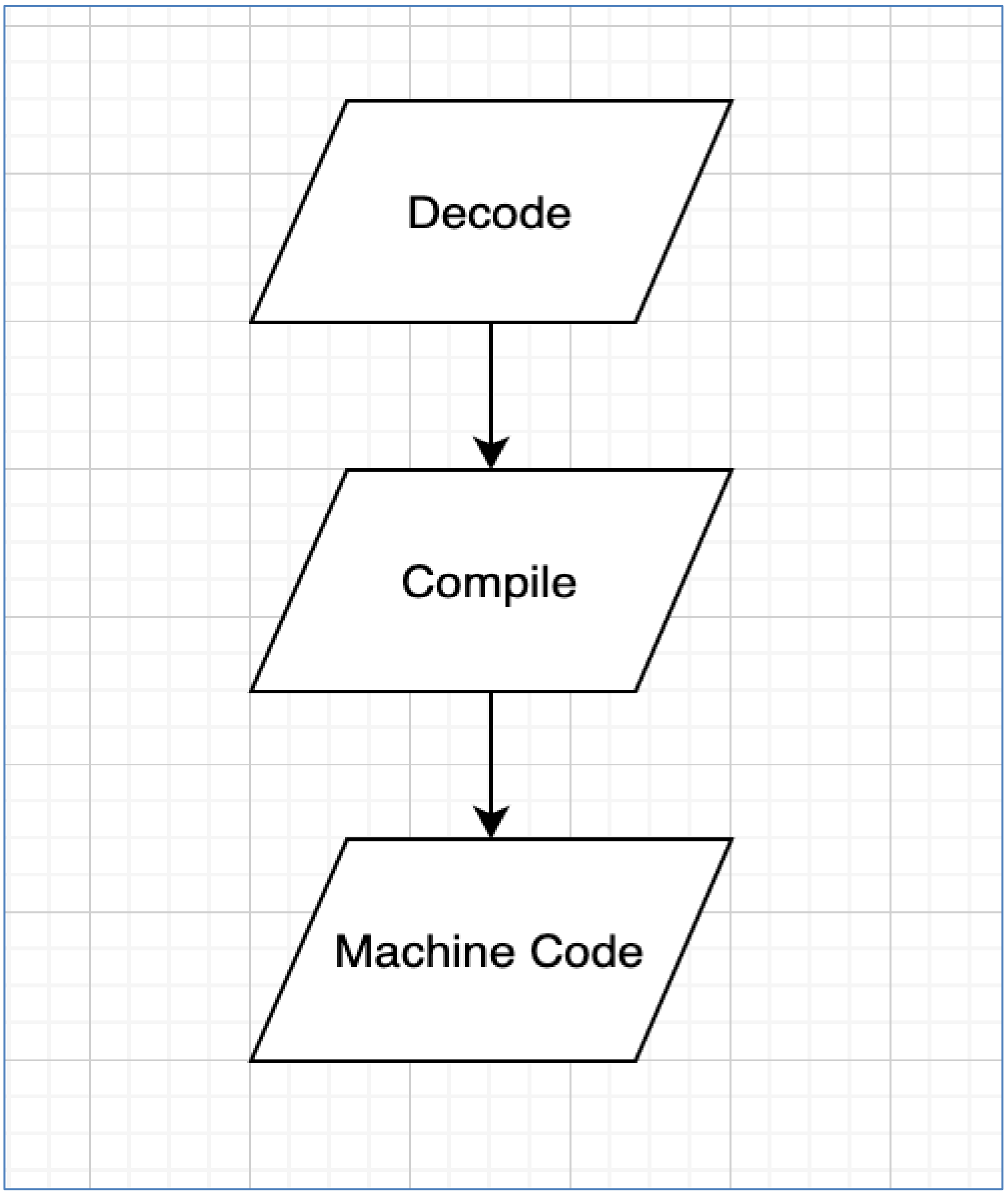


Figure 14. WebAssembly compile flow.

WebAssembly is in byte code format therefore it only needs to be decoded which is a faster process then parsing. Compiling process is also much faster due to most part of the assembly code is compiled before being served to the client. Due to the strictly typed, compiler can trust the code then compile to the optimized machine code without the worries of wrong speculation so that generated machine code does not need to go through optimize and deoptimize process.

When initialized a WebAssembly module, JavaScript with create new memory object which is called Shared Array Buffer. It can create by calling Memory instance in WebAssembly object which is shipped in browser and pass that object in it.

Shared Array Buffer is just a JavaScript object; therefore, JavaScript will also have access to these chunks of memory. Instead of using a memory address, they use an array index to access each box.

Managing memory is also different in WebAssembly from JavaScript. JavaScript is using GC (garbage collector) [22] which keeps looping and checks for unused memory. Using GC can make the programming language tend to be easy to use. However, trade-off is the performance loss. WebAssembly on the other handle is usually compiled from a supported manual memory management. Removing GC, so WebAssembly does not need to do the complicated cleaning process during runtime to keep track memory.

In practice, both JavaScript and WebAssembly are equally fast in performance. The differences are JavaScript code is passed as text to an interpreter then be optimized and de-optimized by an optimizing complier using analytic to generate machine code. On the other hand, WebAssembly is streamed to a WebAssembly complier and then to the optimizing compiler to generate machine code. Since WebAssembly does not go through several optimization and de-optimization process therefore it has better runtime.

3.3.3 Performance comparison

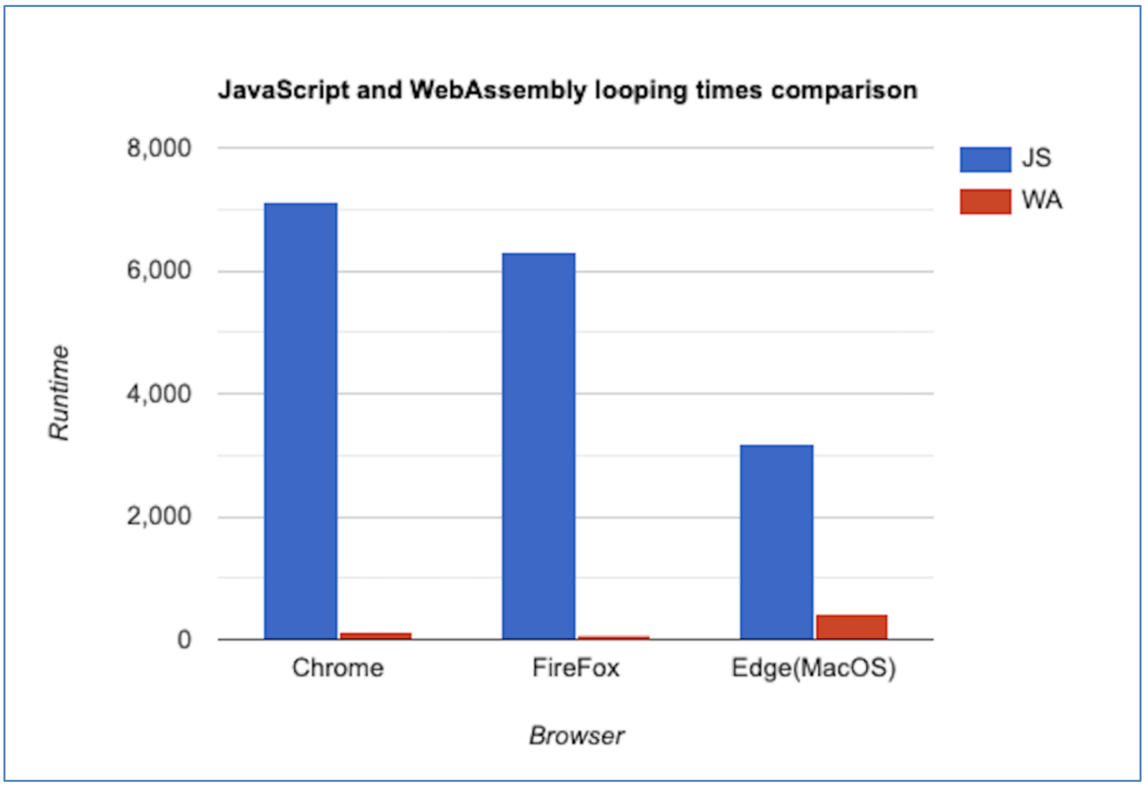


Figure 15. Looping performance comparison.

Figure 15 shows the comparison between JavaScript code and WebAssembly code in three browsers in MacOS operating system. The performed task is a single function that take an initial input as an integer then perform a count operation add up to 10000000 times of repetition. WebAssembly is manually compiled from Rust by just using cargo target to the web. The result shows that replacing a native JavaScript module with a JavaScript module utilizing WebAssembly can boost web application running in MacOS operating system from 6 times to 70 times more performance.

Figure 16 shows the comparison between JavaScript code and WebAssembly code in Chrome, FireFox and Edge in Window 10 operating system. WebAssembly is compile using wasm-pack build command target to the web. The performed task is two function, the first function is checking prime number and the second is finding the index of that prime. The function takes an input as integer which specify the index of the prime. Figure in the chart show the runtime calculation of the 20000th prime number. The result shows that replacing a native JavaScript module with a JavaScript module utilizing WebAssembly can boost web application running in Window 10 heavy calculation times up to 3 times faster.

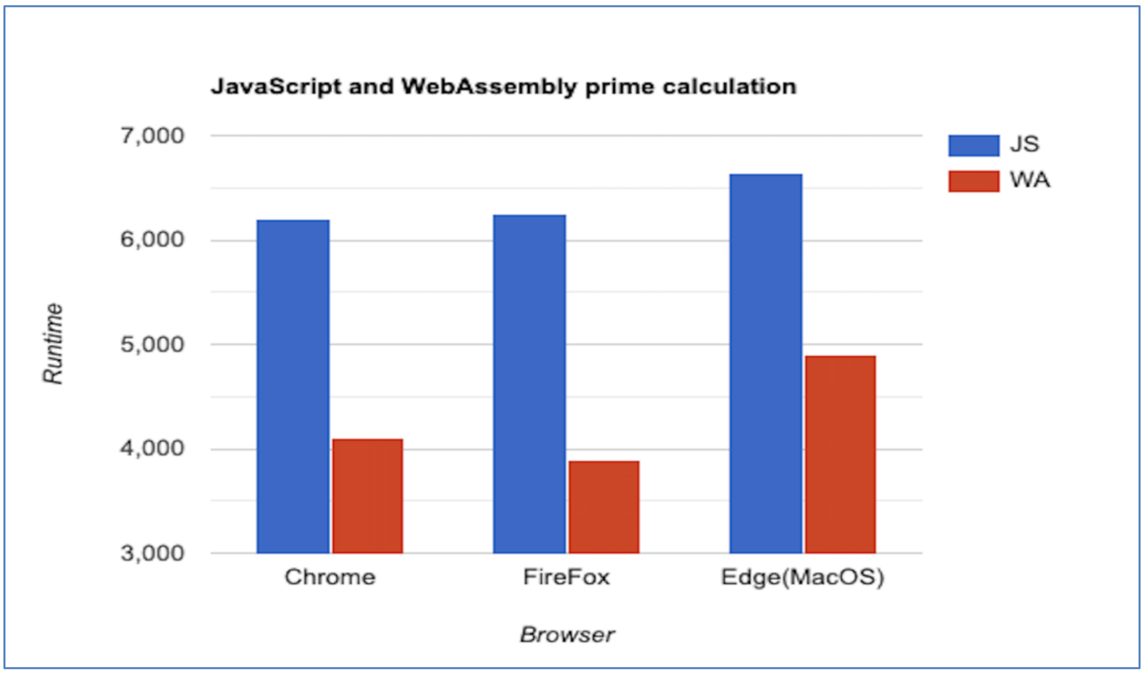


Figure 16. Prime performance comparison.

The reason for the performance drops from 6-70 times to 2 times is that. In the first test, the task is quite simple, and it is compiling manual just using cargo. This can benefit the WebAssembly performance; however, the trade-off are lacking safety check, compatibilities and error handle wrapper. The second one is using wasm-pack which is a delicate tool to compile to Webassembly, it automatically generates JavaScript binding and fallback option for compatibilities. This cause the drop in performance; the WebAssembly is continuously working this to improve the performance.

