WebAssembly and the future of containers

Adibbin Haider and Jakob Holm

I. INTRODUCTION

In the ever changing landscape of web development new technologies always seem to pop up. Since web development usually circles around JavaScript, these new technologies are usually a new JavaScript framework of some sort.

Lately, the maturation of the web has been somewhat held back due to JavaScript being the only natively supported web compilation target. To combat this stagnation, the four major browser vendors (Apple, Google, Mozilla and Microsoft) has joined forces in a common goal - a way of targeting more compilation targets for use in web development. The solution to this problem was named WebAssembly.

WebAssembly is first and foremost built to be executed within web browsers. But, in the same way that Node.js enabled developers to run JavaScript outside of the confinement of browsers, WebAssembly System Interface will enable developers to do the same with WebAssembly.

We wish to investigate what this means for the future of development, and especially the future of containers since WebAssembly together with WebAssembly system interface could have the possibility of making containerized software, such as Docker containers, obsolete.

II. BACKGROUND

In this section we will describe what WebAssembly is and the possibilities it will provide through WebAssembly system interface. We will briefly write about containers also, but we assume some sort of knowledge beforehand since the aim of the essay is to dive into WebAssembly and WebAssembly system interface. But first we will go into the subject of containers.

A. Containers

A container is a type of operating system level virtualization. Meaning, a container instance is running on a host machine with its own isolated user space instance. It is similar to a virtual machine, however, with some important distinctions which results in some key benefits.

A container is an abstraction of the application layer, unlike where a virtual machine is an abstraction of the physical hardware layer. Containers consist of a standard unit of software that has the necessary code and software dependencies packaged. Multiple containers can run on the same host machine and share the OS kernel among the containers. Each of the containers is still running in their own isolated processes in user space [3].

The key feature of containers is that they are not aiming towards hardware level virtualization but instead aiming at an operating system level. This means containers are much smaller and more lightweight. A container only uses a fraction of the memory a virtual machine would use for booting an entire operating system [5].

B. WebAssembly

The current state of the web platform is mostly based on JavaScript, currently the only built-in language for the web. However, it is not well equipped to handle all the requirements of the modern web. The ongoing maturation the web platform has lead to sophisticated and demanding web applications. Demanding applications such as audio and video software, games or applications that are heavily based on interactive 3D visualization - often too demanding for JavaScript .

To meet the demands of the web of today, engineers from the four major browser vendors have come together to solve this problem. Their solution to the problem is something they choose to call WebAssembly.

WebAssembly (Wasm), is the name of the low-level binary the engineers have designed. Instead of designing a new language, they choose to design a new binary for use within web browsers. Instead

of committing to a single programming language, WebAssembly is an abstraction over hardware, meaning that it is language, hardware and platform independent. Developers should be able to choose any desired programming language and build their application and then later compile to WebAssembly binary [6].

Meaning, as a developer, you first choose your programming language to build your project is. It could theoretically be in any language. Currently, WebAssembly supports ten different programming languages that are production ready. Such as C, C#, C++, Go, Rust with more programming languages coming to WebAssembly. Some programming languages are under work in progress, but even more programming languages are stable for usage, but currently not stable enough for production usage [1].

Say a developer has chosen a programming language they wishes to use, for example C++, the developer then uses a C++ compiler that turns the source code into WebAssembly bytecode or directly to a **.wasm** file with the WebAssembly binary code, instead of the usual compilation process. This binary code is then executable in a browser, and also works in parallel with JavaScript in the browser. See figure 1.

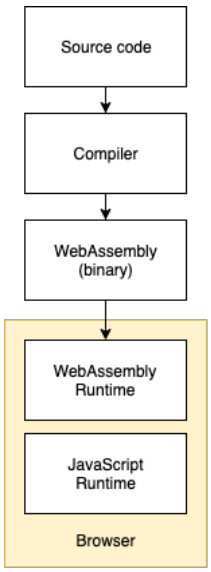

Fig. 1. WebAssembly runtime flowchart.

The creators of WebAssembly designed the low-level binary with specific goals. It needed to be safe, fast, portable and compact. [6].

1) Safe: Safety is crucial, as the WebAssembly code is going to be delivered to different clients and platforms from untrusted sources. The security model of WebAssembly has two goals. Firstly, *"protect users for buggy or malicious modules"*. Secondly, *"provide developers with useful primitives and mitigation's for developer safe applications"* while not breaking the first goal.

To achieve the first goal, WebAssembly modules are executed in a sandboxed environment, isolated from the host at run time. Meaning, as the applications is executed independently, it can not escape the sandboxed environment without using appropriate API:s. To achieve the second goal, WebAssembly has eliminated some dangerous features by supporting memory safety that to avoid inheriting some problems that come with C, C++ and other potential low level languages [6] [11] [4].

