WebAssembly: Basics

Lennard Golsch
Technical University of Braunschweig
Brunswick, Germany
l.golsch@tu-braunschweig.de

Abstract

With the availability of the internet, it is hard to imagine today's world without web applications. By historical accident, JavaScript is the only widely supported programming language on the Web. For a long time no other standard could be established for the calculations on the client side of web applications. The evolution of hardware enables end devices to handle more complex applications. Because of its old-fashioned architecture, JavaScript is an obstacle to the evolution of web applications. For this reason, developers from the relevant browser manufacturers have joined together to develop a solution. As a result, WebAssembly has been available since late 2017. The goal of the development is a supplement to JavaScript, which should improve loading times as well as execution. This paper gives an overview of WebAssembly, discusses the approach as well as with respect to its practical predecessor like Google Native Client.

Keywords WebAssembly, web, bytecode, stack machine

1 Introduction

Exactly 50 years ago Tim Berners-Lee established the foundation for today's *web*. First the web was developed for the transmission and presentation of simple documents over the *internet* in web browsers. However, advanced hardware and software development has had an impact on the web. The simple documents quickly became more complex *web applications*. A web application is an application program based on the client-server model. Instead of being installed like a classic desktop application, a web application is accessible via a browser, for example. Normally, parts are outsourced to the client where the calculations can be performed and executed later to save server-side resources.

Through historical accident *JavaScript* is the only natively supported programming language on the web. Today JavaScript has established itself for client-side calculations and is the main component of web applications on the client side. "The vast majority of websites, including large social networks, such as Facebook and Twitter, makes heavy use of JavaScript for enhancing the appearance and functionality of their services" [24]. Also for this reason, the web is no longer conceivable without web applications. JavaScript is popular because it is available on all common platforms and at the same time it offers relatively good performance. Statistically, Javascript has been the most popular programming language on *Github* over the last five years [7].

However, JavaScript is limited because of its functionality, which affects the performance of the applications. Complex applications like *3D-applications* such as games, are not really feasible. In terms of the relevance and potential of a web application, JavaScript is such an obstacle to the development of the web. In summary, no high-performance standard could establish itself more than JavaScript. For this reason Haas et al. [16] have developed *WebAssembly* (WebAssembly). The motivation behind WebAssembly is generally a new standard for the web, which should offer a performant execution. "WebAssembly is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications" [15].

This paper deals with WebAssembly and is mainly based on the initial paper [16]. In the following chapters background information is given. This is followed by basic information about WebAssembly in terms of motivation, features, implementation and performance. Afterwards related work is presented and a conclusion to WebAssembly follows.

2 Background

The following chapter describes background information about JavaScript and additionally discusses approaches that can be considered as predecessors of WebAssembly.

JavaScript is an interpreted programming language that is object-oriented for client-side programming. Initially it was developed for dynamic *HTML* in web browsers. The first version was developed for the web by Netscape, special Brendan Eich, and released in 1995. Fundamentally, JavaScript is a scripting language that is *interpreted*. JavaScript is available on all relevant platforms. Code is transmitted in *plain text* and is interpreted by an *engine*. But performance today is improved by *just-in-time compilation* (JIT). The engine runs isolated from the operating system context in a *sandbox* and communicates with the web browser. There is an approach with *Node.js* that enables JavaScript on the server side.

With *Java* or Adobes *Flash* there exist similar approaches to JavaScript. However, an additional plugin usually has to be installed for execution. Flash enables the execution of *ActionScript*. ActionScript is interpreted and executed by the Flash plugin very similar to JavaScript. Especially in Flash there have been security vulnerabilities, so that browser manufacturers advise against the use. Flash is used today by 3.2 % of all websites [8] and Adobe has announced that it will stop support and distribution at the end of 2020 [18]. Java applets allow the execution of Java code in web browsers.

The Java code is executed in an isolated *Java VM* that is either integrated in the browser or can be installed with a plugin. Mainly because of security vulnerabilities the leading browsers mostly do not support applets any longer.