By far, WebAssembly outperform JavaScript on heavy task such as looping and calculation. The figure below shows the comparison between JavaScript code and WebAssembly code generate by wasm-pack both performing of inverting colors of a video. The video data in capture from user camera, then being mutated their colors then those data is stream to the canvas.

Figure 17 show the comparison between JavaScript code and WebAssembly code in Chrome browser in MacOS operating system. Result shows that Rust generated WebAssembly code is twice as slow as JavaScript code. We now analyze the below code snippet to explain the WebAssembly sudden performance drop.

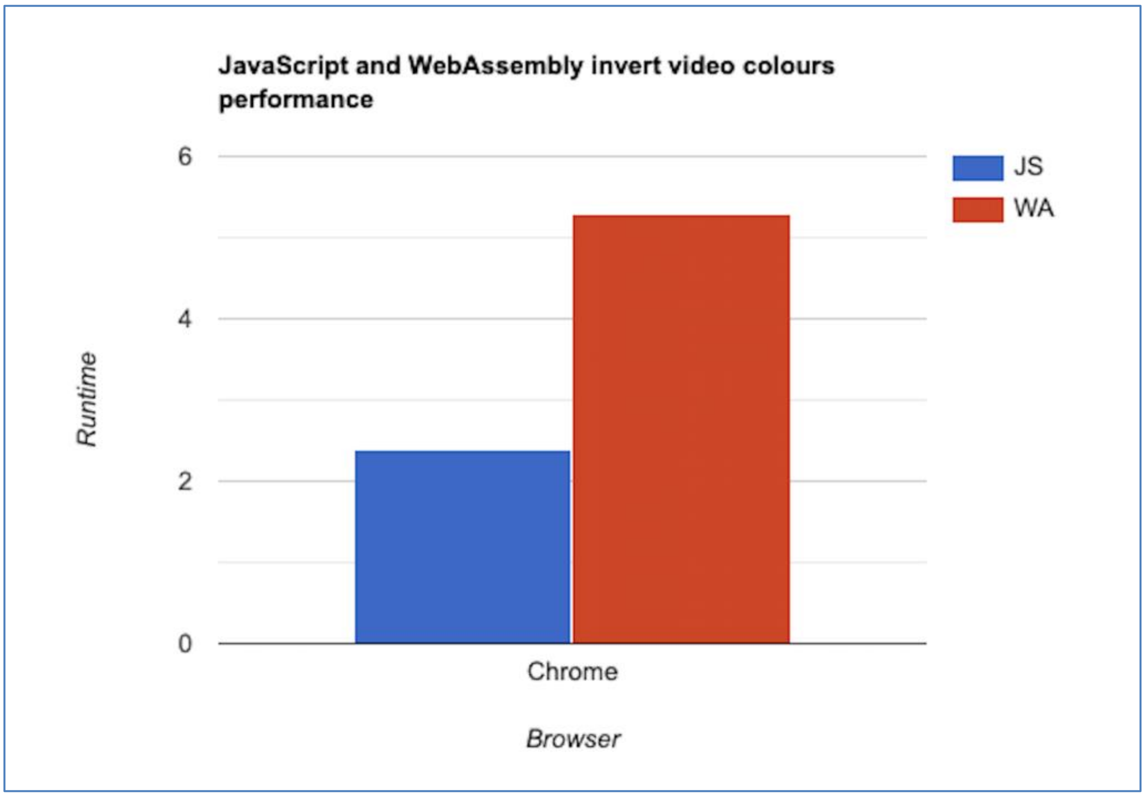
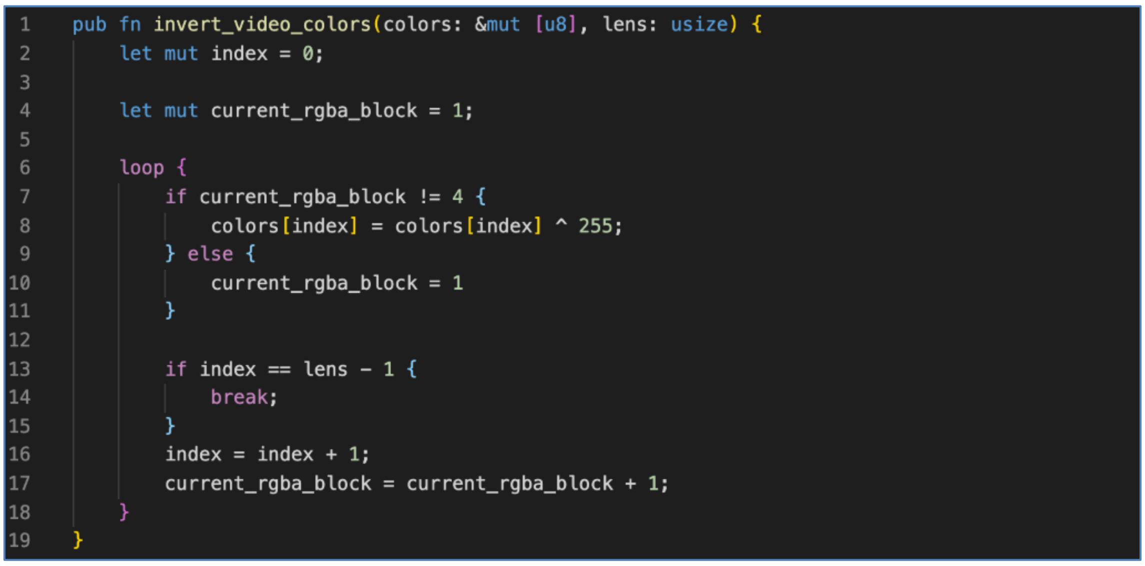


Figure 17. Invert video colours performance comparison in MacOS.



```rust
pub fn invert_video_colors(colors: &mut [u8], lens: usize) {
    let mut index = 0;

    let mut current_rgba_block = 1;

    loop {
        if current_rgba_block != 4 {
            colors[index] = colors[index] ^ 255;
        } else {
            current_rgba_block = 1
        }

        if index == lens - 1 {
            break;
        }
        index = index + 1;
        current_rgba_block = current_rgba_block + 1;
    }
}
```

Figure 18. Invert colours function in Rust.

All the task we made so far is looping function and calculating prime number only accept an integer as an argument. The invert_video_colors function accepts a pointer to the image array buffer as an argument. Since WebAssembly was release as an MVP so that it is missing some feature. At the moment, communicate between JavaScript and We Assembly though variables such as number and string highly efficient, however communicating though pointer to memory is rather expensive process.

The issue can be actress by serialize the image data stream into string format then deserialized back to array stream type when it is passed to WebAssembly. This is just temporary solution due to some performance loss and (de)serialize process might cause the loss in data integrity.

For instance, BSON format file contain not just text but also data type such as number types, (de)serialize that will cause the data miss-type which lead to bug and error. In the near future, there will be more robust binding which and transfer memory from JavaScript heap to WebAssembly linear memory so solve this issue. It can still be proven that WebAssembly perform better by only measure the runtime of the specify part from line 2 to line 18 which skip that communication through pointer from JavaScript to WebAssembly. Runtime performance is shown in the figure below.

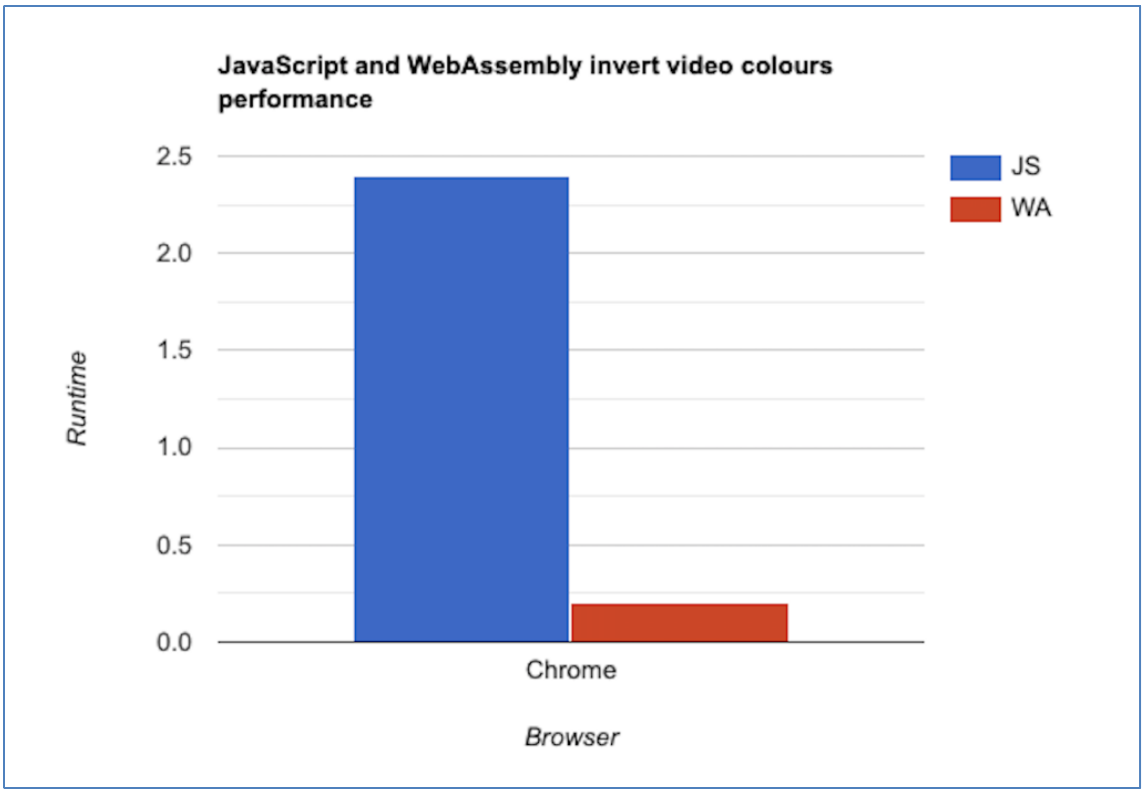
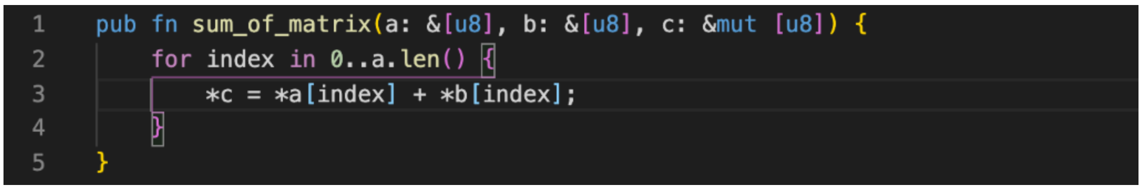


Figure 19. Performance comparison.

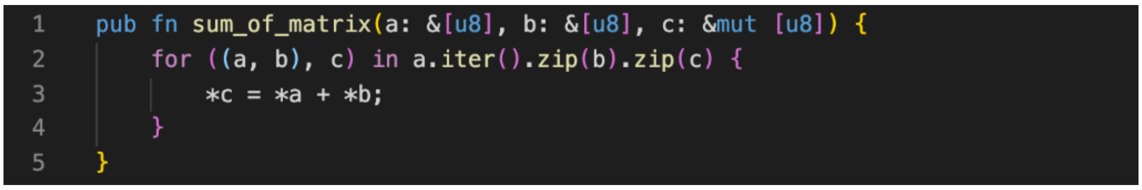
3.3.4 Single instruction, multiple data (SIMD)

SIMD enabled computers to perform a single operation on multiple elements. SIMD most common use case is in heavy task such as contrast in a digital image or adjusting the volume of digital audio [23]. The basic usage of SIMD is available in Rust in a minimum use case. For example:



```rust
pub fn sum_of_matrix(a: &[u8], b: &[u8], c: &mut [u8]) {
    for index in 0..a.len() {
        *c = *a[index] + *b[index];
    }
}
```

Figure 20. Matrix calculation.



```rust
pub fn sum_of_matrix(a: &[u8], b: &[u8], c: &mut [u8]) {
    for ((a, b), c) in a.iter().zip(b).zip(c) {
        *c = *a + *b;
    }
}
```

Figure 21. Matrix calculation with SIMD.

Summing two matrices from the first two array slice then assign those new value to the third argument. The simple solution on figure 20 is looping through individual element from the array slice then add those value to each block in c array slice. However, with SIMD now shipped, the same task can be written as in figure 21.