2) Fast: Low level machine code that is emitted by a C or C++ compiler is often optimized ahead of time and can utilize the full performance of the machine. With WebAssembly, a developer should still be able to keep the optimized performance even after it has been compiled to WebAssembly binary [6].

3) Portable: The modern web platform is accessible on many different devices with different machine architecture, operating systems and browsers. WebAssembly binary must therefore be hardware and platform independent to allow applications to run on all different browsers and devices, while still achieving the same behaviour and experience. WebAssembly today support of encoding the binary to binary format which is executable on different operating systems and instruction set architectures. On the web, and off the web [6] [10].

4) Compact: When accessing a website, the source code is transmitted over the network, therefore it should be as compact as possible. This will reduce load times, save bandwidth and lead to a faster and a more responsive experience on the web [6].

C. Security risks of WebAssembly

The idea when building anything with WebAssembly, is that a developer writes the program in any desired programming language and then

later compiles it to a WebAssembly module. The module is then executable on the browser. This sounds promising, however, it is also a potential security issue.

Unlike JavaScript, where it is possible to see the original source code in clear text in the browser, it's not possible with WebAssembly. WebAssembly modules consist of binary, meaning that in the browser, it is only possible to see the binary and not the original source code. This makes it harder to detect security threats, as potentially malicious code is harder to detect when the code is obfuscated in binary [2].

Furthermore, at the moment there is no way to integrity check a WebAssembly module. This means, there is no way to know if someone has tampered with the WebAssembly module [2].

D. WebAssembly system interface

Since Wasm provides a fast, portable, scalable and secure way of running the same code across a wast array of machines (desktop and mobile) running web browsers, the need for an interface in between Wasm and the systems running it arose. In march of 2019, a new standardization effort to create this interface was announced. This effort was named WASI, the WebAssembly system interface.

The tenants of this effort reeks of Java's original "Write Once, Run Anywhere" motto. But contrary to the Java Virtual Machine that only handles Java, the conceptual machine which handles Wasm does not limit itself to only Java, since potentially any programming language can be compiled to Wasm binary [9]. The goal with WASI is thus to expand the platforms capable of running the same Wasm code from the platforms running the web browsers currently supporting Wasm to a vast array of devices, machines and operating systems.

Another big difference between the JVM and WASI is that JVM is a oracle project at the moment, whereas WASI is based on Wasm which is supported by a multitude of big tech companies (Apple, Google et al.). Thus it could possibly become a bigger standardization effort than the JVM could ever hope to be since the main platform for Wasm is the most ubiquitous platform available - the web [8].

Compiling the same source code against many different targets is not something new. With help from the POSIX standard, developers could easily compile the same source code for use in different kinds of machines. However, one could argue that it is not as portable as Wasm because several compilation targets are needed, see figure 2.

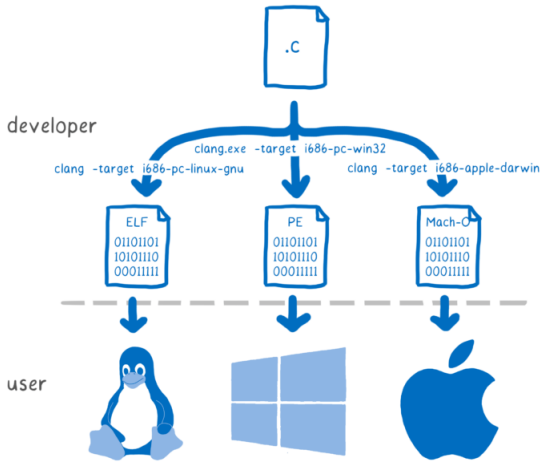

Fig. 2. Figure of C source code compilation that targets three different platforms. This example needs three different compilation targets: ELF (Linux), PE (Windows) and Mach-O (Mac) [9].

The developers of WASI will start developing something that is reminiscent of POSIX. With the goal of a single compilation target, see figure 3.

