OULU UNIVERSITY OF APPLIED SCIENCES

Vili-Petteri Niemelä

WEBASSEMLY, FOURTH LANGUAGE IN THE WEB

WEBASSEMBLY, FOURTH LANGUAGE IN THE WEB

Vili-Petteri Niemelä
Bachelor's Thesis
Autumn 2021
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Software Development

Author: Vili-Petteri Niemelä
Title of the bachelor's thesis: WebAssembly, Fourth Language in the Web
Supervisor: Eino Niemi
Term and year of completion: Autumn 2021 Number of pages: 28 + 2 appendices

JavaScript has been shown to have significant lacks in performance regarding heavier web applications. WebAssembly is a new compilation target for many programming languages that bring near native speeds to solve this problem. WebAssembly binaries can be used in many platforms. Web browsers were the main target in the beginning, but it has since found places in the server side as microservices and even in embedded devices.

For this thesis general information about the subject was gathered from articles and recordings of lectures in conventions. Any deeper study of the specification of WebAssembly was left outside the scope of this thesis and the main focus was on a general introduction to WebAssembly and its possibilities.

The applications and properties of WebAssembly were evaluated in the thesis and it seems to have real potential to change the way people make web, and other applications in the future. A survey into how people have adopted the new technology and what they see in its future was referenced.
This thesis also explains how to make a simple Wasm module with the Rust language that decodes QR code data from images.

Keywords: Programming Languages, Web Browser, Virtual Machine

PREFACE

This thesis was written during 2020 and 2021 as the Bachelor's thesis for Oulu University of Applied Sciences Information Technology degree. The idea for it came from general interest of new technologies in the web and in that space WebAssembly is a new feature that has the potential to change many parts of how the web is used. This thesis was made as a general introduction to WebAssembly and at what state it is in its development at this time.
Thanks to Eino Niemi, my supervising teacher for this thesis and Kaija Posio who proofread the thesis. English is not my native language, so it was an educational and great experience to be able to write this thesis in it.

Oulu, 30.11.2021
Vili-Petteri Niemelä

CONTENTS

VOCABULARY

DoRIoT – Dynamic runtime for organically (dis-)aggregating IoT-processes. A toolset for internet of things devices.

WAMR – WebAssembly Micro Runtime, Small interpreter for Wasm designed to be used in embedded environments.

WASI – WebAssembly System Interface

Wasm – WebAssembly

1 INTRODUCTION

This thesis was written as an introduction to WebAssembly, later referred as Wasm, a new language used in the Web. It was decided that giving a general introduction and overview of Wasm was more relevant than any specific implementation of Wasm, it being still in its early stages. Many of the tools for Wasm are also new and many more are just starting to be developed. Many have even matured quite a bit since this thesis was started.

First, the focus was on the history and development of Wasm and what previous technologies existed, which had tried to solve same, or similar, problems. Wasm differs fundamentally from them, so the focus was given to the design principles of Wasm. A deeper look into internal workings of Wasm was out of the scope of this thesis.

The focus was then shifted to applications of Wasm. In what environments it is currently in use and what benefits it has compared to other, current, and previous, technologies?

Then a simple hands-on example was made to demonstrate how one can compile code written with the Rust language into a Wasm module that can be used on a web page. After that the example application was expanded to have real world application by making a module that decodes QR code with it.

The future of Wasm was then speculated on. A look was taken of a survey on how people have been using Wasm and how they see it in the future. Finally, the changes Wasm might bring to the Web and even outside of it were speculated on.

2 WEBASSEMBLY – WASM

WebAssembly, or Wasm as it is abbreviated, started when two teams: the PNaCl team from Google and the asm.js team from Mozilla, combined their efforts to run bytecode in the browser (1). PNaCl or Portable Native Client is a way to run bytecode on the browser, but it is mostly supported on the browsers of Google. Asm.js is a JavaScript optimization project to increase the speed of JavaScript code execution in the browser. The effort led to the first MVP of WebAssembly in 2017 and acceptance into the web standard in 2019. (2, 3). Now, the specification is maintained by the W3C (the World Wide Web Consortium), and many organizations contribute to it, including all the major browser developers: Mozilla, Google, Apple, and Microsoft. The Bytecode Alliance, a group focussed on developing the Wasm specification further, was formed from many of the groups that developed Wasm. It has also expanded to include other organizations, such as Fastly, Intel, and Red Hat.

Wasm was created to make use of computers processing power more efficiently and portably without giving up on the sandbox environment that browsers provide. Wasm is at its core a specification of a virtual instruction set architecture or ISA (4). It does not model any real computational unit but a virtual one. The communication between a Wasm application and underlying hardware is handled by some runtime that implements the instruction set for the specific hardware targeted. In browsers this is usually the v8-JavaScript engine. This allows same Wasm modules to be run in every platform that has implementation of a runtime which is Wasm specification compliant.