The compilers will apply SIMD to calculate. For instance, both a and b array slice contain 16 elements. Each element is a u8, and so that means that each slice would be 128 bits of data. Using SIMD, compile put both a and b into 128-bit registers then perform add operator on those registers at the same times result in much faster perform.

There are some few drawbacks such as: not all CPU has this feature so in practice there are usually a check for compatibility.

4 WEBASSEMBLY RISKS

4.1 Emscripten – A toolchain for compiling WebAssembly

4.1.1 Overview of Emscripten compiler

According to Emscripten, “Emscripten is an Open Source LLVM to JavaScript compiler” [24]. LLVM is a compiler framework, it is briefly a framework to build up the compiler of a programming language. The initial idea of Emscripten is to compile C/C++ code or any other code which is translatable to LLVM bitcode into JavaScript (figure 22). The output of this process is the code in *asm.js*. *Asm.js* is not a new language, it is a low-level and strict subset of JavaScript and it is supported by all major web browsers.

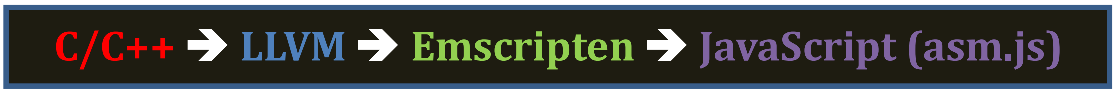


Figure 22. Compiling process of Emscripten.

Emscripten not only helps C/C++ developers to improve the backend performance but also assist Web developers to utilize existing native utilities and libraries. Let’s take a look at the following examples:

- Example 1: We have a website which has “Find the shortest path” feature. User will choose the starting point (S) and the destination (D), the website should return the shortest path from S to D. Normally the process of calculating the path will be handle in the backend by using any backend language such as Python, Rust, C/C++,… For instance, we already had the algorithm in C/C++ and now we want the process to be run on client-side in order to reduce the workload on server-side. With Emscripten, it is easy for C/C++ developers to porting the code from C/C++ to JavaScript automatically without having to learn JavaScript.
- Example 2: Handle SQL database directly from the browser. Usually the database is stored in the backend and any SQL query will be handled on server-side. Nowadays, with the help of Emscripten, Web developers can process SQL query directly from the browser with “sql.js” project. “sql.js is a port of SQLite to Webassembly, by compiling the SQLite C code with Emscripten, with contributed math and string extension functions included.” [25]

Converting C/C++ code into JavaScript is not the only feature that Emscripten can do. Another powerful feature of Emscripten is compiling C/C++ code into WebAssembly binary (WASM). Emscripten uses asm2wasm tool, which is a part of *Binaryen*, to compile C/C++ code into WASM (figure 23). *Binaryen* is a WebAssembly infrastructure library, written in C++, it helps the process of compiling to WebAssembly simple, quick and effective. For example, given the code in figure 24 which describes a sub function in C++, Emscripten would compile it into *asm.js* as was shown in figure 25 and finally convert *asm.js* code into WebAssembly by using *asm2wasm* tool (figure 26). From figure 25, we can clearly see that there is a small difference between *asm.js* and the original JavaScript which is the *bitwise OR* operator. For an easy explanation, the bitwise OR with zero "| 0" converts a value to an integer and ensures that the type of a value is correctly converted when the function is called from outside code. As the result the code can simply perform native integer operations hence significantly improve the performance.

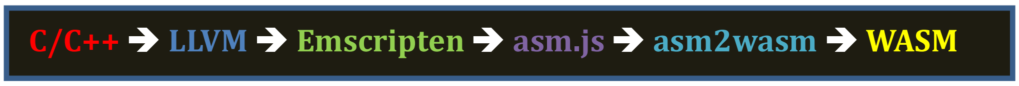
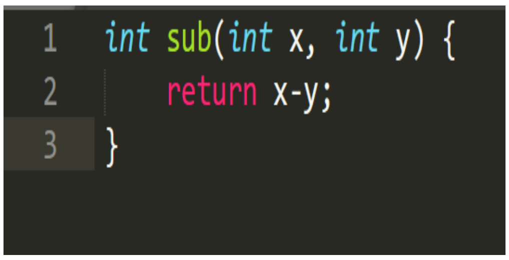
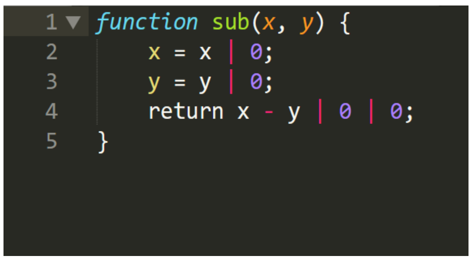


Figure 23. Compiling process from C/C++ to WASM.



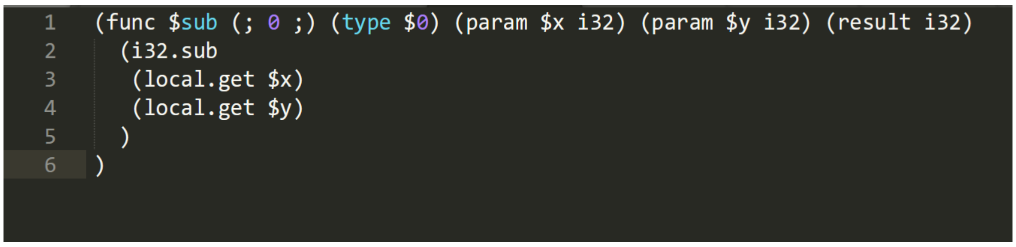
```
int sub(int x, int y) {
    return x-y;
}
```

Figure 25. Sub function in C++.



```
function sub(x, y) {
    x = x | 0;
    y = y | 0;
    return x - y | 0 | 0;
}
```

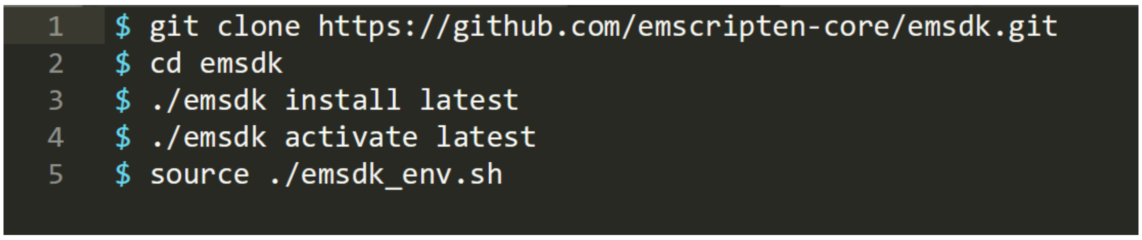
Figure 24. Sub function in ASM.JS.



```
(func $sub (; 0 ;) (type $0) (param $x i32) (param $y i32) (result i32)
  (i32.sub
   (local.get $x)
   (local.get $y)
  )
)
```

Figure 26. Sub function in WebAsemby Text Format.

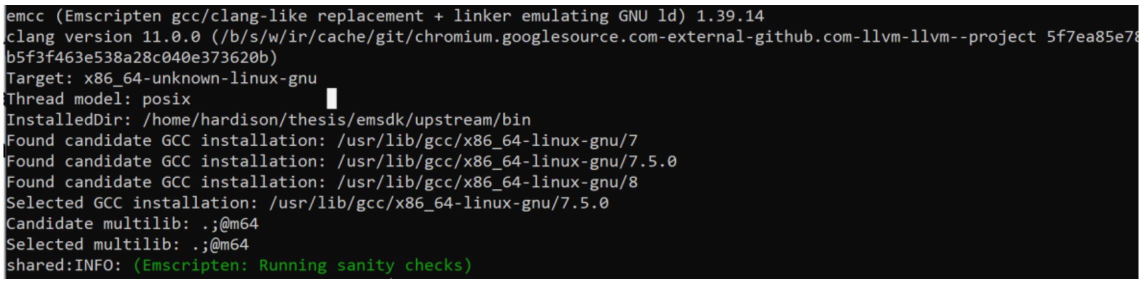
Before we go further, we need to install Emscripten compiler first. In this tutorial, we will use Ubuntu 18.04 LTS to install Emscripten toolchain. A precompiled toolchain can be downloaded via GitHub and installed via Bash script. On the terminal, use the commands in Figure 27 one by one [26].



```
$ git clone https://github.com/emscripten-core/emsdk.git
$ cd emsdk
$ ./emsdk install latest
$ ./emsdk activate latest
$ source ./emsdk_env.sh
```

Figure 27. Emscripten's installation commands.

We can check the installation result by using command ”emcc -v”. If there is no warning (yellow color) and error (red color) that means the installation is succeeded. Figure 28 is an example when the installation is successfully completed.



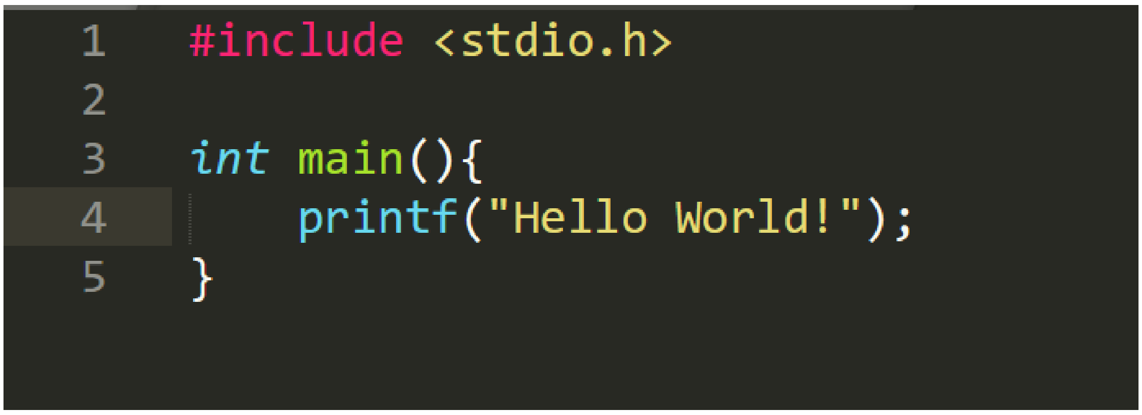
```
emcc (Emscripten gcc/clang-like replacement + linker emulating GNU ld) 1.39.14
clang version 11.0.0 (/b/s/w/ir/cache/git/chromium.googlesource.com-external-github.com-llvm-llvm--project 5f7ea85e7
b5f3f463e538a28c040e373620b)
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /home/hardison/thesis/emsdk/upstream/bin
Found candidate GCC installation: /usr/lib/gcc/x86_64-linux-gnu/7
Found candidate GCC installation: /usr/lib/gcc/x86_64-linux-gnu/7.5.0
Found candidate GCC installation: /usr/lib/gcc/x86_64-linux-gnu/8
Selected GCC installation: /usr/lib/gcc/x86_64-linux-gnu/7.5.0
Candidate multilib: .;@m64
Selected multilib: .;@m64
shared:INFO: (Emscripten: Running sanity checks)
```

Figure 28. Result of "emcc -v" command.

