New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild

Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck

TU Braunschweig, Germany

Abstract. WebAssembly, or *Wasm* for short, is a new, low-level language that allows for near-native execution performance and is supported by all major browsers as of today. In comparison to JavaScript it offers faster transmission, parsing, and execution times. Up until now it has, however, been largely unclear what WebAssembly is used for in the wild. In this paper, we thus conduct the first large-scale study on the Web. For this, we examine the prevalence of WebAssembly in the Alexa Top 1 million websites and find that as many as 1 out of 600 sites execute Wasm code. Moreover, we perform several secondary analyses, including an evaluation of code characteristics and the assessment of a Wasm module's field of application. Based on this, we find that over 50 % of all sites using WebAssembly apply it for malicious deeds, such as mining and obfuscation.

1 Introduction

For a long time, JavaScript has been the only option to create interactive applications in the browser and especially the development of CPU intensive applications, such as games, has been held back by the subpar performance offered by JavaScript. As a remedy, several attempts to bring the performance benefits of native code to the Web have thus been made: Adobe has heavily promoted the *Flash platform*, Microsoft proposed *ActiveX*, and comparatively recently, Google introduced its *Native Client*. However, all these are tied to a specific platform and/or browser and could not gain acceptance on a large scale. While Adobe Flash marks an exception here, it suffered from a number of critical vulnerabilities over the years [26, 40], resulting in dwindling acceptance. By now all these variants have been deprecated [7, 24] or are scheduled for an imminent end [2].

In March 2017 the first version of WebAssembly has been published as a standardized and platform-independent alternative. Only months later, it has been implemented in all four major browser engines [22] and strongly gained traction ever since, as the low-level bytecode language allows for significantly faster transmission, parsing, and execution in comparison to JavaScript [8]. Naturally, also adversaries have quickly taken up the trend and have used WebAssembly to mine memory-bound cryptocurrencies, such as Monero. Consequently, hijacking websites to covertly perform intensive computations has become a worthwhile alternative to dedicated mining rigs. This new form of parasitic computing,

widely called cryptojacking or drive-by mining, has gained momentum on the Web during the all-time high of cryptocurrencies at the end of 2017. Research has recently investigated this phenomenon in particular and has proposed several countermeasures [14, 19].

Apart from this very specific occurrence, however, it remains unclear what WebAssembly is widely used for and whether malware based on this new technology exists next to cryptojacking miners. In this paper, we thus conduct the first comprehensive and systematic investigation in this regard. To this end, we instrumented a browser to collect all WebAssembly code and use a profiler to gather information about the CPU usage of the visited sites. This way, we cannot only detect the *mere presence* of WebAssembly, but also measure the *extent of usage* compared to the time spent on executing JavaScript code. In particular, we find that 1 out of 600 sites among the Alexa 1 million use WebAssembly and one-third of these even spend more than 75 % of the time executing WebAssembly in comparison to time spent in JavaScript code.

Interestingly, we also observe a high amount of code shared between modules. This is in line with previous research on cryptominers and seems little surprising from this point of view. However, we have additionally attributed each collected sample to different categories in a manual effort and thus, identify several distinct use cases that have found widespread adoption so far. *Mining* is only one of them and is applicable for 32 % of all unique samples. The second category with potential malicious intent is the broad field of *Obfuscation*, which amounts for 6 % of the unique samples. This category consists of 10 different samples that obfuscate program code using WebAssembly, such that we conclude that this is only the tip of the iceberg of what we will see in the future. The remaining 64 % is spread over games, custom code, libraries, and environment fingerprinting techniques. In combination, the two malicious categories, however, account for more than half of the web sites that make use of WebAssembly, due to the reuse of the same mining modules on many different sites.

Based on these findings, we further discuss security implications resulting from the introduction of this low-level language into the browser. In summary, our paper makes the following contributions:

- **Large-scale Analysis.** Based on the Alexa Top 1 million websites ranking, we present the first large-scale study on the usage of WebAssembly in the wild.
- **Categorization.** We provide a mapping from the collected samples to concrete use cases and investigate how many of these have malicious intentions.
- **Security Assessment.** We discuss the expected security implications enabled through the introduction of another first-class language into the browser.

The remainder of the paper is structured as follows: In Section 2 we discuss fundamentals and peculiarities of WebAssembly as a platform, before we conduct our study on its prevalence in the wild in Section 3. In Section 4 we then inspect the individual applications of WebAssembly. Based on this, we discuss implications for the security on the Web as well as related work in Sections 5 and 6, respectively. Section 7 concludes the paper.