2.1 Comparison to other languages in the Web

Wasm is the only precompiled, the World Wide Web consortium (W3C) recommended, language that is used in the web environment. JavaScript is compiled and optimized just before or just-in-time (JIT) running it. HTML is a markup language, and CSS is mostly definitions of element styles and sometimes animations. The web environment has had a lack of high-power

language that can calculate heavy operations efficiently. such as 3D graphics and media stream conversions.

Many attempts of running binary code in the browser have been made over the years but none of them have been cross-browser and/or easily portable to other platforms. Microsoft had ActiveX and later Silverlight; Google has Google Native Client (NaCl). Adobe Flash from Adobe, which was cross-browser, was slow and full of security issues, reached its end-of-life December 31, 2020. Adobe Flash had the widest range of compatibility but required the download of flash plugin to work. In contrast, all the major browsers now support Wasm natively, if not the most resent version, then at least the 1.0 version. (5)

2.2 Compilation target for other languages

Unlike JavaScript, which is compiled and runs on the fly, in many cases even before it is completely downloaded, Wasm is precompiled. It is in a bytecode format and does not need compilation. It is not recommended or intended to be used as a programming language as itself but as a compilation target for a expanding list of other languages, including Rust, C and C++. The code can be compiled with tools, such as wasm-pack for programs made with Rust or Emscripten for C. It is also possible to program Wasm modules with specifically designed scripting languages that have been created to make it easier to code for Wasm without needing to learn more hardware-oriented languages, e.g. C. AssemblyScript is one of these, and by-far the most popular one. (6)

Being a compilation target for other languages Wasm opens many possibilities in porting libraries from other languages to it. One could take, for example, a well-established media manipulation tool ffmpeg and port it to Wasm and run it in the browser like Jerome Wu has done (7). Wasm modules are cached in the browser so only the initial download transfers data from the server and after that the user has all the capabilities of the Wasm module even if the device of the user loses the Internet connection. A new version of the module needs to be downloaded only in the case of an update.

There is a text form representation of Wasm modules called WAT or WebAssembly Text Format. As seen from the figure 1 below, it resembles Assembly and is the human readable form of Wasm. One could also make super compact Wasm applications by coding in WAT format directly and compiling it into Wasm bytecode with a wat2wasm tool from the WABT toolchain. Ben Smith showcased this in his presentation in the WebAssembly Summit 2021 by creating a match-three game during his presentation. (8)

```rust
#[no_mangle]
pub extern "C" fn add_one(x: i32) -> i32 {
    x + 1
}
```

```
(module
  (type $t0 (func (param i32) (result i32)))
  (func $add_one (export "add_one") (type $t0) (param $p0 i32) (result i32)
    get_local $p0
    i32.const 1
    i32.add)
  (table $T0 1 1 anyfunc)
  (memory $memory (export "memory") 17))

```

FIGURE 1. Same code written in Rust and in WAT format (Niemelä V-P, 2021)

2.3 Security

In addition to speed, security is one of the most prominent benefits Wasm can provide. By default, Wasm modules have no access beyond their own linear memory and they need to be explicitly given the permission to access resources outside it. This way of structuring runtime eliminates whole categories of security vulnerabilities. How can an application have vulnerabilities regarding filesystem if it cannot even access any part of the filesystem? This type of structuring a system by limiting the capabilities of it as default is called capability-based security.

Although Wasm modules live in predefined linear memory, and it is impossible to overflow out of, there are still risks. It is not as easy to discern what a Wasm module does on a web page since it is not in plain text and needs to be disassembled before it is even close to being human readable. Hiding obfuscated malicious code in these binaries, which get executed after JavaScript glue code calls the module, might be problematic. For example, many of the malicious Wasm modules found in the web are running crypto currency miners. These are stealing calculation power from the computer that is currently visiting the site. Luckily, these are not persistent threats to the visitor and are stopped from running simply by navigating away from the site.

3 APPLICATIONS OF WASM

3.1 Web applications

Wasm brings more capabilities to the web. While JavaScript could do the same tasks as any other programming language, its focus was not on a heavy-duty application, such as 3D graphics or optimized hash calculations. Libraries in other programming languages already exist which have been highly optimized for these tasks and with Wasm these could be ported to the web with a minimal effort compared to developing and optimizing same capabilities in JavaScript.

While cloud services have been able to bring higher performance applications to the web, they need the server-side CPU and GPU usage which costs money to the maintainer and these kinds of services are usually not free. Also, the latency they have in the communication between the user and the server is a problem. Wasm modules on the other hand are run on the machine of the users so there is no need for powerful servers for the developers and no latency in most actions in application for the user. The Wasm application can be downloaded by the user and their computer does the heavy lifting. This kind of a web application does not even need to use the Internet if the browser has a cached version of the Wasm modules it needs to work.