4.1.2 “Hello World!” from C/C++ to WebAssembly

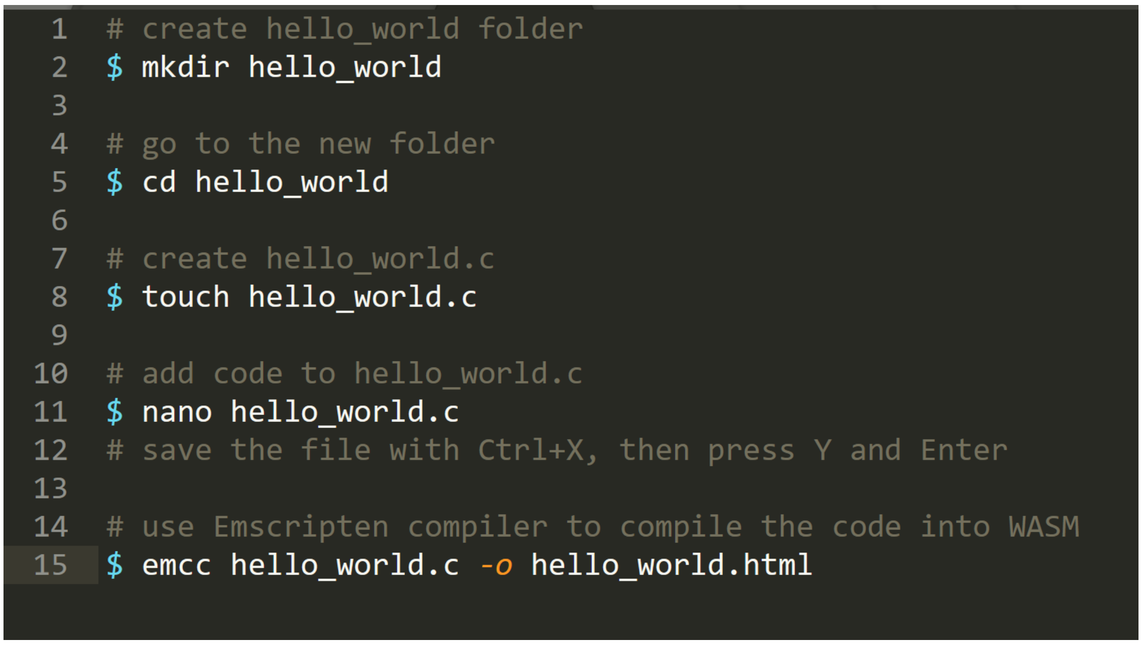
In the previous section, we already had a brief understanding about Emscripten compiler and how to install it. Now we will create our first compiled WebAssembly program with Emscripten.

From the terminal, we create a new folder “hello_world”. Then we create a simple C program which will print “Hello World!” to the console name *hello_world.c* (figure 28). Next we will compile that code into WebAssembly using Emscripten compiler with the output as HTML file. All the steps can be done via the terminal as described in figure 30.



```c
#include <stdio.h>

int main(){
    printf("Hello World!");
}
```

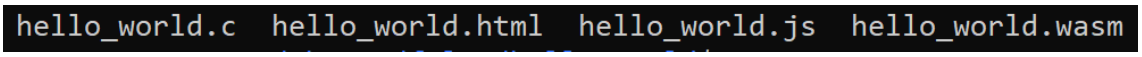
Figure 29. hello_world.c.



```
# create hello_world folder
$ mkdir hello_world

# go to the new folder
$ cd hello_world

# create hello_world.c
$ touch hello_world.c

# add code to hello_world.c
$ nano hello_world.c
# save the file with Ctrl+X, then press Y and Enter

# use Emscripten compiler to compile the code into WASM
$ emcc hello_world.c -o hello_world.html
```

Figure 30. Process of compiling C/C++ to WebAssembly.

Emscripten will compiled the program into 3 separate files: 1 WebAssembly binary (.wasm), 1 JavaScript (.js) and 1 HTML (.html) as showed in figure 31. The WASM binary is the actual compiled program from C/C++ which will be loaded by JavaScript and display the result to HTML. Check the output by using “ls” command from the terminal.



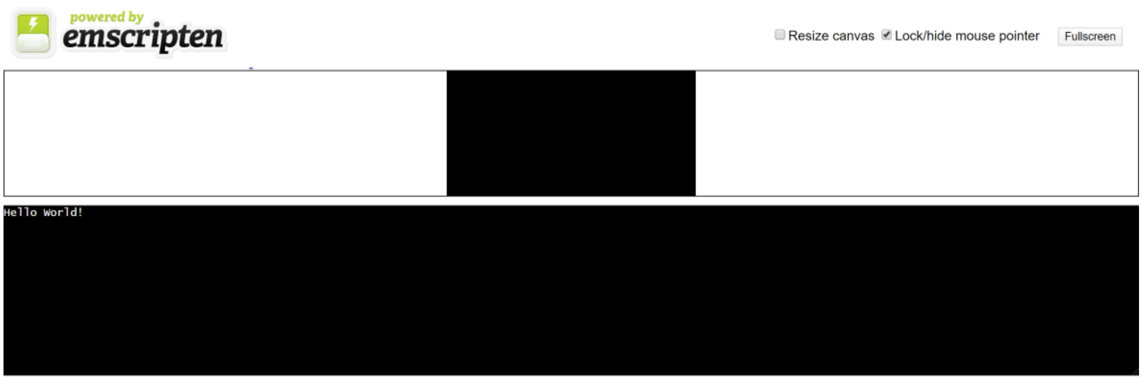
```
hello_world.c  hello_world.html  hello_world.js  hello_world.wasm
```

Figure 31. Result of the compilation.

In order to observe the result, we will need a simple HTTP server to open the HTML file. Fortunately, Emscripten compiler provides *"emrun"* – a command line tool that helps to setup a local web server:

```
$ emrun --no-browser --port 8080 .
```

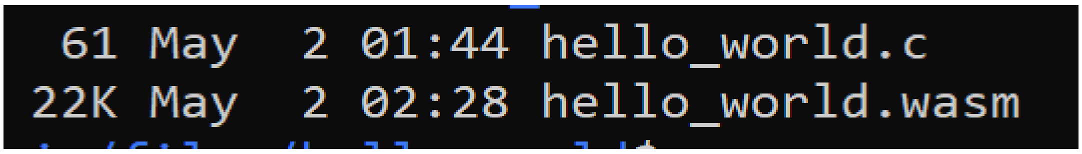
Now we can visit the HTML by open the link *"localhost:8080/hello_world.html"* from the browser. Figure 32 shows the content of *hello_world.html* in the browser. *"Hello World!"* string was printed in the black box which prove that the compiled program is working sucessfully.



```
Hello World!
```

Figure 32. hello_world.html from the browser.

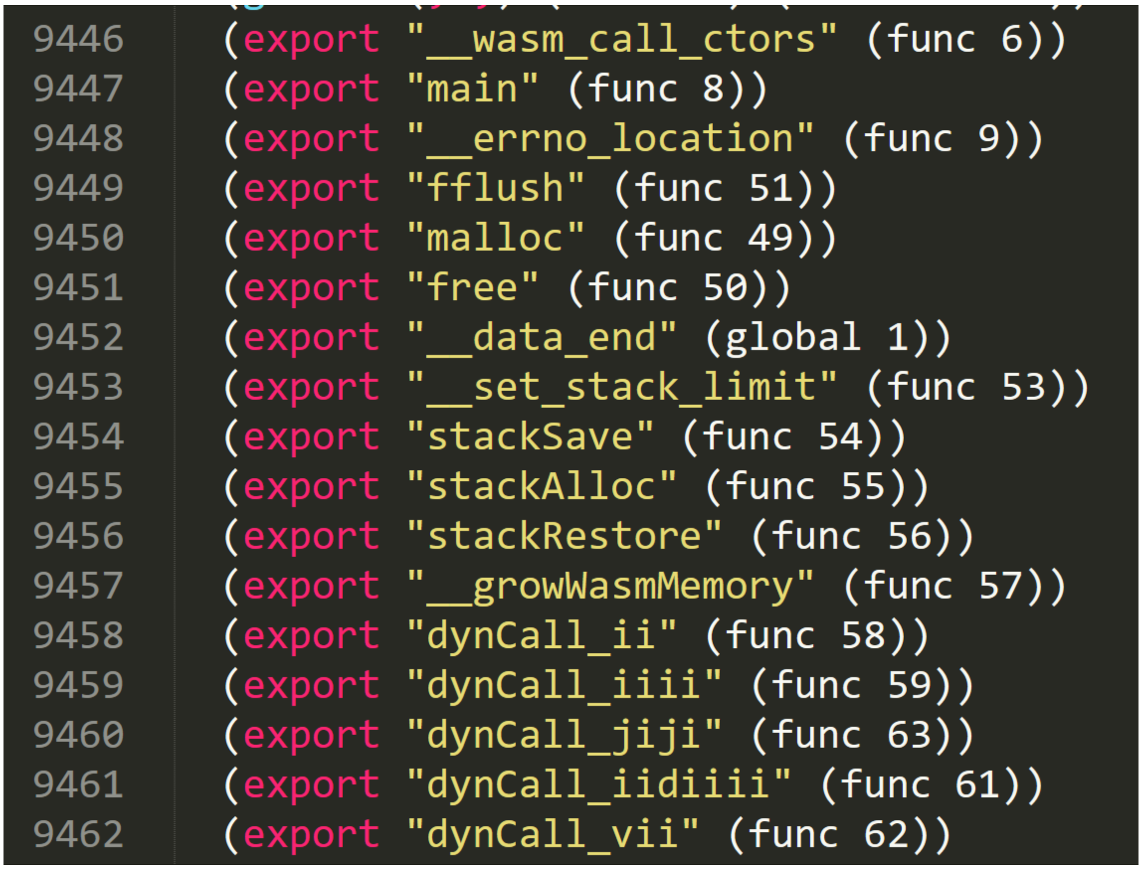
Let's take a deeper look into the compiled files. First we have a WASM binary which is the actual compiled code from C/C++ code. The original size of the C file (hello_world.c) is 61 bytes but the compiled WASM binary (hello_world.wasm) has 22 kilobytes in size (figure 33). This is because of the "stdio.h" import, more specifically the "printf" function. Since WebAssembly does not have "printf" function to call remotely from inside the code, Emscripten has to compiled the "printf" module into WebAssembly (including all the modules that is used by "printf").



```
 61 May  2 01:44 hello_world.c
22K May  2 02:28 hello_world.wasm
```

Figure 33. The size of C file and WASM binary.

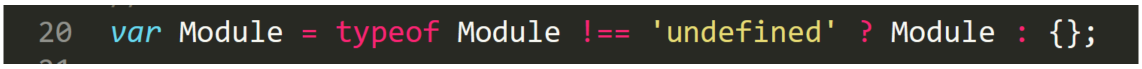
We use *wasm2wat* tool which is part of the WebAssembly Binary Toolkit to convert the WebAssembly binary into WebAsembly text format. From the converted file, in the *Export* section, we will see WebAssembly exports ”main” function as funtion number 8 that we can call from JavaScript later as showed in line 9447 of figure 34.



```
9446    (export "__wasm_call_ctors" (func 6))
9447    (export "main" (func 8))
9448    (export "__errno_location" (func 9))
9449    (export "fflush" (func 51))
9450    (export "malloc" (func 49))
9451    (export "free" (func 50))
9452    (export "__data_end" (global 1))
9453    (export "__set_stack_limit" (func 53))
9454    (export "stackSave" (func 54))
9455    (export "stackAlloc" (func 55))
9456    (export "stackRestore" (func 56))
9457    (export "__growWasmMemory" (func 57))
9458    (export "dynCall_ii" (func 58))
9459    (export "dynCall_iiii" (func 59))
9460    (export "dynCall_jiji" (func 63))
9461    (export "dynCall_iidiiii" (func 61))
9462    (export "dynCall_vii" (func 62))
```

Figure 34. Export section of WebAssembly.

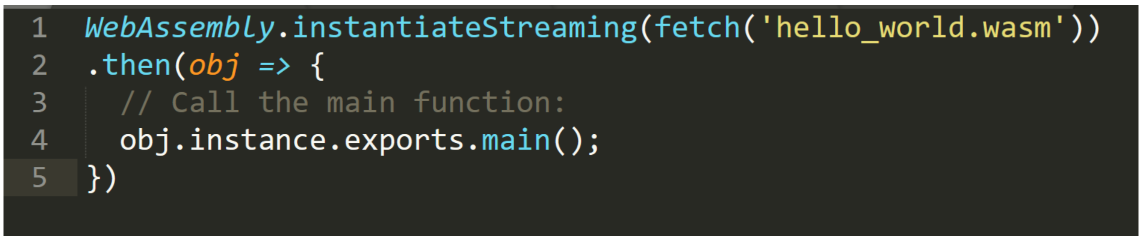
Now we will discuss about how Emscripten call the ”main” function from JavaScript. In order to load and use exported functions from WebAssembly, Emscripten authors created a Module object which they called ”An interface to the outside world”.



```
var Module = typeof Module !== 'undefined' ? Module : {};
```

Figure 35. Defining Module object.

