DD2482 Automated Software Testing and DevOps

Could WebAssembly System Interface replace containers?

Members:
Jakob Holm
Adibbin Haider
Email:
jakobhol@kth.se
adibbin@kth.se

Contents

1 Introduction

On the 27th of March, Solomon Hykes, the founder of Docker, took to Twitter to comment on the recent announcement of the WebAssembly system interface (WASI):

> "If WASM[WebAssembly]+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!" [1]

This peaked our curiosity, having just learned containerizing applications with Docker, would we soon need to learn something new again?

To learn more about this we decided to look further into WebAssembly and its new system interface to learn more about what it was that got Solomon Hykes so excited.

2 Containers

The purpose of this essay is to investigate if the WebAssembly system interface could replace current container technologies. Thus, we think it is good to start with a brief explanation of what a container is.

A container is a type of operating system level virtualization. In other words, a containerized 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 [2].

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 [3].

3 WebAssembly

In order to compare the WebAssembly system interface with containerized applications, it is important that we have a technical understanding of what WebAssembly is. WebAssembly is the foundation and the backbone of WebAssembly system interface.

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 more sophisticated and demanding web applications. Demanding applications such as audio and video software, games or applications that are heavily based on interactive 3D visualizations, which often are too demanding for JavaScript to handle.

To meet the demands of the web of today, engineers from the four major browser vendors (Microsoft, Mozilla, Apple and Google) came 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, WASM is an abstraction over hardware. WASM is meant to be language, hardware and platform independent. Developers should thus be able to choose any desired programming language and build their application and then later compile it to a WASM binary [4].

This is beneficial for many developers, since they can choose any desired programming language to build their project in. It could theoretically be in any language. Currently, WASM supports ten different programming languages on a ”production ready” level. C, C#, C++, Go, Rust are some of them. 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 [5].

When a developer has chosen a programming language she/he desires to use, for example C++. The developer then uses a C++ compiler that turns the source code into WASM bytecode or directly to a **.wasm** file with the WASM 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 3.1.

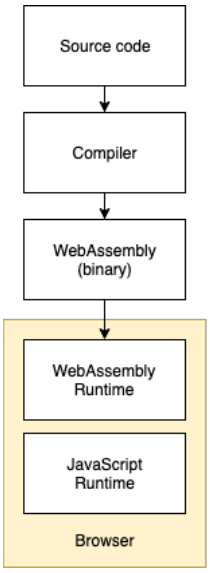

Figure 3.1: WebAssembly runtime flowchart.

The creators of WASM designed the low-level binary with specific goals. It needed to be safe, fast, portable and compact. [4].

Safe
Safety is crucial, as the WASM code is going to be delivered to different clients and platforms from untrusted sources. The security model of WASM 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, WASM modules are executed in a sandboxed environment, isolated from the host system at run time. The applications are executed independently, it can not escape the sandboxed environment without using appropriate APIs. To achieve the second goal, WASM 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 [4] [6] [7].

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 WASM, a developer should still be able to keep the optimized performance even after it has been compiled to WASM binary [4].

Portable
The modern web platform is accessible on many different devices with different machine architecture, operating systems and browsers. WASM 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. WASM 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 [4] [8].

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 [4].

3.1 Security risks of WebAssembly

The idea when building anything with WASM, is that a developer writes the program in any desired programming language and then later compiles it to a WASM 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 WASM. WASM modules consist of binary code, resulting it is not possible to see the original source code when the modules are executed. It is only possible to see binary code of the modules. This could potentially make it harder to detect serious security threats, as malicious source code is harder to detect when the source code is obfuscated in binary [9].

Furthermore, at the moment there is no way to integrity check a WASM module. This means, there is no way to know if someone has tampered with the WASM module [9].

4 Extending WebAssembly with a System interface

Since WASM provides a fast, portable, scalable and secure way of running the same code across a vast 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 (JVM) 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 [10]. 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 [11].

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 4.1.

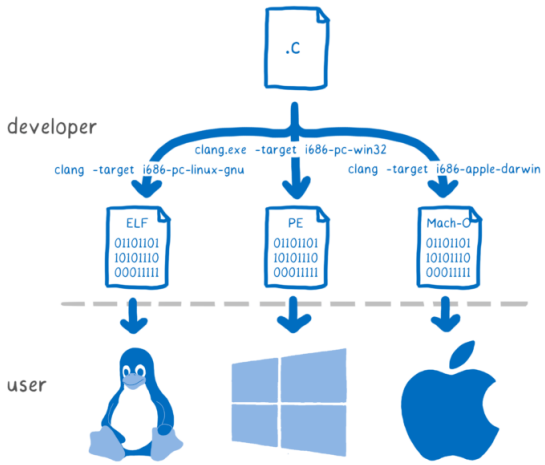

Figure 4.1: Figure of C source code compilation that targets three different platforms. This example needs three different compilation targets: ELF (Linux), PE (Windows) and Mach-O (Mac) [10].

The developers of WASI will start developing something that is reminiscent of POSIX. With the goal of a single compilation target, see figure 4.2.