2 WebAssembly

Often, JavaScript is not sufficient to create efficient implementations, as it requires a costly interpretation within the browser. As a remedy, the WebAssembly standard proposes a low-level bytecode language, that is a portable target for the compilation of high-level languages, such as C/C++ and Rust [30]. The resulting WebAssembly code, or *Wasm* code for short, is then executed on a stack-based virtual machine. Due to its low-level nature, the code can execute at near-native speed, but still runs in a memory-safe, sandboxed environment in the browser and is subject to security mechanisms like the same-origin policy [37]. Moreover, WebAssembly significantly improves on loading time over JavaScript code, due to its space-efficient binary representation [13].

Similar in motivation, but orthogonal to WebAssembly, *asm.js* was introduced in 2013 and implements a subset of JavaScript that is specifically designed to enable fast execution of code on the Web [4]. Some browsers then started to add optimizations specifically for asm.js resulting in further performance gains. However, JavaScript and asm.js are both missing important features for performance critical applications like 64-bit integers operations, threads, and shared memory. Furthermore, while asm.js features comparable fast execution, its parsing performance still is inferior compared to WebAssembly [42].

WebAssembly addresses these shortcomings and significantly improves on the state of the art of efficient computation on the Web. Hence, it quickly gained popularity due to well-advanced dissemination of its implementation. As of the time of writing, WebAssembly has global support of approximately 80 %[1] of all users (including mobile devices). In this section, we give an overview of the WebAssembly format and the corresponding JavaScript API that is used to instantiate and interact with Wasm modules.

2.1 Module structure

WebAssembly is structured in modules, which are self-contained files that may be distributed, instantiated, and executed individually. Each module starts with the magic bytes `0x6d736100`, a null byte and the string `asm`, followed by the version number, which currently is fixed to `0x01`. A WebAssembly module consists of individual sections of 11 different types, such as `code`, `data`, and `import`. Subsequently, we briefly describe the four most relevant section types.

Code Section This usually is the largest section as it encapsulates all function bodies of the WebAssembly module. One example of a function in this code section can be found in Table 1. This example comprises only basic functionality, like control flow statements, addition, and multiplication. Of course, there also exist a number of instructions with more complex mechanics, such as `popcnt`, which counts the number of bits set in a number, or `sqrt`, which calculates the square root of a floating-point value [37]. Note that these instructions do not

[1] Statistics from `http://caniuse.com` in January 2019

involve any registers, but operate on the stack only, which is based in the design of the underlying virtual machine.

Table 1. A simple C function on the left and the corresponding WebAssembly bytecode along with its textual representation, the *Wat* format, on the right-hand side [37].

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

Data Section The optional data section is used to initialize memory, similar to the `.data` section of x86 executables. Amongst others, this section contains all strings used throughout the module. Figure 1 shows the definition of such a data segment in Wat format. In this example, four bytes are saved at the memory offset 8, which results in the number 42 if read as an unsigned integer. The memory index in line 2 references an instance of linear memory and is reserved for future use, as currently only one memory definition is supported.

```
1 (data_segment
2   0                          // memory index
3   (init_expr (i32.const 8))  // byte offset
4   (data 0x2a 0x0 0x0 0x0)    // the data itself
5 )
```

Fig. 1. Initializing memory with the number 42.

Import and Export These sections define the imports and exports of a WebAssembly module. The import section consists of a sequence of imports, which are made available to the module upon instantiation, that is, any listed function can then be used via the `call` opcode. For importing a function the module name,

function name, and its type signature needs to be specified. It is then up to the host environment, for instance the browser, to resolve these dependencies and check that the function has the requested signature [37]. The export section, in turn, defines which parts of the module (functions or memory) are made available to the environment and other modules. Everything declared in this section can also be accessed via JavaScript as well.

2.2 JavaScript API

WebAssembly is intended to complement JavaScript, rather than to replace it. Consequently, it comes with a comprehensive JavaScript API, that allows sharing functionality between both worlds and instantiating WebAssembly modules with only a few lines of JavaScript code. To speed up the initialization of a new module, the Wasm binary format is designed such that a module can be partially compiled while the download of the module itself is still in progress. To this end, the API provides the asynchronous `instantiateStreaming` function, to complement the older synchronous `instantiate` function. Of course, WebAssembly modules may also be instantiated from data, embedded in JavaScript code—for instance, as raw bytes in an `Uint8Array`.

```
1 const obj = {
2     imports: {
3         imported_func: function(arg) { console.log(arg); }
4     }
5 };
6 const wasm = await WebAssembly.instantiateStreaming(
7     fetch('example.wasm'), obj
8 );
9 let result = wasm.instance.exports.factorial(13);
```

Fig. 2. Instantiating a WebAssembly module and calling an exported function.