Many approaches are based on running native code to improve application performance. Already in 1996 Mircosoft published an approach with *ActiveX* to support binaries in the browser. ActiveX requires the *Component Object Model* (COM) for the application. COM is a Microsoft approach and can generally only be found in *Internet Explorer.* ActiveX objects already include compiled and therefore platform-dependent machine code. [21] The architecture of ActiveX is classified as security critical, since the execution is not isolated from the operating system as in a sandbox.

Google has also published an approach with the *Google Native Client* in 2011. NaCl is a sandboxing technology that allows native code to be executed in a *sandbox.* With a special SDK, C/C++ source code can be compiled into a so-called *nexe* module. These modules can be executed independently of the operating system in the in-house chromium environment. However, NaCl supports only a handful of hardware architectures and is therefore not platform independent. [9]

Later, Google introduced *Portable Native Client* (PNaCl). PNaCl is an extension of NaCl that allows native code to be executed platform-independently. Thereby PNaCl supports the execution of native code on all relevant platforms like *Intel x86*, *ARM* or *MIPS*. Native code can be compiled via a compiler into so-called pexe modules which is bytecode. These pexe modules are translated into the platform architecture on the client side where they are executed. Nevertheless, NaCl and PNaCl are only supported by Google Chrome and Chromium and therefore no general approach. [9, 25]

With *asm.js*, Mozilla has provided in 2013 an intermediate language for porting applications written in *C*, for example. This should provide a better performance than with JavaScript. In principle, the code is systematically generated from the native code using a *transcompiler* such as *Emscripten.* A transcompiler is a compiler that specializes in converting the source code of a program into another language. Emscripten is generally able to compile LLVM [20] assembler code to standard JavaScript-Code. It can be used to convert simple *C/C++* which is based on Clang and LLVM to JavaScript code. Fundamental, asm.js is based on a subset of JavaScript, but still achieves better performance than ordinary JavaScript code by adding conventions. Due to the use of JavaScript, asm.js is mostly supported by the common browser manufacturers. [17]

3 WebAssembly

WebAssembly (Wasm) is the result of a merger of the leading browser manufacturers Apple, Google, Microsoft and Mozilla, which joined together in 2015. In principle WebAssembly combines the advantages of the approaches and results of ActiveX, PNaCl and asm.js and was developed mainly by the minds of these techniques. Due to the know-how and the clear goal, the WebAssembly was published relatively quickly. The first WebAssembly version was released in November 2017. To ensure that WebAssembly follows its principles, the developers have set themselves two main goals, semantics and representation [16]:

Safe, fast, and portable semantics WebAssembly should be fast and secure at the same time. The efficiency is to be increased by a language that is oriented to a low-level language. Safety should be given by the execution in a sandbox. Simultaneously, WebAssembly should be operable on every system due to its language, hardware and platform independence. Furthermore, WebAssembly should be deterministic and easy to understand via simple interoperability with the web platform.

Safe and efficient representation WebAssembly should be based on a compact representation that is easy to code. At the same time, the representation should be easy to validate and compile. In terms of JavaScript, bytecode is much more compact and efficient. The compactness is intended to minimize loading times caused by the transmission. In order to do justice to the Web, the representation should support streaming and parallelizability. In addition, WebAssembly code should be easy to produce.

The following chapter deals with the achievement of the goals of the developers, mainly with design and implementation. Measurement results are also discussed. In addition, the development and integration of WebAssembly is covered.

3.1 Module

A WebAssembly binary is defined by a so-called *module.* It is the distributable, loadable and executable code unit in WebAssembly. WebAssembly works according to the *stack machine* principle. Instructions which are executed and the result of each operation is pushed to the stack and the next instruction may push or pop to the stack. A total of 4 *data types* and 67 *instructions* are available. WebAssembly modules are not independently executable, they need a so-called *embedder*, which instantiates the modules and also manages the import and export. More will be discussed in section 3.4.2.