One example of a Wasm application is photo editing software called Photopea. (9) It is currently free to use and has many of the features that traditional image editors, e.g. Photoshop, have. It runs in the browser, and it feels as fast as any native photo-/image editing software, such as Gimp or Photoshop. Because it runs in the browser, one could also run it on a smartphone or, for a better experience, on a tablet. A screenshot of the UI is seen in the figure 2, on the next page, with an example project that has a couple of layers.

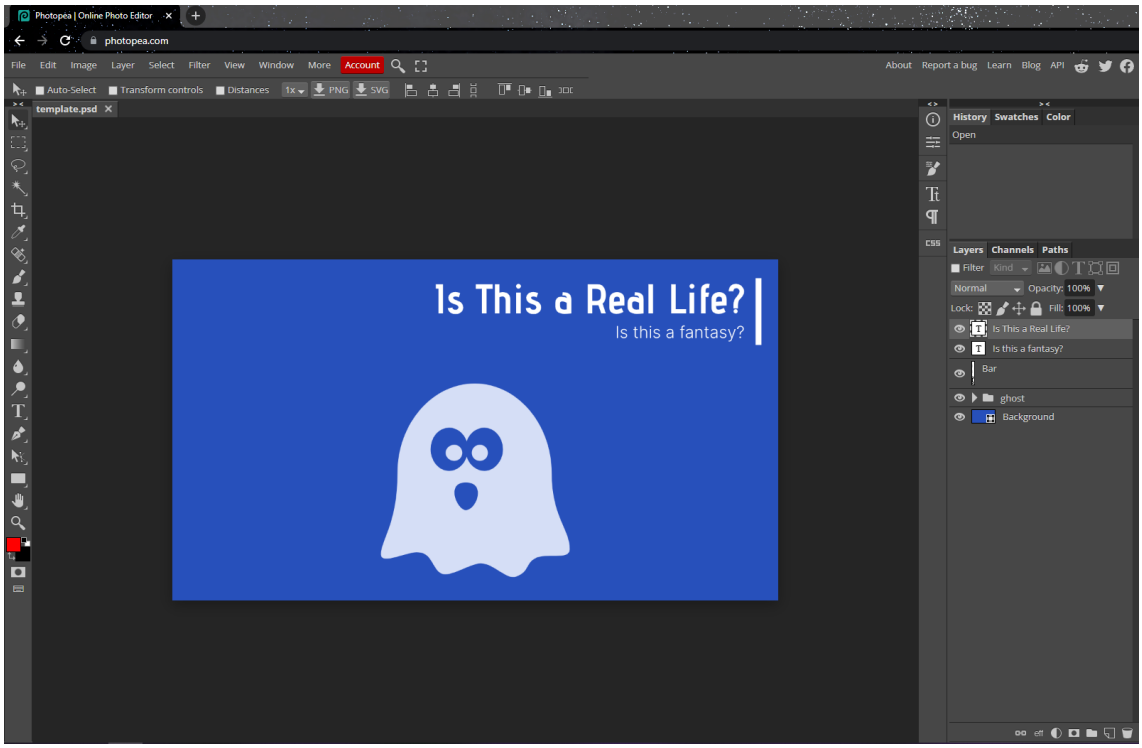

FIGURE 2. UI of Photopea in the browser (9)

3.2 WASI

While Wasm was first targeted for use in the sandbox of the web browser, it is designed in such a way that it can run on any platform if an implementation of Wasm runtime exists. In browsers Wasm runs in the JavaScript engine and in the sandbox of the browser but to run Wasm modules outside of them, an interface with the underlining system is needed. Here WASI (the WebAssembly System Interface) comes into play.

WASI refers to all the different interpreters of Wasm code outside the browser. The Bytecode Alliance has a couple of Wasm runtimes of their own: Wasmtime and WAMR (WebAssembly Micro Runtime). WASI promotes guidelines of how a proper Wasm interpreter should be made so it keeps the sandboxed safety features that it has within browsers by default. Wasmtime is a just-in-time type runtime for Wasm code and WAMR is a Wasm interpreter designed for embedded devices. WAMR has both JIT and AOT, or ahead of time, interpreters. (10)

3.3 The IoT – the Internet of Things

WAMR or WebAssembly Micro Runtime is a runtime developed to run Wasm in embedded systems. Running Wasm in embedded context might sound a little strange but having one module that works in any hardware, provided it runs WAMR or any other Wasm runtime, has huge benefits to software development. Compared to other virtualization technologies, Wasm has a much smaller footprint and overhead. That also lets the programmers to use a language they are used to or a language has the best properties for the project at hand. After compiling to Wasm, the module can be run in any Wasm runtime.