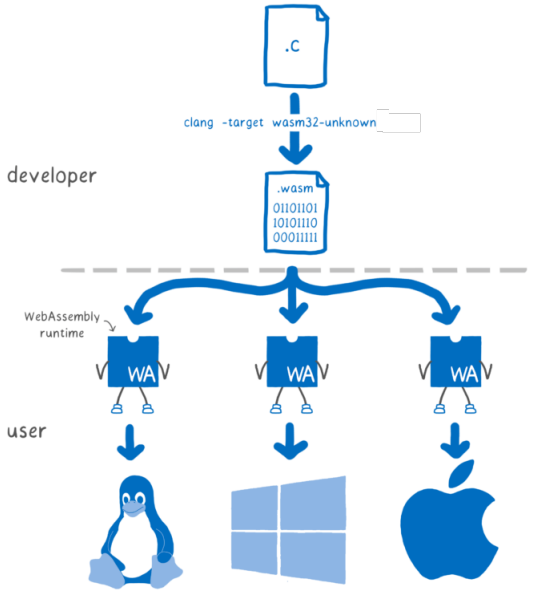

Fig. 3. Figure of C source code compilation that targets three different platforms. This example needs one compilation targets: Wasm [9].

The goal of WASI is not to replace previous standardization efforts, the JVM for example will probably exist just as fine along side WASI. But the possibilities with WebAssembly system interface are big, for example Solomon Hykes the creator of Docker had this to say about Wasm together with WebAssembly system interface:

> ”If WASM+WASI existed in 2008, we wouldn’t have needed to created Docker. That’s how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let’s hope WASI is up to the task!” [7]

III. DISCUSSION

Like with most new technology, it is sometimes difficult to see past the hype and the noise. The WebAssembly system interface suffers from the same fate to some degree. It is easy to see that WASI easily triggers questions like ”How is this different from the JVM/CLR and Java/.Net”. Seeing as WASI is it its infancy (when we wrote this essay, WASI has existed for a little more than a month) it is too early to tell if WASI will be another failed standardization attempt. But the possibilities of running C code in a sandboxed secure environment could lead to improvements definitely. And also, we think that the reasons that CLR and JVM failed its promises of standardization is partly due to they put in huge amounts of efforts into things that did not need solving. This could be rectified with WASI since it is developed with this experience in mind and uses the experiences and wrongdoings of past standardization efforts. One also needs to remember that the JVM and CLR was language platforms firstly, whereas WASI aims to be a more ubiquitous VM solution.

We do not believe that WASI will replace containerized applications in the short term. But in the long term? We do not know but if WASI lives up to its promises it certainly has the ability to replace containers, and the Docker creator also states that he would not have needed to develop Docker in 2008. However, just like Docker and containerized applications now has started to gain traction, we also believe that it will take some time before WASI gains widespread adoptation. And one also needs to remember that WASI still is it its infancy and still under development. We do not know if all the promises will end up being fulfilled, but if they are fulfilled the possibilities of one single compilation target is indeed very nice and promising.

IV. CONCLUSION

It is very hard to tell if WASI will be the ubiquitous ”Write Once, Run Anywhere” solution its creators are promising it to be. But Docker’s Solomon Hykes certainly believes the potential is there. However, we believe that in the short term WASI will not replace containerized software, the likelihood that developers will change their entire technology stack on unsubstantiated promises and hype is extremely small. Furthermore, as the history of software development has shown, the adaptation rate of new technology is sometimes slow (DevOps as a whole only now has found widespread adaptation). As such, if Wasm together with WASI proves it self in real world applications, it might actually gain some traction. Maybe it will co-exist inside actual Wasm-containers alongside today’s Docker containers, only time will tell.

REFERENCES

[1] Steve Akinyemi. A curated list of languages that compile directly to or have their vms in webassembly. https://github.com/appcypher/awesome-wasm-langs. [Online; accessed 29-April-2019].

[2] John Bergbom. Webassembly security: potentials and pitfalls. https://www.forcepoint.com/blog/security-labs/webassembly-potentials-and-pitfalls, June 2018. [Online; accessed 29-April-2019].

[3] What is a container? https://www.docker.com/resources/what-container. [Online; accessed 29-April-2019].

[4] Jonathan Foote. Hijacking the control flow of a webassembly program. https://www.fastly.com/blog/hijacking-control-flow-webassembly/, June 2018. [Online; accessed 29-April-2019].

[5] Containers at google. https://cloud.google.com/containers/. [Online; accessed 29-April-2019].

[6] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, June 2017.

[7] Solomon Hykes. Solomon hykes (@solomonstre) — twitter. https://twitter.com/solomonstre/status/1111004913222324225. [Online; accessed 29-April-2019].

[8] Mozilla tries to do java as it should have been – with a wasi spec for all devices, computers, operating systems. https://www.theregister.co.uk/2019/03/29/mozilla_wasi_spec/. [Online; accessed 29-April-2019].

[9] Standardizing wasi: A system interface to run webassembly outside the web. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/. [Online; accessed 29-April-2019].

[10] Portability. https://webassembly.org/docs/portability/.

[11] Security. https://webassembly.org/docs/security/. [Online; accessed 29-April-2019].