3.1.1 Sections

Subsequently, 11 definitions and sections can be described in one module, which are necessary for its execution. Each module starts with the so-called *magic number*, for example `0x6d736100` (i.e., ' \0asm') and the so-called *version* `0x1`. It is possible to declare a *main function* with `start` in a module, which is called after initialization, see Figure 3. [15, 16] The most important sections include the following:

Data Section This section is similar to the *.data* section of native executables. Among other things, static and global variables are stored here. A global variable consists of a *type*, *initializer* and a *mutability flag*. [15]

Function and Code Section A function is divided into a function section and a code section. In the function section the signature of a function is described. The code section contains the body or the code of this function. [15]

Import and Export These sections define specific imports and exports of a WebAssembly module. WebAssembly supports imports and exports from other modules or the host environment. For example, *functions* or *global variables* can be imported and exported from other modules. Concerning JavaScript, for example, it is possible that a WebAssembly module gets access to the object `console.log` by importing it from the JavaScript environment. Imports can be called using the `call` operator. Once a module instance imports a definition, it is shared with the exporting instance. [15, 16]

Memory Section The description of the *linear memory* is assigned to this section, see section 3.2. The section defines the initial size of the memory, which can be dynamically expanded at runtime. [15]

3.1.2 Representation

The inventors provide two formats for coding a module. There is a binary and a textual representation. A module in binary representation encodes the typical assembly instructions and is declared with the suffix `.wasm`. The inventors attached great importance to a text representation. For this reason, there is a textual representation that encodes the instruction in a module textually. This has the intention that code can be written by hand, is more understandable and therefore easier to debug. Section 3.4.1 discusses the topic more deeply. A module in text representation has the suffix `wat`. There are tools that allow lossless conversion between the two representations[1]. Table 1 shows a comparison of the representation of a WebAssembly module in C++, binary, and text (linear assembly bytecode).

The inventors have chosen to transfer WebAssembly modules usually in binary representation in order to speed up the transfer by this compactness compared to text or native code. But transmission in text representation is possible. Section 4.2 deals with the extent of compactness of the code. Furthermore, modules can be split into parts and thus can be processed separately. In general, the efficiency of the processing is to be improved by streamability. [14–16]

[1]Many short and useful examples can be found in the following repository: https://github.com/mdn/webassembly-examples/

C++	Binary	Text
	20 00	`get_local 0`
	42 00	`i64.const 0`
	51	`i64.eq`
	04 7e	`if i64`
`int factorial(int n) {`	42 01	`  i64.const 1`
`  if (n == 0)`	05	`else`
`    return 1;`	20 00	`  get_local 0`
`  else`	20 00	`  get_local 0`
`    return n * factorial(n - 1);`	42 01	`  i64.const 1`
`}`	7d	`  i64.sub`
	10 00	`  call 0`
	7e	`  i64.mul`
	0b	`end`

Table 1. Sample function, illustrated in C/C++, binary representation and textual assembly representation. [15]

3.2 Memory

Each WebAssembly instance has a specially provided linear standard memory, the so-called *linear memory*, which is either imported or defined within the module in the memory section. A linear memory is a contiguous, byte addressable memory area and can be regarded as an untyped array of bytes. The WebAssembly page size is set to 64 KiB. The linear memory is sandboxed and is shared with the embedder, for example, a JavaScript instance WebAssembly modules share a linear memory with JavaScript. Thus imports of JavaScript are possible in WebAssembly and vice versa. [15]

3.3 Engine

The browser manufacturers extended their existing JavaScript engines to support WebAssembly. Google Chrome uses *V8*, Mozilla Firefox uses *SpiderMonkey*, Apple's *Safari* uses *JavaScriptCore* and *Microsoft Edge* uses its JavaScript engine *Chakra*. As a result, the engines use different approaches, for example for compilation or caching, in order to keep up with the competition. For example, the compilers use the streaming capability of the modules via *HTTP* to reduce compilation time by compiling in parallel in multiple threads. This special subdivision of code and function should above all enable paralization, for example during compilation, in order to make the step easier. These differences are particularly noticeable in the execution time. Important approaches of the engines are described below:

Compiler Compiling is the core of an engine. This step can be very noticeable in the execution time, as depicted in Section 4.1. The engines follow different approaches when starting the compilation. For example, Microsoft's Chakra takes a unique approach to compilation. It tries to optimize the commissioning and the memory consumption at the first execution by initially naive compiling. Mozilla's SpiderMonkey engine uses his *Baseline JIT* [12], designed for a quick

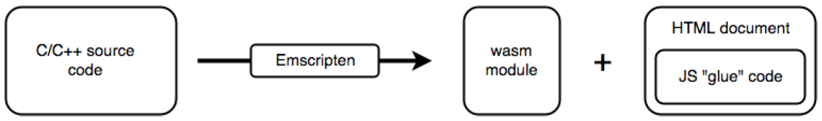

Figure 1. General sequence of porting native code with Emscripten into a WebAssembly module and glue code. [3]

start and mainly handles validation. At the highest compilation level, the JIT of JavaScript is then reused by all four engines for the maximum peak performance of WebAssembly. [16]

Validation Because modules come from non-secure sources, they must first be validated to ensure safe execution. In general, modules are validated on-the-fly, with only a few simple rules as in JavaScript, for example, in order to maintain performance. In principle, *typing rules*, which deal with contexts and instructions for example, are used. [16]

Caching WebAssembly modules can be cached in conjunction with the JavaScript API. The compiled code can be cached into the *IndexedDB* of the web browser. This means that the same code may not have to be recompiled after several attempts, but can be loaded from the cache. So far only V8 and SpiderMonkey support the concept of caching. [16]

3.4 Deployment

One design goal was it to produce WebAssembly code easily. Meanwhile, WebAssembly is noticeably represented in the Web. Statistics show that 1 out of 600 sites of the *Alexa Top 1 million websites* run WebAssembly. [22] In this section approaches for developing a WebAssembly module are described as well as the integration into a platform like the browser.

3.4.1 Development

There are many approaches to create WebAssembly modules. Approaches exist for converting code from other languages to WebAssembly, as well as manual approaches for developers to produce code themselves. Developers can convert modules into a textual representation to better understand the code. For this reason it is also possible to develop modules at assembly level in text representation, like in Figure 3. This section describes common and efficient approaches [3] for creating WebAssembly:

Transcompiling with Emscripten Already at the time of asm.js the approach was pursued to convert code with a transcompiler into a target language. "WebAssembly is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications" [15]. Emscripten supports the conversion of high-level languages with LLVM-front-end to WebAssembly, like C/C++ or Rust. Currently, WebAssembly does not include a native *Garbage Collector* (GC). For this reason, only high-level languages are currently compatible with manual memory management. Nevertheless, a GC in the form of a module can be added and thus dynamic languages can be executed.

Figure 1 illustrates the concept behind the transcompilation of native code in WebAssembly with Emscripten. Emscripten automatically converts native C/C++ code into a module and *glue code*. The glue code embeds the module and is indispensable for the execution of the module. For more information see section 3.4.2. [3]

AssemblyScript *AssemblyScript* allows the development of WebAssembly modules manually. It is a strict subset of *TypeScript*. In a figurative sense, this approach can be used to compile TypeScript in WebAssembly. However, there are limitations so that TypeScript code cannot be compiled naively without modifications in WebAssembly. TypeScript is usually based on the features of JavaScript. AssemblyScript addresses the features of WebAssembly. This is noticeable, for example, in the typing or also in the avoidance of dynamic elements such as JavaScript. [2]

3.4.2 Embedding

Basically WebAssembly modules cannot be executed independently and are therefore dependent on a system that instantiates the module. For the integration of WebAssembly modules into web applications a JavaScript API is available. *Node.js* is based on V8 and supports for example the integration of WebAssembly modules.