Lecture notes by Karl Fessel, André Dietrich and Sebastian Zug mapped the use of Wasm in the IoT environment. In the notes they consider the benefits of using a multi-language compatible compilation target VM as a platform for IoT devices. They mention that most constrained environments must be programmed with C, C++, Java, Python or JavaScript. C and C++ being the most common ones. Java is the only one before Wasm with platform independent modules that run a Java virtual machine inside the embedded device. Fessel et al used Wasm to make it possible to write code for embedded devices with any language that compiles into Wasm, by augmenting DoRIoT project's tooling stack with WAMR from the Bytecode Alliance. This, they said made it possible to choose the programming language most suitable for tasks in hand, and most importantly it would not limit the programmers when choosing tools. (11)

4 EXAMPLE APPLICATION

Next a sample application was made. It was made following a basic tutorial made by Mozilla on how to compile Rust code into a WebAssembly module and use it on a pure HTML page or on a page served with NodeJS. Rust was chosen as the language to use for this example program. This was because Rust was among the first languages with supported compilation to Wasm and Mozilla (current developers of Rust) which were in the group that started the specification process for Wasm. The example was made from two parts. The first part is based on the guide by Mozilla and is a Wasm version of a “hello world” project. The second one was made for this thesis. The functionality of the app was expanded in the second part with Rust crates rqrr and image. The libraries in the Rust language are called crates. The resulting Wasm module was a basic QR code reader application. (12)

4.1 Tools

As Rust was chosen, the basic Rust SDK, or the Software Development Kit, was needed. A browser, Mozilla Firefox was chosen, because it came with Ubuntu, to run the example and the crate of Wasm tools called wasm-pack for Rust. These were installed on a clean install of Ubuntu 20.04 virtual machine.

4.2 Setup

First it was made sure that build tools were installed in the system so that the installation of the Rust language development kit could be continued with. Then the Rust SDK build tools were installed via an install script called rustup found in the home page of Rust. A basic installation was chosen from the installation options.

The next step was to install wasm-pack, the Wasm toolchain of Rust, with the library management tool of Rust called Cargo. With wasm-pack the Rust Wasm

projects could now be initialized, and Rust source files could be compiled into Wasm bytecode files and JavaScript clue-code. It has different build targets for different target environments depending on if the user wants to target pure browser compatibility or if the user going to serve the module with NodeJS.

4.3 First Part – Hello World

The project was initialized by invoking cargo with the option “new” and with a name for the project. This generates the project directory with initial files. Then the Rust source file was modified to include the wasm_bindgen library. The library comes with wasm-pack and is needed to expose functions for JavaScript to use. JavaScript functions could also be imported to the Rust code with the extern keyword.

The default project was modified to use JavaScript´s alert function in its own public wasm_bindgen exposed function that takes a string as a parameter and adds it to a “hello” message before using the newly concatenated string in an alert call. After modifying the Rust file, the project’s Cargo.toml file, which defines the package for compilation, needed to be configured. This includes e.g. a list of dependencies, version number.

Now the project was ready to be built into a Wasm file and associated glue code. A Wasm-pack tool was used to compile the project and a web was chosen as the target. This created an independent Wasm file and the clue code JavaScript that could now be added as a module into any web page. The module was imported with script tags with a module as the type. The “import” keyword was used to bring the Wasm module’s glue code and chosen functions into the browser’s scope. The automatically generated clue code function called “init”, which was used to instantiate the Wasm file, and the “hello” function were imported.

To use the imported Wasm function, which was made in Rust, the initialization function had to be called. Also, an asynchronous “then” handler had to be chained to trigger after the Wasm module had been instantiated. After the

Wasm module had been instantiated, the functions defined in the module were ready to be called.

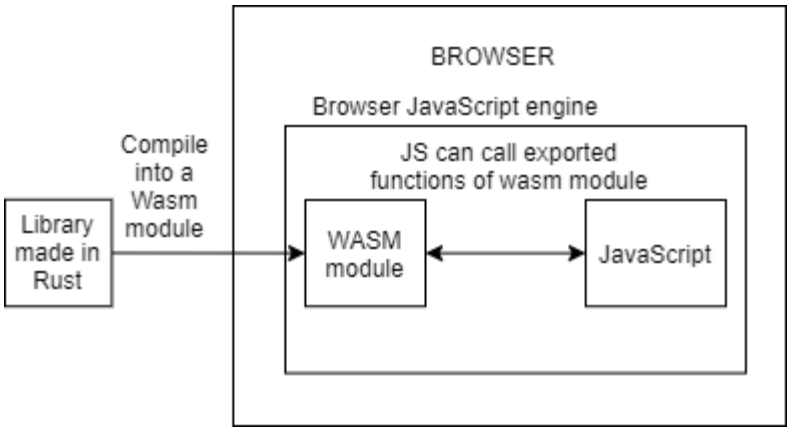

FIGURE 3. Simplistic example of how Wasm is used in the Web (Niemelä V-P, 2021)

The figure 3 above visualizes the relations with a browser, Rust and JavaScript in a basic form. The compilation of Rust code into Wasm and JavaScript code is done outside the browser and after that the browser calls Wasm module's APIs with JavaScript.