One short example on how to instantiate a Wasm module and interact with it in only a few lines of code can be seen in Figure 2. In this example, the first parameter of the `instantiateStreaming` call in line 7 uses the `fetch` function to load a module over the network. The second parameter is an object specifying the imports for the module and, in this example, exposes the `console.log` function to the WebAssembly environment. This is necessary to grant the Wasm code access to functions from the JavaScript domain. The same restrictions, for instance, also apply to access and modifications to the DOM. In line 9, the exported function `factorial` is invoked to pass execution to the WebAssembly module. The corresponding Wasm code is the called, executed, and the value returned to the JavaScript environment. Alternatively, the `factorial` function could be changed to make use of the imported `console.log` functionality to directly print the value from within the Wasm module.

3 WebAssembly in the Wild

In order to comprehensively assess the WebAssembly landscape, we first need to obtain a representative data set on its current usage. In the following, we describe our crawling setup to collect Wasm samples in the wild. We then report on its prevalence in general, analyze the extent of usage in more detail, and also investigate the similarity between our collected samples.

3.1 Data collection

We conduct our study on the Alexa list of the Top 1 Million most popular sites[2]. Throughout this paper, a *site* refers to one entry in the Alexa list and consists of the *pages* that share the same origin with it. We conducted a preliminary study that revealed that a significant fraction of the Wasm code is not loaded when only visiting the front page of each domain. Therefore, and in line with previous research [19], we also randomly select three links from the front page to subpages within that site and visit these, too. This way, we could identify about 25% more sites that use WebAssembly and collected about 40% more unique samples compared to a crawl of the same sites without any subpages. While visiting three links still only approximates the actual usage in the wild, this setup represents a tradeoff between completeness and feasibility of the study. On each page, we wait until the browser fires the load event or a maximum of 30 seconds pass. On top of that, we always wait an additional 5 seconds on each page after the load event, to allow for dynamically loaded content in general and the dynamic loading of larger Wasm files in particular.

3.2 Implementation

To collect the Wasm modules, we visit all these pages with our crawler written in NodeJS. Under the hood, the crawler uses *Google Chrome v73.0.3679* as its browser, which we instrument via the *DevTools Protocol* [6]. Previous research mentions the undocumented `-dump-wasm-module` flag for Chrome's Wasm compiler as a simple option to save all executed Wasm modules [19]. However, this flag seems to be no longer functional in recent versions of Chrome. Another option to collect all Wasm code is hooking the DevTools Protocol's `Debugger.scriptParsed` event and filtering for the `wasm://` scheme. Though this event only gives us the Wat for each parsed function and not the whole Wasm module in its original form and hence important parts of the module, like the memory section, are not available.

For comprehensiveness, we instead transparently hook the creation of all JavaScript functions which can compile or instantiate Wasm modules. Figure 3 demonstrates how we hooked `instantiate` and the corresponding async version called `instantiateStreaming`. For `compile` and its async counterpart, the process is identical. For the `WebAssembly.Module` constructor, on the other hand, we use the built-in `Proxy` object to create a trap for the `new` operator [see 23].

[2] `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip` (from 21. December 2018)

```
let original = WebAssembly.instantiate;
WebAssembly.instantiate = function(bufferSource) {
    //Log bufferSource to backend here
    return original.call(WebAssembly, ...arguments);
};
WebAssembly.instantiateStreaming = async function(source, obj) {
    let response = await source;
    let body = await response.arrayBuffer();
    return WebAssembly.instantiate(body, obj);
};
```

Fig. 3. Modifying the instantiation of WebAssembly to collect the raw Wasm bytes.

3.3 Prevalence

With this setup, we visited the Alexa Top 1 million over a time span of 4 days. Of the initial 1 million, 52,296 front pages could not be successfully visited: About 63% of these failed because the DNS did not resolve and another 27% did not load within our timeout of 30 seconds. The rest failed due to various reasons, such as too many redirects or SSL errors. In the following, we report on the usage of WebAssembly on the remaining 947,704 sites in an aggregated fashion, i.e. if we find the same Wasm binary on different subpages belonging to the same site, we only report it once. In total, we visited 3,465,320 pages, as some front pages had less than three internal links.