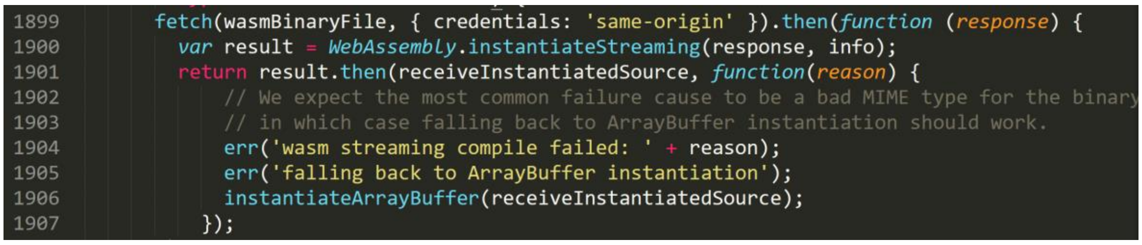
Before we can use the exported functions from WebAssembly binary, we first need to fetch the file into memory, then instantiate and run it. Emscripten uses *Fetch* API and *WebAssembly* object to complete this task. ”The WebAssembly JavaScript object acts as the namespace for all WebAssembly-related functionality”, according to MDN Web docs [27]. WebAssembly object is currently supported by most major browsers such as Firefox, Chrome, Edge… The fastest way to fetch a WASM binary is using WebAssembly.instantiateStreaming() method, which will compiles and instantiates the WASM binary from the **fetch** function's result. For example if we want to load ”hello_world.wasm” and call the ”main” function, we can simply use the code in figure 36.



```
WebAssembly.instantiateStreaming(fetch('hello_world.wasm'))
.then(obj => {
  // Call the main function:
  obj.instance.exports.main();
})
```

Figure 36. Example of fetching and instantiating hello_world module.

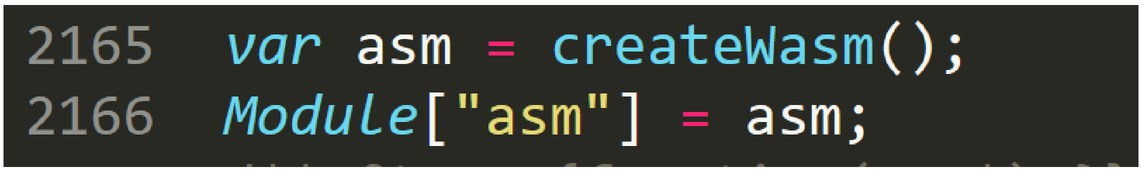
The actual code from the compiled JavaScript has a small diffference but the idea is still the same. The variable *wasmBinaryFile* is the name of the WebAssembly binary ”hello_world.wasm”. This process belongs to *createWasm* function.



```
fetch(wasmBinaryFile, { credentials: 'same-origin' }).then(function (response) {
  var result = WebAssembly.instantiateStreaming(response, info);
  return result.then(receiveInstantiatedSource, function(reason) {
      // We expect the most common failure cause to be a bad MIME type for the binary
      // in which case falling back to ArrayBuffer instantiation should work.
      err('wasm streaming compile failed: ' + reason);
      err('falling back to ArrayBuffer instantiation');
      instantiateArrayBuffer(receiveInstantiatedSource);
    });
```

Figure 37. Fetching and instantiating the module in Emscripten.

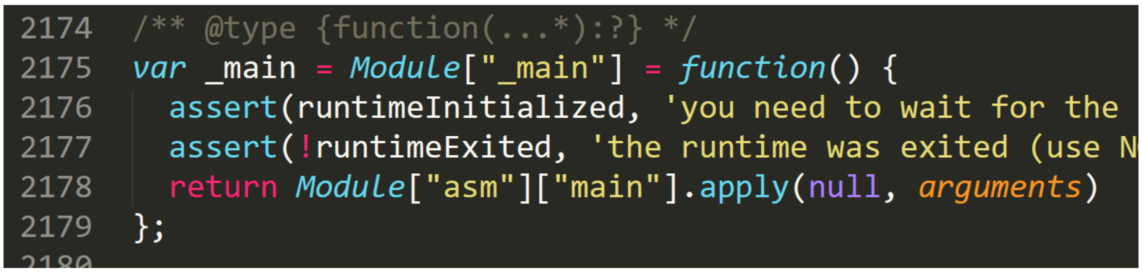
The **createWasm** function returns a list of exported functions and it will be stored in Module[”asm”].



```
var asm = createWasm();
Module["asm"] = asm;
```

Figure 38. Emscipten stores exported function in Module["asm"].

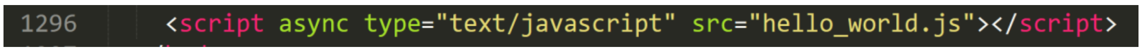
The ”main” function now can be called by using *Module[”asm”][”main”].* Emscripten assigns the ”main” function as *Module[”_main”]* for later usage.



```
/** @type {function(...*):?} */
var _main = Module["_main"] = function() {
  assert(runtimeInitialized, 'you need to wait for the
  assert(!runtimeExited, 'the runtime was exited (use N
  return Module["asm"]["main"].apply(null, arguments)
};
```

Figure 39. Emscripten assigns Module["_main"] as the "main" function.

Finally the JavaScript will call the main function and return the value of it. In this case we will have ”Hello World!” string as a result. The last part of the process is importing the JavaScript in the HTML file as showed in figure 40.



```
  <script async type="text/javascript" src="hello_world.js"></script>
```

Figure 40. Import JavaScript file in HTML.

4.1.3 Using JavaScript in C/C++

Emscripten toolchain also comes with another powerful feature which allows us to call JavaScript from C/C++. By importing “emscripten.h” library in the code, we can write “inline JavaScript” using EM_ASM() macro. According to Emscripten, “EM_ASM() is a convenient syntax for inline JavaScript. This allows you to declare JavaScript in your C code ‘inline’, which is then executed when your compiled code is run in the browser” [28].

Let’s create a simple program that will display an alert using EM_ASM() macro. First we create a new folder for the program with ”mkdir” command.

```
$ mkdir inline_javascript
```

Then we create a new source code file name ”inline_javascript_alert.c”

```
$ touch inline_javascript_alert.c
```

Next we want to add the code to the file. Figure 41 shows the code that we will use. If the code is successfully compiled with Emscripten and loaded to the browser, it will display an alert with ”Hello World!” message.

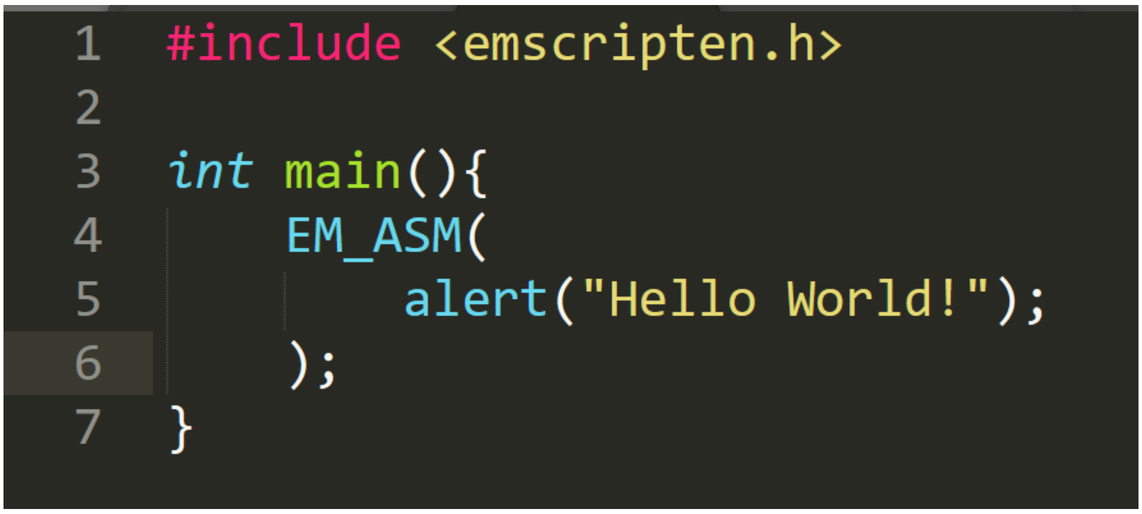

```c
#include <emscripten.h>

int main(){
	EM_ASM(
		alert("Hello World!");
	);
}
```

Figure 41. inline_javascript_alert.c.

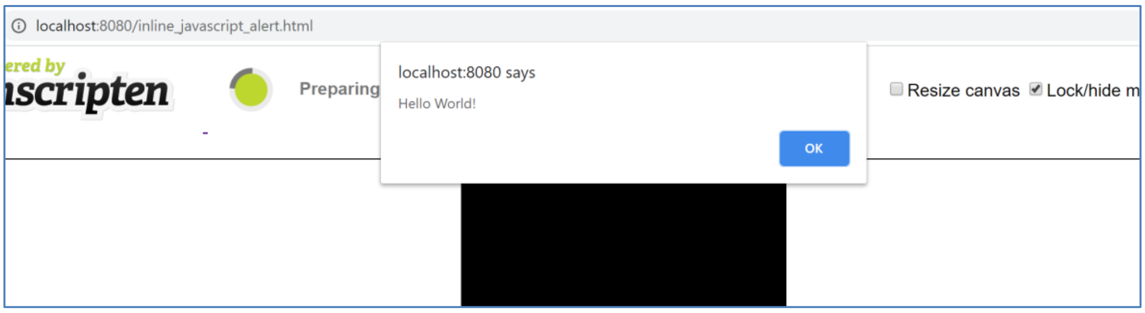
Then we use Emscripten to compiled the code into WebAssembly.

$ emcc inline_javascript_alert.c -o inline_javascript_alert.html

As we had discussed in the last section, we will need to serve this HTML over a server in order to test the result. Emscripten comes with *“emrun”* tool which helps us quickly create a local server.

$ emrun --no-browser --port 8080 .

Finally we visit the address *http://localhost:8080/inline_javascript_alert.html* on the browser. We should see an alert pops up with ”Hello World!” message as showed in figure 42.

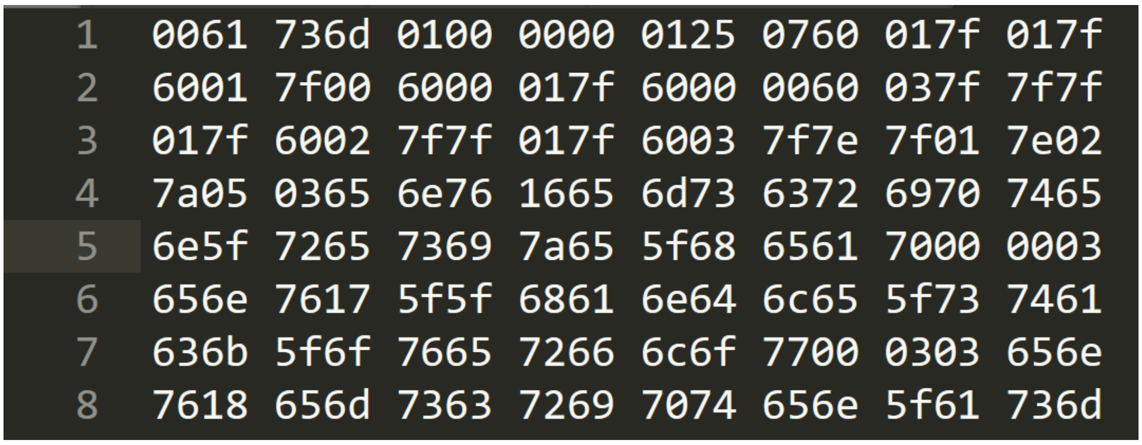

localhost:8080/inline_javascript_alert.html

localhost:8080 says

Hello World!

Figure 42. An alert with "Hello World!" message.

The ability of using inline JavaScript with Emscripten library is quite dangerous for the users because the author can run unwanted activities such as stealing the cookies, capturing the keystroke without the user consent. Since the WebAssembly module that is loaded to the webpage is binary that means we can only see bytecodes when we open the WASM file. Figure 43 shows the first 8 lines of the compiled WebAssembly binary inline_javascript_alert.wasm. Without converting the file into WebAssembly text format (WAT), we hardly know what the code does that allows malware author avoid detection from other security products on Web. An example to abuse this feature is that we can create a keylogger which log the keystrokes of the user and send them back to a remote server. We will discuss about this topic in chapter 4.3.



```
0061 736d 0100 0000 0125 0760 017f 017f
6001 7f00 6000 017f 6000 0060 037f 7f7f
017f 6002 7f7f 017f 6003 7f7e 7f01 7e02
7a05 0365 6e76 1665 6d73 6372 6970 7465
6e5f 7265 7369 7a65 5f68 6561 7000 0003
656e 7617 5f5f 6861 6e64 6c65 5f73 7461
636b 5f6f 7665 7266 6c6f 7700 0303 656e
7618 656d 7363 7269 7074 656e 5f61 736d
```

Figure 43. First 8 lines of inline_javascript_alert.wasm.