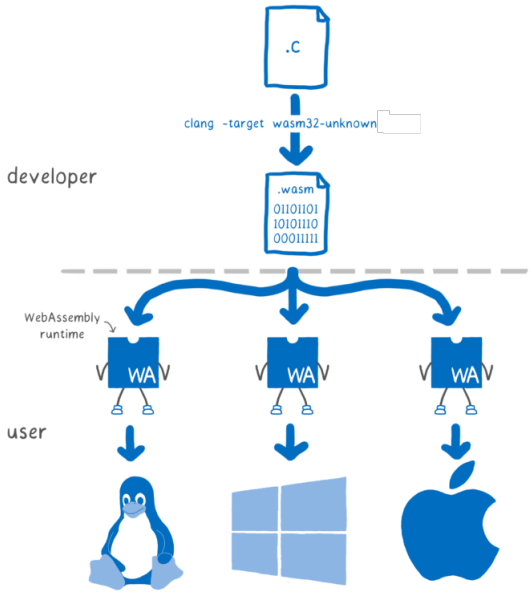

Figure 4.2: Figure of C source code compilation that targets three different platforms. This example needs one compilation targets: WASM [10].

The goal of WASI is not to replace previous standardization efforts, the JVM for example will of course continue to co-exist just as fine along side WASI.

Regarding security WASI will also offer the same sandboxed environment that WASM offers, but it will extend upon it with capability-based security concepts. Hence, WASM through WASI will get closer to the principle of least privilege - only access to resources it need to execute the jobs [10].

5 Comparison between Containers and the WebAssembly system interface

When comparing containers with WASI, we can also compare containers with WASM. WASM is the foundation of WASI, and its features extends over to WASI. We found two main areas where the technologies and their platform differs. Portability and security.

5.1 Portability

One of the main features of WASM is portability. It should be hardware and platform independent. Together with the system interface, WebAssembly modules can run outside of the browser. Thus, as a developer you can write your code in any desired language and then compile your code to a WebbAssembly module which can be executed on any hardware or any platform through WASI. Container providers, such as Docker, are also portable but with some caveats. For example, in order to build a Docker image of Java application, one will need a Java base image. Should a base image not exist for the language of your choosing, you will not be able to containerize it.

5.2 Security

By default, most containers are running with root access. This can enable attack vectors for an attacker. In other words, containers are not properly sandboxed. Since WASI is an extension of WebAssembly, it will offer the same sand boxed environment, built with capability-based security in mind. This will result in a safer environment.

Even though WASM + WASI is sandboxed, it does still have some known vulnerabilities at its current state. With the lack of tooling and integrity check it is difficult to ensure a WASM module is safe to use and the modules has not been tampered with.

6 Discussion

Like with most new technology, it is sometimes difficult to see past the hype and the noise. We think that WASI suffers from the same fate to some degree.

We do not believe that WASI will replace containerized applications in the short term, even if it technically could. However, in the long term? It is hard to tell, firstly since WASI is such a new technology, we cannot really tell what WASI could accomplish on a technical level. Mostly since we find it hard to believe that WASI, specifically as a container alternative, will be widely adopted. Looking back on the DevOps field as a whole, it took some time before DevOps with containers found widespread adaptation. If companies has just started to convert their stack to, say Docker containers using Kubernetes/Openshift for orchestration, will they start changing their entire stack? Change their stack to something new that has not been properly tested? Most likely not. We cannot, today, be sure that WASM + WASI will offer the same streamlined possibilities that current DevOps technologies does.

As any other new technology or programming language, you almost always have the problem with lack of proper documentation and tooling. This is a further obstacle for companies and

other potential early adopters. Lack of documentation could hinder the curious early adopter. Lack of tooling could hinder companies to invest their time in WASM + WASI as it may not be supported together with the rest of their stack.

Running WASM + WASI on a backend is promising indeed, programmers will then be able to run C programs (or any language for that matter) in a secure and properly sandboxed environment. This is one of the best features that WASM + WASI currently has to offer. Furthermore, one also needs to remember that WASM, and especially WASI, still is it their infancy and still under development. We do not know if all the promises will end up being fulfilled. If they are fulfilled the possibilities of one single compilation target is indeed very desirable and promising.

7 Conclusion

For the key differences between current container technologies and WASM + WASI, see table 7.1.

In a naive scenario, WASM together with a system interface is a competitor or replacement of containerized applications. However, firstly the creators and pioneers of WASI needs to actually implement all of their promises. Hardware and platform agnostic while retaining close to native performance. If the developers of WASI truly achieve what Java once tried to, i.e. be the ubiquitous ”write once run anywhere” solution, there is indeed a potential outcome where WebAssembly and the WebAssembly system interface replaces the containers of today.

However, in a more realistic scenario, there will still be a need for containerized applications. WebAssembly, with the system interface, will probably co-exist in WASM + WASI containers alongside ”ordinary” containers. Only time can tell if WebAssembly together with the system interface is actually going to replace Docker.

Summary		
Technology	Pros	Cons
Containers	• adapted by the industry • documentation • stable	• less secure
WASM + WASI	• language agnostic • sandboxed environment (more secure)	• uncertain future • immature technology • lack of documentation • lack of tooling • attractive for malicious usage

Table 7.1: Summary of the key differences between current container technologies and WASM + WASI.

References

[1] Solomon Hykes. Solomon hykes (@solomonstre) | twitter. https://twitter.com/solomonstre/status/1111004913222324225. [Online; accessed 28-May-2019].

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

[3] Containers at google. https://cloud.google.com/containers/. [Online; accessed 27-May-2019].

[4] 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.

[5] 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].

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

[7] 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].

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

[9] 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].

[10] 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 28-May-2019].

[11] 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].