Overall, we discovered 1,639 sites loading 1950 Wasm modules, of which 150 are unique samples. This means that some Wasm modules are popular enough to be found on many different sites, in one case the exact same module was present on 346 different sites. On the other hand, 87 samples are completely unique and were found only on one site, which indicates that many modules are a custom development for one website. On some pages, we found up to 10 different Wasm modules, with an average of 1.22 modules per page on sites that use WebAssembly at least once. Moreover, Figure 4 shows that sites with a lower Alexa rank[3] also tend to use WebAssembly more often.

Regarding the initiator of the modules, on 1118 sites the module was instantiated by a first-party script, while on 795 sites the module came from a third-party script or iframe with another origin. In the second case, the site's administrator might not even be aware that WebAssembly is used on his/her site. Note that there is some overlap, as some sites used multiple modules, especially since we also crawled several subpages for each site. Of these, on 1349 sites, the majority of instantiations happened inside a dedicated *WebWorker*. Code in such a WebWorker executes in a separate thread and does not block interaction with the page during intensive computations. Any JavaScript code that runs in a worker is likely computationally intensive and thus an ideal target to implement in WebAssembly instead.

[3] A lower rank means a more popular site, e.g. `google.com` has rank 1

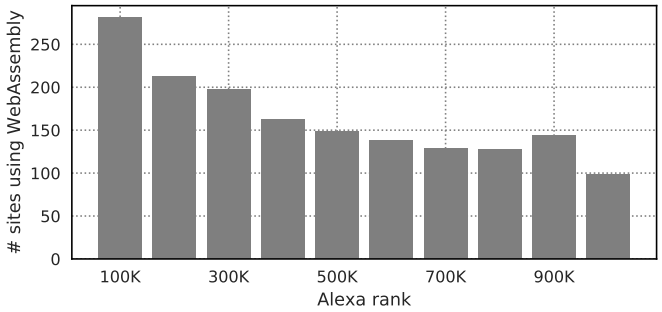


Fig. 4. Distribution of Alexa sites using WebAssembly in bins of 100,000 sites.

3.4 Extent of usage

The mere instantiation of a Wasm module, however, does not mean that a site is actively using the module's code. Some sites just test if the browser does support WebAssembly, while other sites are actually relying on the functionality the module exposes. A first indicator for this is the size of the module: the smallest was only 8 bytes, while the largest was 25.3 MB with a median value of 99.7 KB per module, as can be seen in Figure 5. On the other hand, the JavaScript code on the sites using Wasm had a median size of 2.79 MB. This shows that currently the amount of Wasm code is often only a fraction of the total code base on a site. Nevertheless, this should be seen only as a rough estimate as the comparison between a text-based and a binary format is inherently unfair.

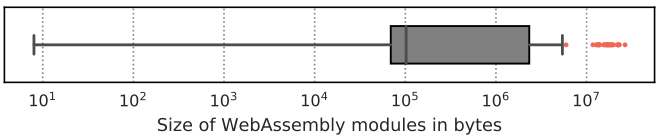


Fig. 5. Distribution of the size of WebAssembly modules. The box represents the middle 50% of the data, with the line in the box at the median. Outliers are shown as red dots.

To conduct a more in-depth analysis of how many sites make any significant use of the loaded Wasm modules, we use Chrome's integrated performance profiler. This profiler pauses the execution at a regular interval and samples the call stack, which enables us to estimate how much time is spent executing JavaScript code and how much time is spent in Wasm code. For this analysis, we revisited all pages on which we found a Wasm module with the profiler active both during the page load and for an additional 5 seconds after the page had finished loading.

With this approach, we find that 1121 sites spent more time on executing JavaScript code, while 506 spent more time in Wasm code. For the remaining

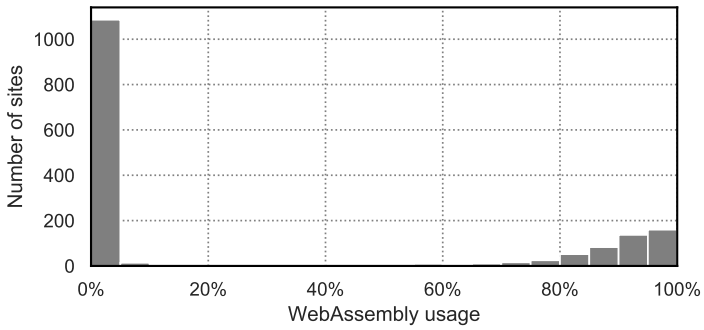


Fig. 6. Distribution of the execution time spent in Wasm code. The sites are heavily biased towards either extreme, with no middle ground.