4.2 WebAssembly cryptomining

4.2.1 Overview of cryptomining

In the last few years, “cryptocurrency” has become more and more popular term all over the world. As stated by Wikipedia, “A cryptocurrency (or crypto currency) is a digital asset that is used as a medium of exchange wherein individual digital token coin ownership records are stored in a digital ledger or computerized database using strong cryptography to secure financial transaction record entries, to control the creation of additional digital token coin records, and to verify the transfer of token coin ownership” [29]. Unlike paper money which has physical form, cryptocurrency exists in a block of code which is generated by solving a complicated mathematical problem.

Cryptomining is a process of validating and updating the transactions into the database and in exchange, miners will be awarded an amount of cryptocurrency. To perform the mining task, we would need a lot of computing power from the hardware and a cryptomining software. Cryptomining software can be either a computer software or a web application. Traditionally, miners will use computer software (on Mac, Linux, Windows) because it can utilize most computing power from the hardware.

The problem of using computer software is that miners have to invest large amount of money to setup the machine and pay the electricity bills. To solve this problem, web cryptomining was introduced in 2011. In short, web cryptomining is implementing JavaScript to a website or a page on the website to perform the cryptomining through the browser using the local computing resources from the visitors. The advantage of web cryptomining is that miners do not need high cost hardware or knowledge about cryptomining. Beginner miners can start mining coins with just the internet connection and a legitimate cryptocurrency wallet.

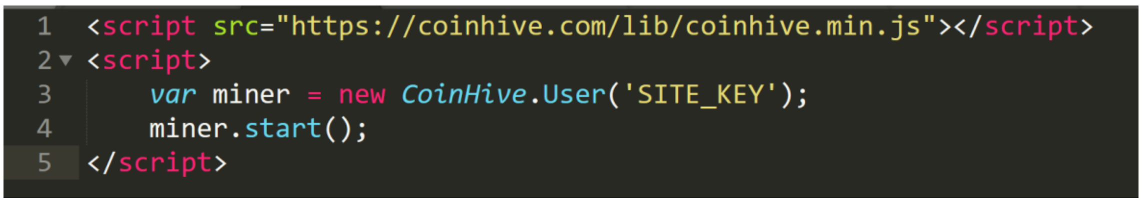
The dangerous thing about web cryptomining is that it can be performed without user's consent. This type of activity is called “Cryptojacking”. Attackers do cryptojacking in 2 ways: trick the user to click on a malicious link that loads cryptomining script on the browser or hack a website and inject cryptomining code to it. The cryptomining script should work in the background and it is difficult for the user to notice until their machine slows down or lags in execution. [29]

In 2017, web cryptomining had risen again because of the new programming language – WebAssembly. WebAssembly bring more performance to web application due to its capability to utilize computer hardware such as the processor and memory which JavaSript cannot do. The most popular web cryptomining that utilize WebAssembly is Coinhive cryptomining.

4.2.2 Coinhive cryptomining

Coinhive is a web cryptomining service which was designed to mine Monero cryptocurrency. Although Coinhive was annouced to be shutdown on March of 2019, it is still a need to make a research about how Coinhive implement WebAssembly into their application.

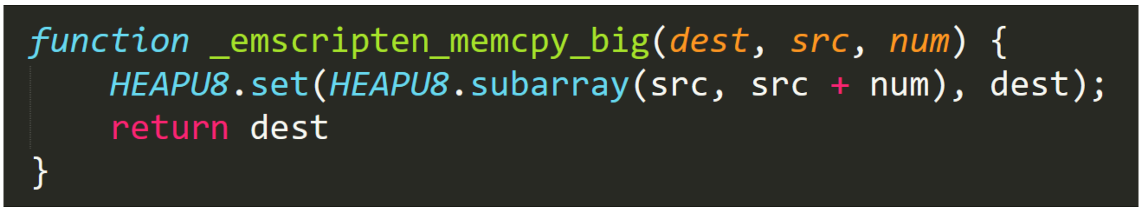
Coinhive cryptomining was made so that it is easy to enable the mining on a website. The user just need to register for a site key and put a few lines of code into the website as showed in figure 44. [31]



```
<script src="https://coinhive.com/lib/coinhive.min.js"></script>
<script>
    var miner = new CoinHive.User('SITE_KEY');
    miner.start();
</script>
```

Figure 44. Block of code to execute Coinhive cryptomining.

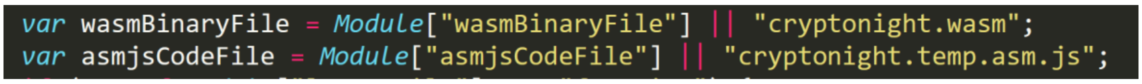
Thanks to *x25* on GitHub, we have a proof of concept for Coinhive cryptomining in *"Coinhive stratum mining proxy"* project. The core of Coinhive is in the file *miner.min.js* [32] of the project, we will use this file in further analysis. There are several evidences prove that Coinhive uses Emscripten compiler in their JavaScript program but the strongest evidence is the appearance of ***_emscripten_memcpy_big*** function (figure 45).



```
function _emscripten_memcpy_big(dest, src, num) {
    HEAPU8.set(HEAPU8.subarray(src, src + num), dest);
    return dest
}
```

Figure 45. The evidence of Emscripten compiler in Coinhive source code.

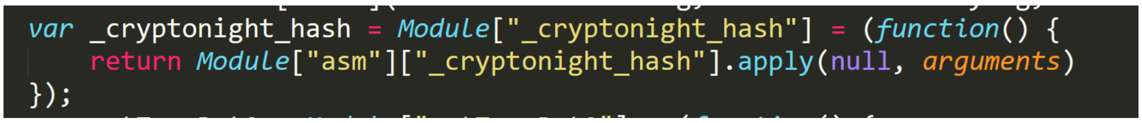
There are 2 filenames that we need to take notice in the code: *cryptonight.wasm* and *cryptonight.temp.asm.js*. Cryptonight is a hashing algorithm used in cryptocurrency mining which was first introduced in 2012 by Bytecoin. Coinhive uses Emscripten compiler to compile Cryptonight algorithm into WebAssembly binary and ASM.JS in order to run the algorithm on browser with near native speed. The JavaScript code will check whether the current browser supports WebAssembly or not, if the browser does not have WebAssembly object it will use ASM.JS file instead.



```
var wasmBinaryFile = Module["wasmBinaryFile"] || "cryptonight.wasm";
var asmjsCodeFile = Module["asmjsCodeFile"] || "cryptonight.temp.asm.js";
```

Figure 46. 2 filenames in the source code.

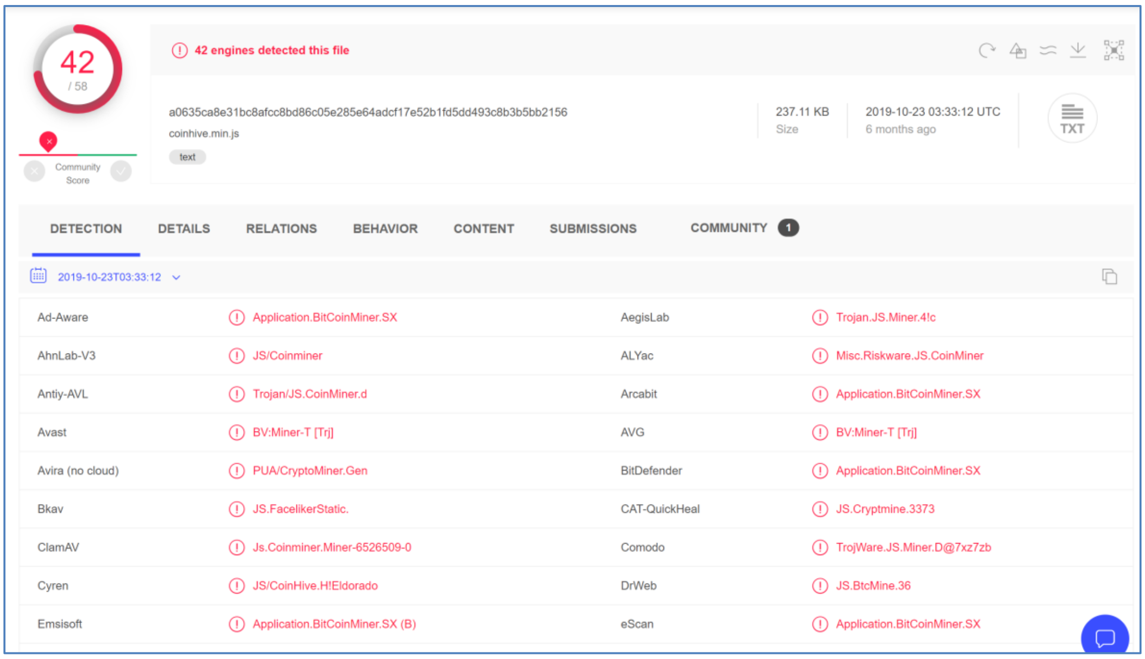
Assuming that the browser supports WebAssembly and the WebAssembly binary is fetched to the website successfully, Coinhive will call *_cryptonight_create* and *_cryptonight_hash* functions from WASM's exported functions to start running the hashing algorithm.



```
var _cryptonight_hash = Module["_cryptonight_hash"] = (function() {
    return Module["asm"]["_cryptonight_hash"].apply(null, arguments)
});
```

Figure 47. Coinhive call _cryptonight_hash function.

The original JavaScript file from the link https://coinhive[.]com/lib/coinhive.min.js is now detected by several security products in VirusTotal. Fortunately the server of Coinhive is now dead therefore the user is temporary safe from Coinhive cryptomining. That being said, it does not mean cryptomining in general is dead, the user always has to be careful when visit a new website and the web developer need to enhance their security in order to prevent any code injection from attackers.



42 engines detected this file

a0635ca8e31bc8afcc8bd86c05e285e64adcf17e52b1fd5dd493c8b3b5bb2156
coinhive.min.js
text

237.11 KB
Size

2019-10-23 03:33:12 UTC
6 months ago

2019-10-23T03:33:12

Ad-Aware	Application.BitCoinMiner.SX	AegisLab	Trojan.JS.Miner.4!c
AhnLab-V3	JS/Coinminer	ALYac	Misc.Riskware.JS.CoinMiner
Antiy-AVL	Trojan/JS.CoinMiner.d	Arcabit	Application.BitCoinMiner.SX
Avast	BV:Miner-T [Trj]	AVG	BV:Miner-T [Trj]
Avira (no cloud)	PUA/CryptoMiner.Gen	BitDefender	Application.BitCoinMiner.SX
Bkav	JS.FacelikerStatic.	CAT-QuickHeal	JS.Cryptmine.3373
ClamAV	Js.Coinminer.Miner-6526509-0	Comodo	TrojWare.JS.Miner.D@7xz7zb
Cyren	JS/CoinHive.H!Eldorado	DrWeb	JS.BtcMine.36
Emsisoft	Application.BitCoinMiner.SX (B)	eScan	Application.BitCoinMiner.SX

Figure 48. VirusTotal result for coinhive.min.js file.