4.4 Second Part – QR reader

The aim of the second part was to make a Wasm module that would take in image data and return the possible QR code data contained within. To handle image data Rust crate called Image (13) was used and to read QR data from image objects a crate called rqrr (14) was used. These were added to the toml file of the project under dependencies list.

At first the sample program was made with an internal buffer for the pixel data of the image, a function to return a pointer to the beginning of the buffer and a function to scan the image data for a QR code. The QR code, if found, was returned with a call to the alert-function of JavaScript that was imported into the context of Rust with the wasm_bindgen library. (15)

This design had many problems. One was that the image size was an invariable number of pixels. So, all images had to be scaled down, up, or cropped to fit in the array. Another was that Rust does not support global buffers and the calls that accessed it had to be wrapped within the “unsafe” block of Rust. This is by design because in applications that use multiprosessing, this might cause a collision in data access. The final problem was that the module did not return the data found from the QR code but passed it instead to a JavaScript function that showed it in a pop-up in the browser. This sort of data return type has very limited usability in practice.

Luckily, wasm_bindgen had many compatibility wrappers that support more datatypes than Wasm does. Wasm only has four basic datatypes: 32-bit integer, 64-bit integer, 32-bit float, and 64bit integer. Wasm-pack has automatic clue code generation for conversion between many of the types from Rust to JavaScript. This was taken advance of, and the code was refactored. Instead of the global buffer, the QR code check function was changed to take an unsigned 8-bit integer array pointer, width, and height as parameters and give a string as the return value. This way the module was no longer limited to images of one size and the buffer was no longer needed. Also, a function that has a return value is more useful in the real world than one that calls a set function like alert. (Appendix 1)

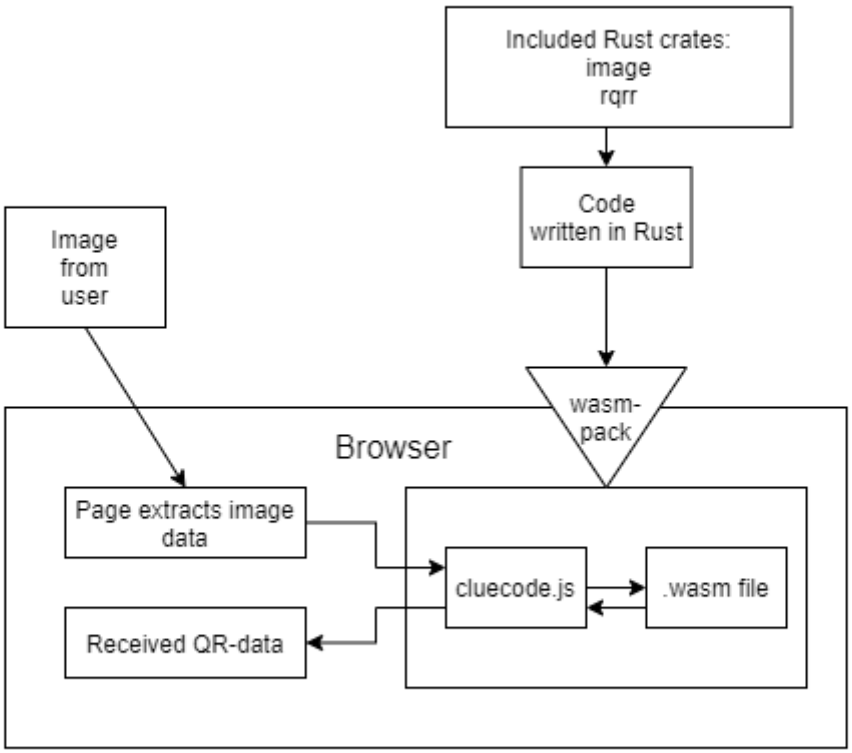

FIGURE 4. Description of Wasm QR-code reader module relations (Niemelä V-P, 2021)

The figure 4 above visualizes roughly the relation between the Rust libraries, the browser, and user input data, in this example QR code reading application. The Rust libraries are included to this project that is compiled into clue code and a Wasm file. After this the browser, the user, and the application handle the application.

The Wasm module was used on a test web page where one could upload an image and JavaScript would extract the byte array from it and use it, width, and height of the image to call the function of the initialized Wasm module that extracts data from the QR code the image might contain. The result is returned as a string to JavaScript. Then the result was printed to an alert box if a non-empty string was received. (Appendix 2)

The speed of the module compared to a one written in JavaScript was then tested. After running the Wasm read function for one image in a loop for 1,000 times, the average time for one iteration was around 125ms and the same test for a JavaScript QR code decoder library called jsqrcode (16) was around 30ms. So, in this case the Wasm module was a lot slower. This was mostly due the non-existent optimization of the Rust code and probably too complicated way to call the rqrr library. This could be optimized further, and at least comparable or faster speeds could be achieved with a smaller module size.