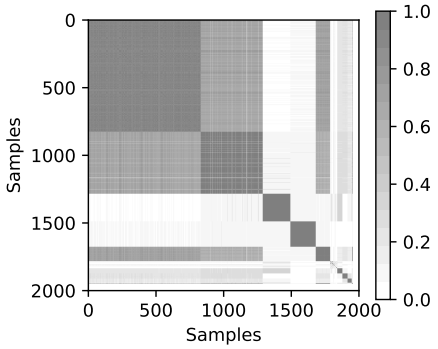
12 sites we have no profiling data, which was caused by the fact that we did the profiling analysis in a second crawl shortly afterward and some sites became unavailable in the meantime. As Figure 6 shows, there is a distinct contrast between the majority of sites that use WebAssembly almost not at all and some other sites that nearly exclusively spending time in the Wasm code. For this analysis, we measure the execution time of Wasm and JavaScript code and exclude all other factors like idle times when waiting for network responses. For example, a site with a Wasm usage of 90% thus spent the remaining 10% executing JavaScript code.

3.5 Code similarity

Finally, we analyze the diversity of the WebAssembly code that we have gathered. While we observe a rather large amount of identical code samples (only 10 % are unique instances), determining the similarity of WebAssembly code requires a fuzzy analysis, as minor perturbations of the files obstruct the application of exact matching.

We thus employ techniques from information retrieval that can cope with noisy data. In particular, we conduct an n-gram analysis, where the bytecode is first partitioned into byte sequences of length n and then mapped to a vector space by counting the occurrences of n-grams [see 32]. This vectorial representation enables us to compute the *cosine similarity* between all samples and generate the similarity matrices shown in Figure 7a. The columns and rows of the matrices are arranged using hierarchical clustering, such that larger groups of similar code samples become visible.

In total, we identify 11 clusters of similar code. Particularly eye-catching are the two largest groups at the top, left corner of the figure. First, with 831 and 456 samples they comprise more than half of the overall WebAssembly code which we have collected. Second, the distinction between both is not as clear as to other clusters in our dataset. This indicates a certain similarity wrt. to the used



(a) Cluster distances

Cluster	# samples
1	831
2	456
3	204
4	186
5	107
6–11	166
Total	1,950

(b) Cluster sizes

Fig. 7. Similarity analysis of WebAssembly code from the Alexa 1 million ranking.

instructions. Likely, samples from both groups make use of heavy computations such as rendering graphics or computing large numbers. Clusters 2 and 3 are smaller (~200 samples) but more clearly separated from the remaining samples. For cluster 5, note the (dark gray) horizontal and vertical bars. These indicated that this group again shares similarity with the first two clusters. Presumably, these samples represent an alternative implementation of the same task or the same implementation compiled with different tools, leading to minor differences on the byte level without affecting the code's functionality. The remaining clusters are rather small and expose only little structure.

In the following section, we further investigate the use cases and the origin of the samples represented in these clusters and thus attempt to shed light on the overall use of WebAssembly in the wild.

4 Applications of WebAssembly

In the previous section, our dynamic analysis has shown significant different usage, with some pages spending more time in Wasm code than in JS, while many others barely used the Wasm module at all. Additionally, our clustering has shown that many modules are part of a larger group of similar modules. These analyses both indicate that there a few very different use cases for which WebAssembly is used today.

To investigate this further, we manually analyzed all 150 collected Wasm modules. We first inspected the modules themselves and looked at function names and embedded strings to get an idea about the likely purpose of the module. In order to confirm our assumptions, we visited one or more websites that used the module and investigated where the Wasm module is loaded and how it interacts with the site. Thereby, we arrived at the following six categories: Custom, Game, Library, Mining, Obfuscation, and Test. The first three are of benign nature,

Table 2. The prevalence of each use case in the Alexa Top 1 Mio. As some sites had modules from multiple categories, the sum exceeds 100%.