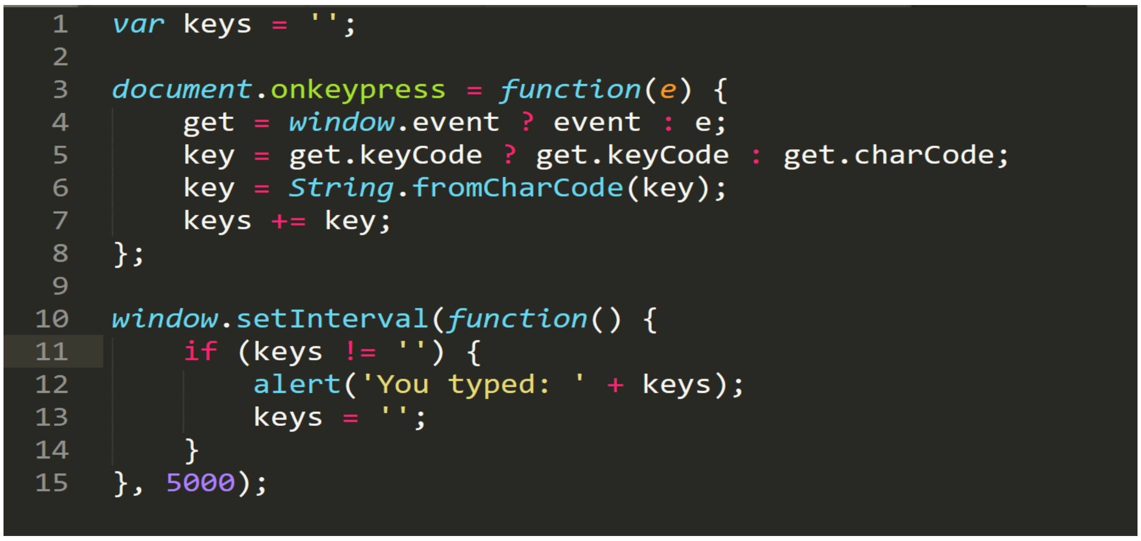
4.3 WebAssembly keylogger

4.3.1 Overview of keylogger

A keylogger can be either software or hardware that monitors and records (logs) the keystrokes on the keyboard without user’s awareness. A keylogger is often used by attackers to steal password or confidential information (such as bank account, credit card information, medical data,…). In this section, we will discuss about software-based keylogger and how it can be extremely dangerous when combining with WebAssembly.

A malicious software-based keylogger is usually a computer program which was dropped by attackers either via email or website. A software-based keylogger can be classified into separate categories such as hypervisor-based keylogger, kernel-based keylogger, API-based keylogger, JavaScript-based keylogger,… [33]

JavaScript-based keylogger is a malicious script which is written in JavaScript and it is injected into a website (either the malware author’s website or a vulnerable website) to record the keystroke from the user input. This process can be achieved by listening to the key events in JavaScript such as *onkeydown* event or *onkeypress* event. Figure 28 displays a simple demo of a JavaScript-based keylogger. It listens to *onkeypress* event and get the key’s character by converting the keycode into the charcode. For example, when the user presses key “a”, the *onkeypress* event will return keycode == 97 (ASCII value of “a” character) and then it will be converted to the “a” character with *fromCharCode()* function. Moreover, the code also pops up an alert every 5 seconds to display which key the user pressed on the website in the last 5 seconds. The keylogger’s author may obfuscate the code before they inject it to any website in order to avoid the detection from other security products as well as the web administrator.



```
var keys = '';

document.onkeypress = function(e) {
    get = window.event ? event : e;
    key = get.keyCode ? get.keyCode : get.charCode;
    key = String.fromCharCode(key);
    keys += key;
};

window.setInterval(function() {
    if (keys != '') {
        alert('You typed: ' + keys);
        keys = '';
    }
}, 5000);
```

Figure 49. An example of JavaScript-based keylogger [34].

4.3.2 Create a keylogger in WebAssembly

When WebAssembly was introduced along with Emscripten toolchain, JavaScript-based keylogger had reached a new stage. The JavaScript-based keylogger is now more difficult to be detected by security products or web administrator because it does not use a plain-text JavaScript but uses WebAssembly binary instead. In this demo, we will utilize Emscripten library which allows us to use inline JavaScript from C/C++ (as we discussed in part 4.1.3).

First we will create a new folder for our project name “keylogger” and go inside the folder.

```
$ mkdir keylogger && cd keylogger
```

Next we will create a C file to store the source code of the keylogger.

```
$ touch keylogger_javascript.c
```

With any text editor, we will put the source code into *keylogger_javascript.c* (Appendix 1). Here we will use the same code from the last section for the demo.

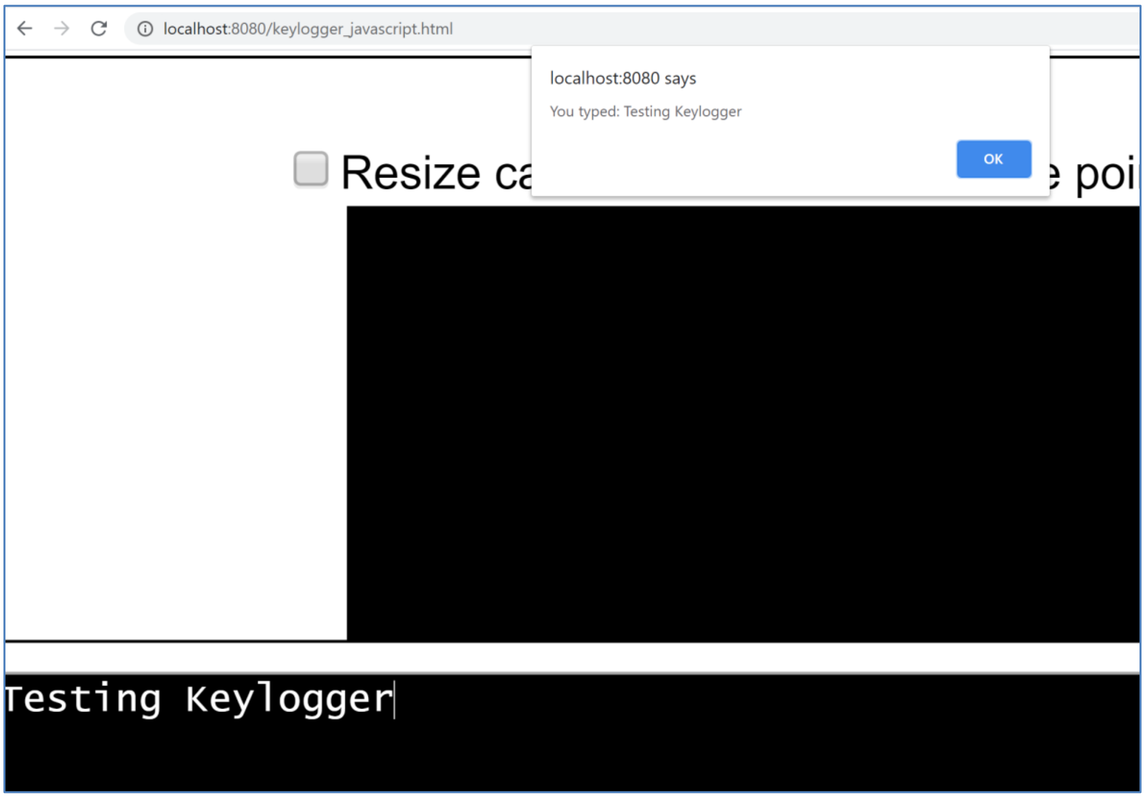
After that we will compile this C code into WebAssembly binary with the help of Emscripten compiler.

```
$ emcc keylogger_javascript.c -o keylogger_javascript.html
```

Now we should have 1 WebAssembly binary, 1 JavaScript file that import the WASM binary and 1 HTML file which we will use to test our keylogger. Once again, we have to setup a local webserver to serve the HTML file. Emscripten toolchain comes with a convenient tool *emrun* which helps us to quickly set the server up.

```
$ emrun --no-browser --port 8080 .
```

Finally we visit the address http://localhost:8080/keylogger_javascript.html to check the result. Figure 50 displays the final result of the keylogger when we input "Testing Keylogger" string into the black textbox, the page pops up an alert with the message "You typed: Testing Keylogger". It proves that the keylogger works as we expected.


localhost:8080/keylogger_javascript.html

localhost:8080 says

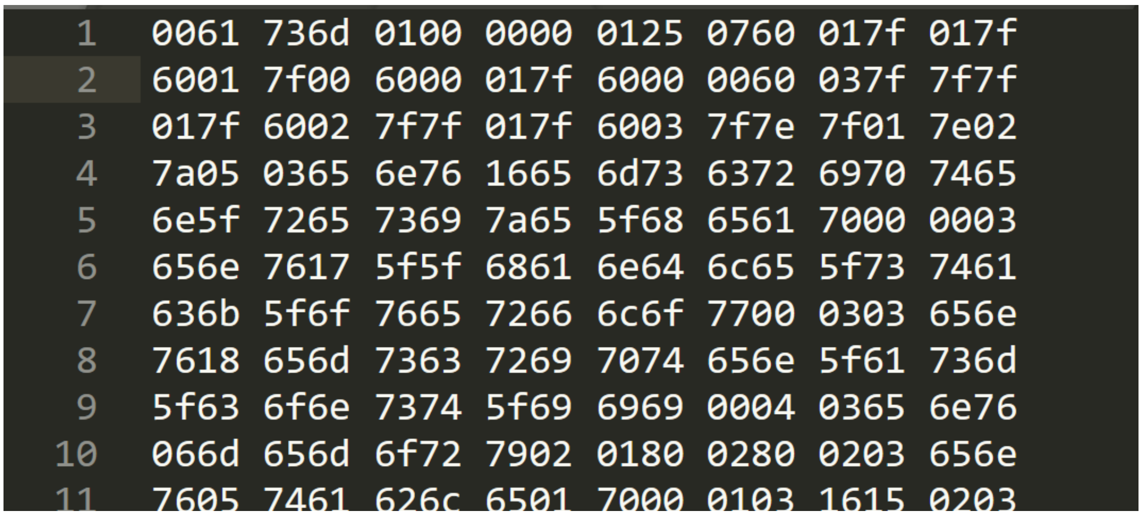
You typed: Testing Keylogger

```
Testing Keylogger
```

Figure 50. keylogger_javascript.html on the browser.