Because of the attractive features of WebAssembly, developers have also made efforts to use WebAssembly beyond the browser. *WebAssembly System Interface* (WASI) provides a system interface to run WebAssembly outside the web. [10, 15] The following section deals with approaches for embedding:

JavaScript API A motivation behind WebAssembly is fundamentally high-performance web applications. WebAssembly cannot run stand-alone and depends on an embedder. For this reason there is an approach to integrate WebAssembly modules with JavaScript. WebAssembly Modules can be managed in web applications using a JavaScript API. The API provides functions with which it is possible to load or compile WebAssembly modules.

```
1   var importObj = {js: {
2      import1: () => console.log("hello,"),
3      import2: () => console.log("world!")
4   }};
5   fetch('demo.wasm').then(response =>
6      response.arrayBuffer()
7   ).then(buffer =>
8      WebAssembly.instantiate(buffer, importObj)
9   ).then(({module, instance}) =>
10     instance.exports.f()
11  );
```

Figure 2. JavaScript glue code that initializes and executes the module from Figure 3. [13]

Figure 2 describes the instantiation of a module which is represented in Figure 3. First the module is fetched and instantiated in line 8. During instantiation, both the module and imports are passed. The functions from lines 2 and 3 are used. Then the function f from the module is called in JavaScript in line 10. After execution, "hello, " and "world!" are logged in the console.

```
(module
   (import "js" "import1" (func $i1))
   (import "js" "import2" (func $i2))
   (func $main (call $i1))
   (start $main)
   (func (export "f") (call $i2))
)
```

Figure 3. Sample code that imports and executes functions through the glue code in Figure 2. [13]

WASI WASI is a system interface for the WebAssembly platform. The main goal is to create another platform for WebAssembly in addition to the browser. Worth mentioning implementations are *Wasmtime* [6] or *Lucet* [4]. [10]

4 Measurements

Main design goals of WebAssembly are a fast execution time and also a compact presentation. Haas et al. [16] evaluated WebAssembly 2017 with the introduction using PolyBenchC[2] to confirm performance. For this asm.js and native code was compared with WebAssembly. The following chapter deals with measurement, which provides information about the performance, but also about the compactness of the code.

[2]PolyBenchC is a benchmark suite of 30 numerical computations with static control flow, extracted from operations in various application domains (linear algebra computations, image processing, physics simulation, dynamic programming, statistics, etc.). [5]

More recent benchmarks and criticism can also be found in section 5.

4.1 Performance

Figure 4 shows the results of the benchmark compared to native code. For the measurement, the individual test cases were converted from PolyBenchC to WebAssembly, executed and compared with the execution time of native code. PolybenchC was chosen because the tests do not execute *system calls* and are therefore compatible with the current architecture. The individual modules were each executed with V8 and SpiderMonkey. Generally, there is not the best engine. V8 performs best in some test cases, SpiderMonkey in others. In the diagram the difference of the execution time is marked with *difference between VM's*. The X-axis shows the individual PolyBenchC test cases and the Y axis shows the execution time normalized to native code. The measurement shows that WebAssembly can compete with native code in execution. Overall, the results show that 7 benchmarks are in the range of 10 % of native code. Almost all tests could be performed with half speed.

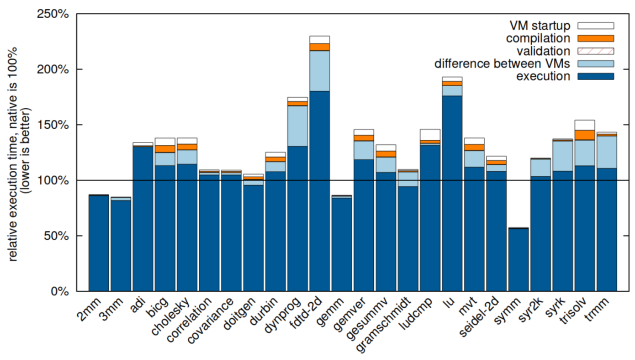

Figure 4. Relative execution time of the PolyBenchC benchmarks on WebAssembly normalized to native code. [16]

The execution time was also compared to asm. js. Therefore, the PolyBenchC tests were converted to asm.js. Tests show that WebAssembly is 34 % faster than asm.js.