Category	# of unique samples		# of websites		Malicious
Custom	17	(11.3 %)	14	(0.9 %)	
Game	44	(29.3 %)	58	(3.5 %)	
Library	25	(16.7 %)	636	(38.8 %)	
Mining	48	(32.0 %)	913	(55.7 %)	✗
Obfuscation	10	(6.7 %)	4	(0.2 %)	✗
Test	2	(1.3 %)	244	(14.9 %)	
Unknown	4	(2.7 %)	5	(0.3 %)	
Total	150	(100.0 %)	1,639	(100.0 %)	

but modules of the categories *Mining* and *Obfuscation* use WebAssembly with malicious intentions. We consider testing for WebAssembily support in general as neither benign nor malicious itself and thus see the category *Test* as neutral.

4.1 Results

The largest observed category implements a cryptocurrency miner in WebAssembly, for which we found 48 unique samples on 913 sites in the Alexa Top 1 Million. With 44 samples we found almost as much different games using Wasm, but in contrast to the miners, these games are spread over only 58 sites and thus often only appeared once. For 4 modules we could not determine their purpose and labeled them as *Unknown*. Of these, 2 did not contain a valid Wasm module, but the sites attempted to load it as such regardless. Table 2 summarizes our results and shows that with 56%, the majority of all WebAssembly usage in the Alexa Top 1 Million is for malicious purposes. The remainder of this section first shortly describes the benign and neutral use cases and then proceeds to look into the malicious usages of WebAssembly in more detail.

4.2 Benign: Custom, Games and Libraries

Modules in the *Custom* category appeared to be a one-of-a-kind experiment, e.g. one was a fancy background animation and another collected module contained an attempt to write a site mostly in C# with cross-compilation to Wasm. *Games* are arguably also of custom nature and often only found on one specific site. However, they are a very specific subset of custom code, of more professional nature and often also have a clear business model (e.g. in-game purchases or advertisements). *Library*, on the other hand, describes Wasm modules that are part of publicly available JavaScript libraries. For example, the library *Hyphenopoly*[4] uses WebAssembly under the hood to speed up its algorithm for word hyphenation. In this case, the use of WebAssembly might not be the result of an active decision by the site's developer.

[4] https://github.com/mnater/Hyphenopoly/issues/13

4.3 Neutral: Test

As described in Section 3.3, some sites loaded WebAssembly modules with a size of only a few bytes. Manual investigation showed that the only purpose of these modules is to test whether WebAssembly is supported by the visitor's browser. We discovered such test modules on 244 sites. Of these, 231 sites then proceeded to load another module after this test, while, on the other hand, 13 sites only made the test without executing any further Wasm code afterward. The latter might, for example, use this in an attempt to fingerprint their visitors. However, due to the lack of information we gain from such a small module, we see these as neither benign nor malicious.

4.4 Malicious: Mining

The category *Mining* includes all modules that are used to mine for cryptocurrencies in the browser. Mining is a basic building block of cryptocurrencies, such as Bitcoin or Ether, and refers to the process of solving computational puzzles to validate transactions and generating new coins of the currency [25]. Newer, *memory-bound* currencies build on computational puzzles that are memory intensive and thereby reduce the advantage of specific hardware over commodity processors [33, 36]. Consequently, the resulting currencies can be profitably mined on regular computer systems and thus open the door for the widespread application of cryptocurrency mining. Unfortunately, this development has also attracted miscreants who have discovered cryptocurrencies as a new means to generate profit. By tricking users into unnoticeably running a miner on their computers, they can utilize the available resources for generating revenue—a strategy denoted as *cryptojacking* or drive-by mining [19].

A novel realization of this strategy is injecting mining code into a website, such that the browser of the victim mines during the website's visit. First variants of these attacks have emerged with the availability of the CoinHive miner in September 2017 [1, 20]. This software cleverly combines recent web technologies to implement a miner that efficiently operates on all major browsers. In particular, WebAssembly is a perfect match for implementing mining software, as it enables compiling cryptographic primitives, such as specific hash functions, from a high-level programming language to low-level code for a browser. Since CoinHive's launch, several similar variants that all implement the underlying CryptoNote protocol have been developed, including sites like JSECoin and CryptoLoot.

Our analyses of the miner samples show that they exhibit several unique traits when compared to modules from all other categories. For one, their size is rather uniform, ranging from 23.4 KB to 119.1 KB with a standard deviation of only 16.4 KB. In contrast, for all the other categories combined we observed a standard deviation of 7.74 MB. Moreover, the code of the collected WebAssembly miners is also rather similar. Coming back to the code similarity analysis, the clusters number 3 and 6 in Figure 7a consist entirely out of miners. Except for 3 outliers in the very small cluster 7, these two clusters also contain all the mining

samples from our dataset. Moreover, we also found that on 26 % of all sites with a miner, a significant fraction of these was instantiated by third-party code.