4.3.2 Infection vectors and mitigation

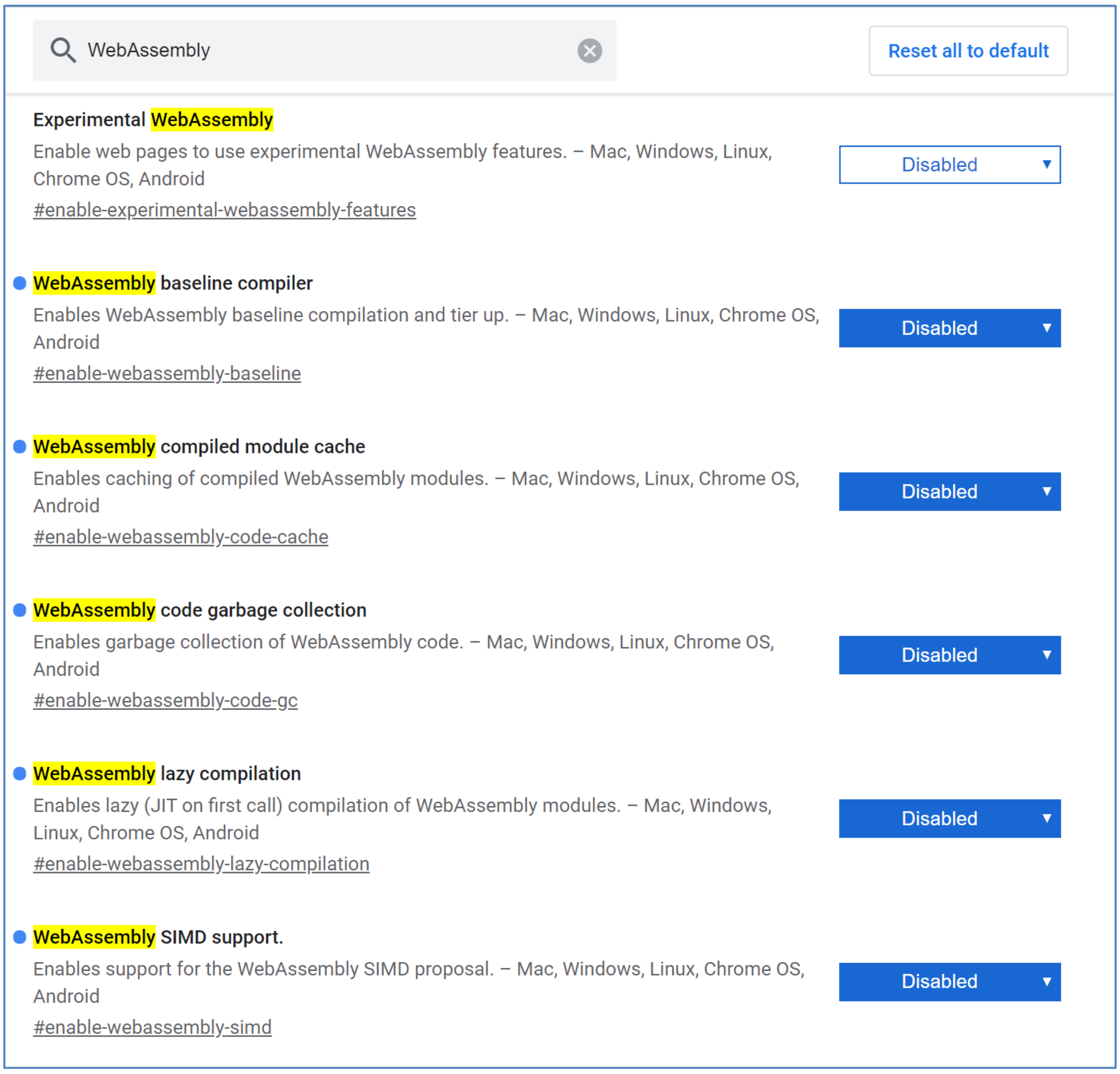
JavaScript-based keylogger is usually distributed via phishing email or a compromised website. Unfortunately the method of using inline JavaScript in C/C++ is not the only way to perform stealthy keylogging. Attackers can use a native C/C++ keylogger and compiled them into WebAssembly which makes the keylogger even harder to be detected by security products. On the browser, we can only see the binary format of WebAssembly (as showed in figure 51) and without converting it to WebAssembly text format, we hardly understand what does the code do or even when have a WebAssembly text format, it is still difficult to understand the source code.



```
0061 736d 0100 0000 0125 0760 017f 017f
6001 7f00 6000 017f 6000 0060 037f 7f7f
017f 6002 7f7f 017f 6003 7f7e 7f01 7e02
7a05 0365 6e76 1665 6d73 6372 6970 7465
6e5f 7265 7369 7a65 5f68 6561 7000 0003
656e 7617 5f5f 6861 6e64 6c65 5f73 7461
636b 5f6f 7665 7266 6c6f 7700 0303 656e
7618 656d 7363 7269 7074 656e 5f61 736d
5f63 6f6e 7374 5f69 6969 0004 0365 6e76
066d 656d 6f72 7902 0180 0280 0203 656e
7605 7461 626c 6501 7000 0103 1615 0203
```

Figure 51. The first 10 line of keylogger_javascript.wasm in hex format.

To mitigate this issue, the user should use a trusted security product if it is possible or disable WebAssembly on the browser. With Chrome browser, we can go to chrome://flags/#enable-webassembly and search for ”WebAssembly” keyword then set every entries in the result as ”Disabled” as showed in figure 52. [35]



WebAssembly

Reset all to default

Experimental WebAssembly
Enable web pages to use experimental WebAssembly features. – Mac, Windows, Linux, Chrome OS, Android
#enable-experimental-webassembly-features
Disabled

WebAssembly baseline compiler
Enables WebAssembly baseline compilation and tier up. – Mac, Windows, Linux, Chrome OS, Android
#enable-webassembly-baseline
Disabled

WebAssembly compiled module cache
Enables caching of compiled WebAssembly modules. – Mac, Windows, Linux, Chrome OS, Android
#enable-webassembly-code-cache
Disabled

WebAssembly code garbage collection
Enables garbage collection of WebAssembly code. – Mac, Windows, Linux, Chrome OS, Android
#enable-webassembly-code-gc
Disabled

WebAssembly lazy compilation
Enables lazy (JIT on first call) compilation of WebAssembly modules. – Mac, Windows, Linux, Chrome OS, Android
#enable-webassembly-lazy-compilation
Disabled

WebAssembly SIMD support.
Enables support for the WebAssembly SIMD proposal. – Mac, Windows, Linux, Chrome OS, Android
#enable-webassembly-simd
Disabled

Figure 52. Disable all WebAssembly feature in Chrome.

On the other hand, web developer need to enhance the security of the website by doing pentesting for any possible vulnerabilities such as XSS, SQL injection,… Moreover web administrator need to monitor the website carefully for any suspicious code that appears on the website. These practices will help to reduce the risks of WebAssembly being abused by attackers.

5 CONCLUSION

Firstly, this thesis aimed to analyze the portability and the performance of WebAssembly. Secondly, the thesis aimed to research how WebAssembly can be abused in the real world to perform malicious deeds. To analyze the portability and the performance of WebAssembly, we first research WebAssembly Ecosystem with Rust, then we used Rust and AssemblyScript to perform some tasks and compiled them to WebAssembly binary, then we compared the runtime of the same task which was written in JavaScript to have an overview of WebAssembly performance. The abuse of WebAssembly in the real world was researched by collecting, analyzing real cryptominer sample and making a simple keylogger which can be used directly in a website.

As a result in Chapter 3, we proved that with WebAssembly, web application can be run at near native speed due to its compact and minimal size, hence, the process of decoding and compiling the code is faster. The task that was compiled to WebAssembly runs faster than the same task that was written in JavaScript. Moreover, the process of compiling the code in Rust is easily achieved due to several existing toolkits. On the other hand, WebAssembly also raises concerns about security. As we had discussed in Chapter 4, hackers can abuse WebAssembly for malicious purposes, for example, carrying out cryptomining on the user browser or logging the keystroke of the user to collect sensitive data without any detection.

We have provided a very basic knowledge about WebAssembly in this thesis. We hope this thesis would help web developers and security researchers in developing the web environment more efficiently and safely.

REFERENCES

[1] World Wide Web Consortium. (n.d.). World Wide Web Consortium (W3C) brings a new language to the Web as WebAssembly becomes a W3C Recommendation. [online] Available at: https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en

[2] webassembly.org. (n.d.). WebAssembly. [online] Available at: https://webassembly.org/

[3] MDN Web Docs. (n.d.). Understanding WebAssembly text format. [online] Available at: https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format [Accessed 14 Jun. 2020].

[4] webassembly.github.io. (n.d.). Introduction — WebAssembly 1.1. [online] Available at: https://webassembly.github.io/spec/core/intro/introduction.html [Accessed 14 Jun. 2020].

[5] GitHub. (2020). WebAssembly/wabt. [online] Available at: https://github.com/WebAssembly/wabt [Accessed 14 Jun. 2020].

[6] LLVM. (n.d.). Clang C Language Family Frontend for LLVM. [online] Available at: https://clang.llvm.org/.

[7] MDN Web Docs. (n.d.). Using the WebAssembly JavaScript API. [online] Available at: https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API [Accessed 14 Jun. 2020].

[8] MDN Web Docs. (n.d.). WebAssembly.instantiate(). [online] Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/instantiate [Accessed 14 Jun. 2020].

[9] MDN Web Docs. (n.d.). TextDecoder. [online] Available at: https://developer.mozilla.org/en-US/docs/Web/API/TextDecoder [Accessed 14 Jun. 2020].

[10] GitHub. (2020). rust-lang/rustup. [online] Available at: https://github.com/rust-lang/rustup.

[11] GitHub. (2020). rust-lang/cargo. [online] Available at: https://github.com/rust-lang/cargo [Accessed 14 Jun. 2020].

[12] GitHub. (2020). rustwasm/wasm-pack. [online] Available at: https://github.com/rustwasm/wasm-pack [Accessed 14 Jun. 2020].

[13] GitHub. (2020). rustwasm/wasm-bindgen. [online] Available at: https://github.com/rustwasm/wasm-bindgen [Accessed 14 Jun. 2020].

[14] GitHub. (n.d.). rustwasm/wasm-bindgen. [online] Available at: https://github.com/rustwasm/wasm-bindgen/tree/master/crates/js-sys [Accessed 14 Jun. 2020].

[15] GitHub. (n.d.). rustwasm/wasm-bindgen. [online] Available at: https://github.com/rustwasm/wasm-bindgen/tree/master/crates/web-sys [Accessed 14 Jun. 2020].

[16] MDN Web Docs. (n.d.). MediaStream. [online] Available at: https://developer.mozilla.org/en-US/docs/Web/API/MediaStream [Accessed 14 Jun. 2020].

[17] docs.rs. (n.d.). wasm_bindgen_futures - Rust. [online] Available at: https://docs.rs/wasm-bindgen-futures/0.4.13/wasm_bindgen_futures/ [Accessed 14 Jun. 2020].

[18] MDN Web Docs. (n.d.). SharedArrayBuffer. [online] Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer [Accessed 14 Jun. 2020].

[19] GitHub. (2020). WebAssembly/proposals. [online] Available at: https://github.com/WebAssembly/proposals [Accessed 14 Jun. 2020].

[20] Aboullaite, M. (2017). Understanding JIT compiler (just-in-time compiler). [online] Aboullaite Med. Available at: https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/.

[21] Picado, J. (2017). Abstract syntax trees on Javascript. [online] Medium. Available at: https://medium.com/@jotadeveloper/abstract-syntax-trees-on-javascript-534e33361fc7 [Accessed 14 Jun. 2020].

[22] MDN Web Docs. (n.d.). Memory Management. [online] Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management [Accessed 14 Jun. 2020].

[23] doc.rust-lang.org. (n.d.). SIMD for faster computing - The Edition Guide. [online] Available at: https://doc.rust-lang.org/edition-guide/rust-2018/simd-for-faster-computing.html.

[24] emscripten.org. (n.d.). About Emscripten — Emscripten 1.39.17 documentation. [online] Available at: https://emscripten.org/docs/introducing_emscripten/about_emscripten.html [Accessed 14 Jun. 2020].

[25] GitHub. (2020). sql-js/sql.js. [online] Available at: https://github.com/sql-js/sql.js [Accessed 14 Jun. 2020].

[26] emscripten.org. (n.d.). Emscripten SDK (emsdk) — Emscripten 1.39.17 documentation. [online] Available at: https://emscripten.org/docs/tools_reference/emsdk.html#emsdk [Accessed 14 Jun. 2020].

[27] MDN Web Docs. (n.d.). WebAssembly. [online] Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly [Accessed 14 Jun. 2020].

[28] emscripten.org. (n.d.). Interacting with code — Emscripten 1.39.17 documentation. [online] Available at: https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html [Accessed 14 Jun. 2020].

[29] Wikipedia Contributors (2019). Cryptocurrency. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Cryptocurrency.

[30] GitHub. (n.d.). x25/coinhive-stratum-mining-proxy. [online] Available at: https://github.com/x25/coinhive-stratum-mining-proxy [Accessed 14 Jun. 2020].

[31] Segura, J. (2019). Cryptojacking in the post-Coinhive era. [online] Malwarebytes Labs. Available at: https://blog.malwarebytes.com/cybercrime/2019/05/cryptojacking-in-the-post-coinhive-era [Accessed 14 Jun. 2020].

[32] GitHub. (n.d.). x25/coinhive-stratum-mining-proxy. [online] Available at: https://github.com/x25/coinhive-stratum-mining-proxy/blob/master/static/miner/miner.min.js [Accessed 14 Jun. 2020].

[33] Wikipedia Contributors (2019). Keystroke logging. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Keystroke_logging.

[34] Stack Overflow. (n.d.). html - Javascript keylogger (for ethical purpose). [online] Available at: https://stackoverflow.com/questions/53139197/javascript-keylogger-for-ethical-purpose [Accessed 14 Jun. 2020].

[35] Hu, J. (2016). How To Enable WebAssembly In Chrome. [online] Next of Windows. Available at: https://www.nextofwindows.com/how-to-enable-webassembly-in-chrome [Accessed 14 Jun. 2020].

Appendix 1. keylogger_javascript.c

```
#include <emscripten.h>

int main() {
    EM_ASM
    (
        var keys = '';

        document.onkeypress = function(e) {
            get = window.event ? event : e;
            key = get.keyCode ? get.keyCode : get.charCode;
            key = String.fromCharCode(key);
            keys += key;
        };

        window.setInterval(function() {
            if (keys != '') {
                alert('You typed: ' + keys);
                keys = '';
            }
        }, 5000);
    );
    return 0;
```