4.2 Compactness

In addition to PolyBenchC, other test cases were also used to make the compactness more meaningful. Figure 5 compares the size between WebAssembly (generated from asm.js inside V8), minified asm.js, and *x86-64* native code. asm.js was compared with code from Unity Benchmarks. Native code was compared with tests from PolyBenchC and SciMark. The X axis represents the size of native code (blue) and asm.js (yellow) in bytes compared to the size of WebAssembly on the Y axis. Any point below the diagonal represents a function for which WebAssembly is smaller than the corresponding other representation. On average, WebAssembly code is 62.5 % the size of asm.js (median 68.6 %) and 85.3 % of native x86-64 code size (median 78 %).

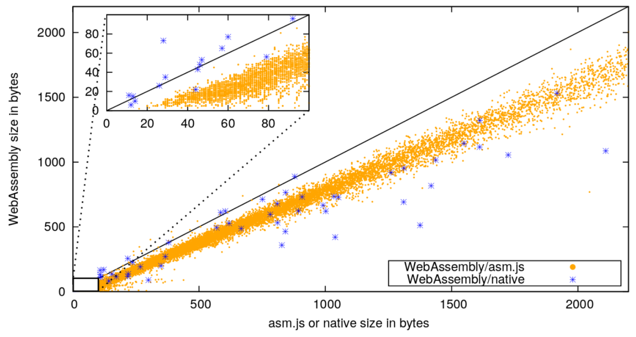

Figure 5. Binary size of WebAssembly in comparison to asm.js and native code. [16]

5 Related Work

As already described in section 3 WebAssembly is based on the approaches of (P)NaCl [25] and asm.js [17].

The WebAssembly masterminds Haas et al. [16] describe the implementation of WebAssembly even more deeply in their work and go into semantics in more detail.

However, Abhinav Jangda et al. [19] show in their work from 2019 that meanwhile 13 tests can approach the speed of native code and thus show that WebAssembly can achieve much better performance related to PolyBench. Nevertheless, they question the benchmarks, because in their opinion PolyCBenchmark is not meaningful enough for practical applications because of the missing syscalls. For this reason they have developed *BROWSIX-WASM*, an approach of *BROWSIX*[3] extended by WebAssembly that allows unmodified WebAssembly-compiled Unix applications directly inside the browser to get meaningful benchmarks that treat native code and web assembly. Benchmark Suite SPEC CPU 2017[4], that simulates real user applications, was used for the evaluation. The result of their evaluation is that on average, WebAssembly is 1.55× slower than native code in Chrome and 1.45× slower than native in Firefox. They mainly make the register accesses responsible for the poor performance. Also, the jitted WebAssembly has about 1.75× more instructions than the native code. Compared to asm.js, WebAssembly is 1.3× faster.

Musch et al. [22] discuss in their work the prevalence of WebAssembly in the web and deal with malicious intentions. For this purpose they have analyzed the *Alexa Top 1 Million* [1]. They have identified that up to 1 of 600 sites run WebAssembly code. Related to this, they have detected that over 50 % use WebAssembly for malicious intentions, such as *mining* and *obfuscation*.

6 Conclusion

The relevant browser manufacturers have developed a joint approach with WebAssembly that does justice to its goals. WebAssembly combines the advantages of the older approaches of the individual browser manufacturers and thus eliminates the disadvantages. The ActiveX security problem has been improved by the Sandbox. The missing platform independence with PNaCl has been resolved. The worse performance with asm.js via JavaScript was fixed by the new language. It is to be seen as a result of the cooperation and therefore refers to all common web browsers.

In summary, an approach was developed that can execute code in a secure, platform-independent, operating system-independent and language-independent environment with high performance, thus scratching the performance level of native code more and more. Evaluations show that the performance has not yet reached the end. Compared to asm.js, WebAssembly achieved results that are 1.3× faster. Compared to native code, WebAssembly can't shine yet, but is constantly improving. It has to be considered that the project is only four years old and still has a lot of potential. Statistics show that WebAssembly is more and more used. Thus, the web is now also prepared for demanding web applications.