As expected, sites with Wasm miners also use these modules much more extensively than modules from any other category. Coming back to our measurement of the JS and Wasm CPU usage with the profiler described in Section 3.4, we can now (after the manual categorization of the samples) analyze how many of the sites with a high amount of Wasm usage are running a miner. As shown in Figure 8, of the 506 sites with over 50% WebAssebmly usage with 497 sites the vast majority of these is, indeed, mining for cryptocurrencies. Regarding the 9 sites with high CPU usage from Wasm code not related to web-based mining, we found that 8 were caused by libraries for fast 64 bit operations or video streaming, and 1 by a game. On the other hand, this means that 416 sites do not mine for cryptocurrencies, despite the presence of a mining module. Our manual investigation has shown several reasons why a miner might be present but not active: (1) A mining script is included, but the miner is not started or was disabled and the script not removed. (2) The miner only starts once the user interacts with the web page or after a certain delay. (3) The miner is broken, either because of invalid modifications or because the remote API has changed. (4) The WebSocket backend is not responding, which prevents the miner from running.

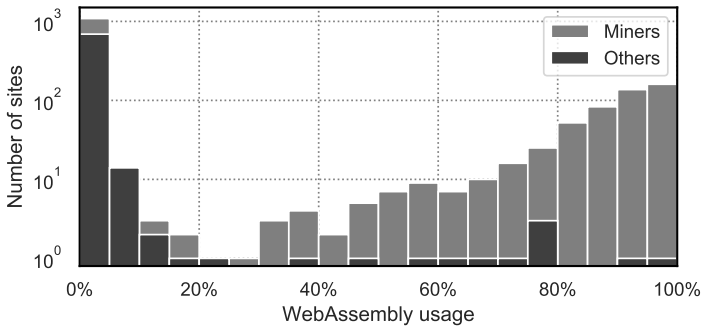


Fig. 8. Distribution of the execution time spent in Wasm code by category. Note that this is based on the same data as Figure 6, but this time with the y-axis is on a log scale for visibility of the smaller values. As a result of the log scale, the inactive miners on the very left appear smaller than they actually are.

4.5 Malicious: Obfuscation

While cryptojacking certainly did get a lot of attention from the academic community in 2018 [e.g. 14, 19, 29, 31], this is not the only type of malicious WebAssembly code already in use in the wild. Rather than using WebAssembly

for its performance improvements, some actors abuse its novelty instead. Figure 9 shows the HTML and JavaScript code embedded into the memory section of a Wasm module we found. Through this obfuscation of hiding the JavaScript code in a Wasm module, malicious actors likely can prevent detection by analysis tools that only support JavaScript and do not understand the Wasm format. The code tries to create a pop-under, which is an advertisement that spawns a new window behind the current one and is basically the opposite of a pop-up. The idea is that this way, the window stays open for a very long time in the background until the user minimizes or closes the active browser window in the foreground. Another 8 modules also contained code related to popups and tracking in the memory section, likely in an attempt to circumvent adblockers.

The last of the 10 modules, which employed obfuscations via WebAssembly, implemented a simple XOR decryption (and nothing else). This could, for example, be used to decrypt the rest of a malicious payload. However, in this case, the module seemed to not be used at all after the initialization. Nevertheless, we see these first, simple examples as evidence that malicious actors are already slowly moving towards WebAssembly for their misdeeds and we expect more sophisticated malware incorporating Wasm code to emerge in the future.

```
<script>
    var popunder = {expire: 12,
    url: '//hook-ups-here.com/?u=813pd0x&o=4gwkpzn&t=all'};
</script>
<script src='//hook-ups-here.com/js/popunder.js'></script>
```

Fig. 9. Code to create a pop-under advertisement, which was found in the memory section of a WebAssembly module.

5 The future of malicious WebAssembly

In the previous section, we showed that WebAssembly is actively used for malicious purposes. However, cryptocurrency miners would also have been possible without the introduction of WebAssembly, the mining would just have been less efficient and thus also less profitable. In fact, the popular CoinHive mining script contains a fallback to asm.js, in case WebAssembly is not supported by the visitor's browser [9].