5 FUTURE OF WASM

5.1 State of WebAssembly -survey

In June of 2021, the creator of WebAssembly Weekly newsletter Colin Eberhardt made the first State of WebAssembly survey. These kinds of surveys have been made for other languages for years, but this was the first for Wasm. In the survey Eberhardt surveyed which programming languages people were using to make Wasm modules, what were they using Wasm for, and what features programmers were looking for in the future of Wasm. The survey was answered by 250 people. (17)

In the survey the most used language was Rust followed by C++. This makes sense since these were the first to compile into Wasm. The third most used language was AssemblyScript, which is a TypeScript like language that was designed to be compiled into Wasm. This trend continued when the most wanted language support was asked. The most wanted was Rust, but the second and third places were swapped between C++ and AssemblyScript. (17)

Next the survey asked what people use Wasm for, and on what area they imagine it will have the most impact. The survey found out that people were mostly using Wasm for web development. The second and third place use cases were gaming and serverless applications. This mirrors also in the areas people were predicting to be impacted. The first was web development and the second and third places were swapped, like in the previous question pair. The fourth in both of these pairs was containerisation. Eberhardt also quotes a tweet by the co-founder of Docker, Solomon Hykes, in which Hykes implies that if Wasm and WASI existed in 2008, they would not have had to develop Docker. This is an example of the potential of Wasm in the server-side of web. (17)

The next pair of questions was about the future features of Wasm and what features people wish that Wasm had. The most expected features were threads, WASI, and Interface Types. The most wished features were debugging, better APIs, and tooling. Lastly the survey mapped the demographic of the surveyed people and how long they had been aware of Wasm. In the

mapping people were asked about their familiarity with JavaScript, back-end programming, and Wasm. Most had at least medium familiarity with all the categories. As for familiarity with Wasm, it was mostly new to them; the most common answer was under one year. (17)

5.2 Wide-spread use in web

The whole reason Wasm was initially developed was to have more effective computation on the Web. Many applications in the web are going to implement Wasm modules to make their services more fluent. Especially the heavier functions of JavaScript that could be run more efficiently with Wasm. Also, the calculations that have been too heavy for JavaScript and traditionally ran in the server-side, could be brought to the browser of the user, thus reducing latency of the application, and saving bandwidth. This applies also to the server side where functions as a service type instance of applications, or serverless as they are called, are being replaced with ones made with Wasm. Container type of instances could be replaced with orders of magnitudes lighter versions of applications made with Wasm, enabling to run more services with the same number of resources and leading to an improved efficiency and a lower energy usage.

Whole new types of applications, which were previously impossible to run in the web browser, are to be seen, and the current running applications are going to be more and more fluent.

5.3 Interface Types

While Wasm modules can be programmed in many languages and compiled into Wasm bytecode that can be understood by the runtime it lives in, communication between modules written in different languages is not as simple. This is where a future feature proposal of interface types come in. A standardized layer of code on top of Wasm core spec that handles the type

conversions of different types of data Wasm modules could communicate with. (18)

5.4 Shift in third-party module safety

JavaScript third-party modules have a history of security concerns. A common vulnerability of JavaScript libraries has been that every library has the same access rights as its parents or grandparents. This gives a filesystem or a secrets storage access to libraries that have nothing to do with either of them. This is illustrated in the figure 5 by Lin Clark.

Even if a library in JavaScript is safe now it may become hostile later when for example the maintainer changes or a malicious commit manages to sneak into the repository. Even benign code, which has bugs in it can be turned into an access point for malicious deeds because of this inherited rights design.

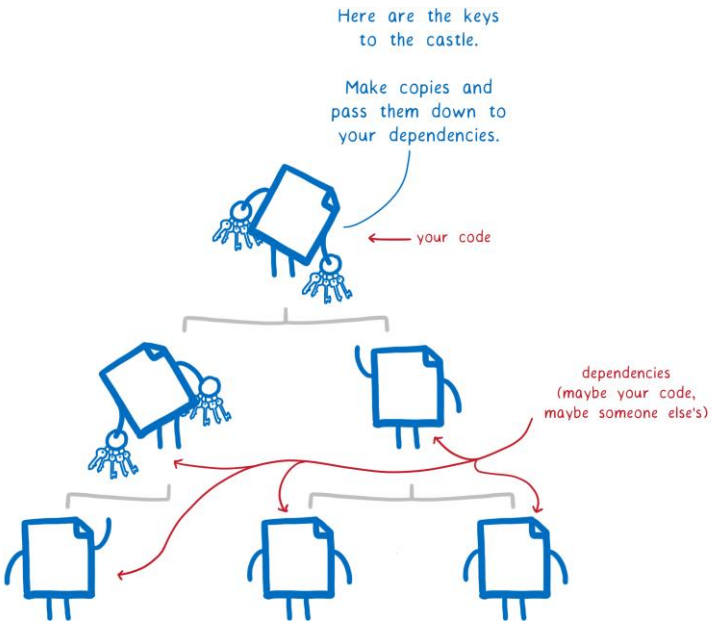

FIGURE 5. Traditional inheritance of right of modules, by Lin Clark (19)

Long term aim of Wasm has been suggested to change the paradigm of programming for modules towards a safer direction. Modules are to communicate via specified interfaces that allow only specified communications between modules to prevent a malicious behaviour. A huge benefit of WebAssembly is that each module only has access to its own linear memory and needs implicit access to outside resources.