[3]Browsix is a framework that closes the interface between operating systems and the browser and allows unmodified programs that expect a Unix-like environment to run directly in the browser. [23]

[4]The SPEC CPU 2017 benchmark package contains SPEC's next-generation, industry-standardized, CPU intensive suites for measuring and comparing compute intensive performance, stressing a system's processor, memory subsystem and compiler. [11]

References

[1] Anon. [n.d.]. Alexa. https://www.alexa.com. Accessed: 2019-11-28.
[2] Anon. [n.d.]. The AssemblyScript Book. https://docs.assemblyscript.org. Accessed: 2019-11-07.
[3] Anon. [n.d.]. Concepts. https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts. Accessed: 2019-11-02.
[4] Anon. [n.d.]. Lucet, a native WebAssembly compiler and runtime. https://github.com/fastly/lucet/wiki. Accessed: 2019-11-07.
[5] Anon. [n.d.]. polybench. https://sourceforge.net/p/polybench/wiki/Home/. Accessed: 2019-11-05.
[6] Anon. [n.d.]. A small and efficient runtime for WebAssembly & WASI. https://wasmtime.dev.
[7] Anon. [n.d.]. The State of the OCTOVERSE. https://octoverse.github.com. Accessed: 2019-11-16.
[8] Anon. [n.d.]. Usage statistics of Flash as client-side programming language on websites. https://w3techs.com/technologies/details/cp-flash/all/all. Accessed: 2019-11-07.
[9] Anon. [n.d.]. Welcome to Native Client. https://developer.chrome.com/native-client. Accessed: 2019-11-15.
[10] Lin Clark. 2019. Standardizing WASI: A system interface to run WebAssembly outside the web. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, note = Accessed: 2019-11-07.
[11] Standard Performance Evaluation Corporation. [n.d.]. SPEC CPU 2017. https://www.spec.org/cpu2017/. Accessed: 2019-11-15.
[12] Jan de Mooij. [n.d.]. The Baseline Interpreter: a faster JS interpreter in Firefox 70. https://hacks.mozilla.org/2019/08/the-baseline-interpreter-a-faster-js-interpreter-in-firefox-70/. Accessed: 2019-11-27.

[13] Daniel Ehrenberg. 2019. WebAssembly JavaScript Interface. https://www.w3.org/TR/wasm-js-api-1/. Accessed: 2019-11-07.

[14] W3C WebAssembly Community Group. [n.d.]. Introduction. https://webassembly.github.io/spec/core/intro/introduction.html. Accessed: 2019-11-01.

[15] W3C WebAssembly Community Group. [n.d.]. WebAssembly. https://webassembly.org. Accessed: 2019-10-31.

[16] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web Up to Speed with WebAssembly. *SIGPLAN Not.* 52, 6 (June 2017), 185–200. https://doi.org/10.1145/3140587.3062363

[17] David Herman, Luke Wagner, and Alon Zakai. 2014. asm.js Working Draft. http://asmjs.org/spec/latest/. Accessed: 2019-10-31.

[18] Martin Holland. [n.d.]. Adobe verabschiedet sich von Flash: 2020 ist Schluss. https://www.heise.de/newsticker/meldung/Adobe-verabschiedet-sich-von-Flash-2020-ist-Schluss-3783264.html. Accessed: 2019-11-07.

[19] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 107–120. https://www.usenix.org/conference/atc19/presentation/jangda

[20] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[21] Microsoft. 1996. Microsoft Announces ActiveX Technologies. https://news.microsoft.com/1996/03/12/microsoft-announces-activex-technologies/. Accessed: 2019-10-31.

[22] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. *New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild*. 23–42. https://doi.org/10.1007/978-3-030-22038-9_2

[23] Bobby Powers, John Vilk, and Emery D. Berger. 2017. Browsix: Bridging the Gap Between Unix and the Browser. *SIGOPS Oper. Syst. Rev.* 51, 2 (April 2017), 253–266. https://doi.org/10.1145/3093315.3037727

[24] Konrad Rieck, Tammo Krueger, and Andreas Dewald. 2010. Cujo: Efficient Detection and Prevention of Drive-by-download Attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10)*. ACM, New York, NY, USA, 31–39. https://doi.org/10.1145/1920261.1920267

[25] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IN PROCEEDINGS OF THE 2007 IEEE SYMPOSIUM ON SECURITY AND PRIVACY*.