With WebAssembly it does not matter if a module tries to be malicious if the context where it is used does not allow it to access new resources. The modules are given only the rights they need and no more. This isolation, which has been given the term nanoprocess, is illustrated below in figure 6. (19)

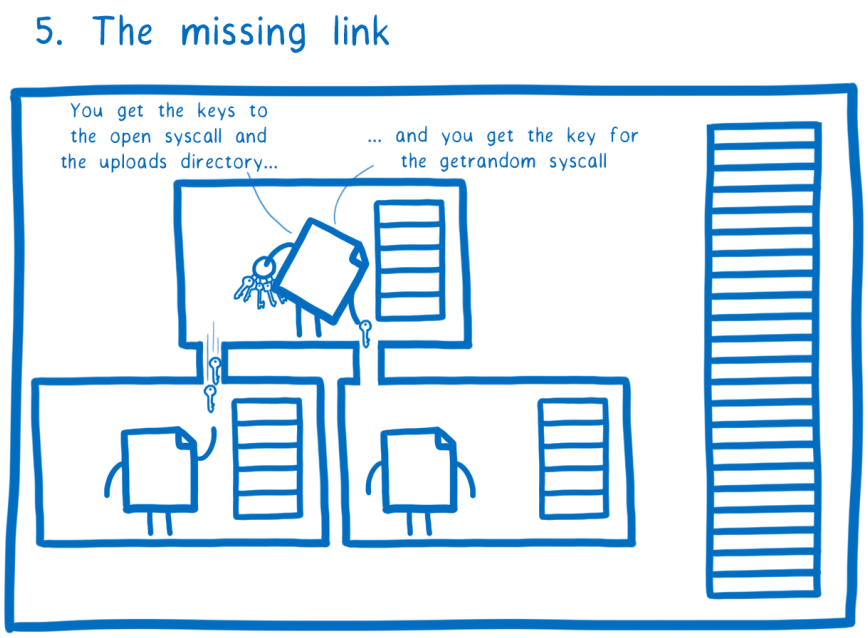

FIGURE 6, Nanoprocess model of rights, by Lin Clark (19)

6 CONCLUSION

Wasm seems to have a great potential to be the driving engine of the future Web applications and perhaps even as a backbone of computer services in general through WASI implementations. After the Wasm tools have matured, users or even developers probably will not need to know much about Wasm as it is going to be behind the scenes as a platform, compilation target or in other framework roles.

Wasm, its tools, and supported languages have developed a lot during the making off this thesis and the speed of development does not seem to be slowing down. Many new tools have emerged, and some have either been deprecated or fused into other projects. Based on the survey, programmers who use Wasm seem to have strong ideas where it is headed and what they would like to see in the future from the language.

Even if Wasm was originally targeted as a speedup for the web environment, it seems to have found its place also outside of the web browser with implementations, such as Wasm runtimes like Wasmtime or WAMR. Though many of the implementations are still juvenile, they have made big changes to services and have even bigger possibilities in the future.

The aim of this thesis was to investigate Wasm as a new and upcoming technology. This was succeeded in many ways, but only as a shallow examination. Currently Wasm is trying to find foothold in many different areas of computing and there is so much experimentation that it is impossible to cover them all in a single text. The first part of the sample program was easy to implement due to the excellent guide by Mozilla. The second part had some trouble, caused by the lack of experience in Rust by the author and the lack of solidified best practices of implementing shared memory, for example, between Rust code and JavaScript.

REFERENCES

1. arsTECHNICA. The Web is getting its bytecode: WebAssembly. 2015. Date of retrieval 19.9.2021 https://arstechnica.com/information-technology/2015/06/the-web-is-getting-its-bytecode-webassembly/

2. Couriol, B. InfoQ, WebAssembly 1.0 Becomes a W3C Recommendation and the Fourth Language to Run Natively in Browsers. 2019. Date of retrieval 25.3.2021 https://www.infoq.com/news/2019/12/webassembly-w3c-recommendation/

3. WEBASSEMBLY. Date of retrieval 25.11.2020 https://webassembly.org/

4. Rossberg, A. 2019. WebAssembly Core Specifications. Date of retrieval 6.5.2021 https://www.w3.org/TR/wasm-core-1/

5. Adobe. 2021. Adobe Flash Player EOL General Information Page. Date of retrieval 25.10.2021 https://www.adobe.com/fi/products/flashplayer/end-of-life.html

6. AssemblyScript. 2021. A TypeScript-like language for WebAssembly. Date of retrieval 25.10.2021 https://www.assemblyscript.org/

7. FFMPEG.WASM. Date of retrieval 25.10.2021 https://ffmpegwasm.netlify.app/

8. YouTube. 2021. Ben Smith — Raw Wasm: Hand-crafted WebAssembly Demos. Date of retrieval 25.10.2021 https://youtu.be/qEq3F9Z8z6w

9. Photopea. 2021. Date of retrieval 8.11.2021 https://www.photopea.com/

10. Wasmtime. A small and efficient runtime for WebAssembly & WASI. Date of retrieval 25.10.2021 https://wasmtime.dev/

11. Fessel, K. et al. Informatik 2020. Lecture Notes, Programming IoT applications across paradigms based on WebAssembly. Date of retrieval

31.10.2021 https://dl.gi.de/bitstream/handle/20.500.12116/34705/C25-7.pdf

12. MDN Web Docs. 2021. Compiling from Rust to WebAssembly. Date of retrieval 25.10.2021 https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_wasm

13. GitHub, image-rs/image. Date of retrieval 25.10.2021 https://github.com/image-rs/image

14. GitHub, WanzenBug/rqrr. Date of retrieval 25.10.2021 https://github.com/WanzenBug/rqrr

15. GitHub, rustwasm/wasm-bindgen. Date of retrieval 25.10.2021 https://github.com/rustwasm/wasm-bindgen

16. GitHub, LazarSoft/jsqrcode. Date of retrieval 25.10.2021 https://github.com/LazarSoft/jsqrcode

17. Logic, S. & Eberhardt, C. 2021. The State of WebAssembly 2021. Date of retrieval 25.10.2021 https://blog.scottlogic.com/2021/06/21/state-of-wasm.html

18. GitHub, WebAssembly, Interface Types Proposal, Date of retrieval 25.10.2021 https://github.com/WebAssembly/interface-types/blob/main/proposals/interface-types/Explainer.md

19. Clark, L. 2019. Announcing the Bytecode Alliance: Building a secure by default, composable future for WebAssembly. Date of retrieval 25.10.2021 https://hacks.mozilla.org/2019/11/announcing-the-bytecode-alliance/

APPENDICES

Appendix 1 Code Example 1: Rust Code

Appendix 2 Code Example 2: Use of Rust Module in JavaScript

CODE EXAMPLE 1: RUST CODE

Repository: https://github.com/Wilssoni/qr-wasm

At commit 03b7e0d0b0653a6bffe5e28ca718d4e2de4e77ef level

```
use wasm_bindgen::prelude::*;
use image;
use rqrr;

#[wasm_bindgen]
pub fn read_qr(buff:&mut [u8], height: usize, width: usize) -> String
{
    let mut returnstring: String = "".to_string();
    let mut imgbuf = image::RgbaImage::new(width as u32,height as
u32);
    for y in 0..(height as u32) {
        for x in 0..(width as u32) {
            let i: usize = (width * (y as usize) + (x as usize)) * 4;
            //console::log(&format!("{}", i));
            let r: u8 = buff[i];
            let g: u8 = buff[i+1];
            let b: u8 = buff[i+2];
            let a: u8 = 255;
            imgbuf.put_pixel(x,y, image::Rgba([r, g, b, a]));
        }
    }
    let img2 = image::DynamicImage::ImageRgba8(imgbuf);
    let mut img3 = rqrr::PreparedImage::prepare(img2.to_luma8());
    let grids = img3.detect_grids();
    if grids.len() > 0 {
        let (_meta, content) = grids[0].decode().unwrap();
        returnstring = content;
    }
    return returnstring;
}
```

CODE EXAMPLE 2: USE OF RUST MODULE IN JAVASCRIPT

Repository: https://github.com/Wilssoni/qr-wasm

At commit 03b7e0d0b0653a6bffe5e28ca718d4e2de4e77ef level

```
  <script type="module">
  import init, {read_qr} from "./pkg/qr_wasm.js";

  init().then((instance) => {
    var canvas = document.getElementById("canvas1");
    var img = new Image();
    img.src = "src/images/qrcode.png";
    var context = canvas.getContext('2d');
    img.onload = function() {
      context.clearRect(0, 0, canvas.width, canvas.height);
      context.drawImage(img, 0, 0, canvas.width, canvas.height);
      var imageBytes = context.getImageData(0, 0, canvas.width,
canvas.height).data;
      var qrstring = read_qr(imageBytes, canvas.width, canvas.height);
      if (qrstring != "") {
        alert(qrstring);
      }
    }

    var file = document.getElementById("fileselector");
    file.onchange = function() {
      img.src = URL.createObjectURL(this.files[0]);
    };
  });
  </script>
```