On the other hand, WebAssembly does open the door for completely new obfuscation techniques not possible before and also enables the circumvention of existing analysis systems and defensive mechanisms. For example, established approaches like Zozzle [11], Rozzle [18], JStill [41], JForce [17], and Dachshund [21] all predate the introduction of WebAssembly to the browser. As these systems rely on parsing and inspecting JavaScript code to detect attacks, they cannot defend against new attacks implemented in Wasm code. Moreover, many are specifically

designed to work with the peculiarities of JavaScript and the fundamental differences between the two languages would require a non-trivial amount of work to adapt these approaches to the stack-based virtual machine running the WebAssembly.

As attackers tend to take the path of least resistance, WebAssembly looks like a promising way to write stealthy malware for the browser in the years to come. The phases from the first, simple attempts to hide code towards sophisticated obfuscation via WebAssembly could roughly progress in the following way:

Embedded JavaScript code. As described in Section 4.5, one simple approach to hide JavaScript during the transmission is to embed it into a WebAssembly module. However, this is likely only effective again Adblockers and other filters that employ rules on the network level.

Loader in Wasm. To take it one step further, a small module with one WebAssembly function could unpack and decrypt the actual JavaScript payload. This approach effectively defeats all static analysis tools that only understand JavaScript code without much effort from the malware author. However, as with the previous approach, the code then later needs to be added to the DOM or invoked via `eval`. Therefore, dynamic analysis would not be affected.

Full implementation in Wasm. In contrast to the previous to approaches, a full implementation of the exploit in WebAssembly would defeat systems for the automated static as well as dynamic analysis of JavaScript code. Furthermore, such modules are also harder to manually analyze than JavaScript code, due to the low-level nature of WebAssembly code.

Fully intertwined code. Even more complicated to analyze and detect would be malware that is neither exclusively JavaScript nor WebAssembly at its core, but both at the same time. With constant switches between the two domains, even within functions, the malware would never exist as a whole on either side. Therefore, any analysis system would require support for both languages and needs to be able to keep track of all possible switches and interactions between the two domains.

6 Related work

To the best of our knowledge, at time of writing there have been no peer-reviewed publications on the security aspects of WebAssembly. There are, however, already some tools in the work by academics, e.g., a dynamic analysis framework for WebAssembly called *Wasabi* [39]. In the following, we discuss related work on malicious JavaScript in general and the detection of cryptocurrency miners in particular.

6.1 Malicious JavaScript

The analysis of JavaScript is complicated by its dynamic nature and because malicious samples are also often obfuscated. Therefore, there have been many academic works over the years trying to tackle this problem. To detect drive-by attacks, Cova et al. [10] created JSAND, which uses anomaly detection combined with an emulated execution to generate detection signatures. With a similar goal, Cujo by Rieck et al. [27] uses static and dynamic code features to learn malicious patterns, detects them on-the-fly via a web proxy and consecutively can prevent their execution. Zozzle, on the other hand, is a malware detector by Curtsinger et al. [11], which runs inside the browser and uses mostly static features from the AST together with a Bayes classifier. Xu et al. [41] also use mostly static analysis, but with the goal to actually revert previously applied obfuscation steps. In 2016, Stock et al. [34] presented their work on creating signatures for JavaScript drive-by exploit kits. They found that while the obfuscated code changes frequently, the underlying unpacked code evolves much more slowly, which aids the detection process.

Some solutions were also proposed to deal with evasive malware, which is not only obfuscated but actively tries to avoid detection. Kolbitsch et al. [18] created Rozzle, an approach to trigger environment-specific malware via JavaScript multi-execution. This way, they can observe malicious code paths without actually satisfying checks for browser or plugin versions. Improving on this, Kim et al. [17] presented their work on forced execution to reveal malicious behavior, with a focus on preventing crashes. To detect evolving malicious JavaScript samples, Kapravelos et al. [16] designed Revolver, which utilizes similarities in samples compared to older versions of the same malware. Their rationale is that malware authors react to detections by anti-virus software and iteratively mutate their code to regain their stealthiness.

6.2 Cryptocurrency Mining

Web-based cryptojacking is a novel attack strategy that received the attention of the research community in 2018, with several papers on the topic in the same year. The study by Eskandari et al. [12] was the first to provide a peek at the problem. However, the study is limited to vanilla CoinHive miners, and the underlying methodology is unsuited to detect alternative or obfuscated mining scripts. The work by Hong et al. [14] uses a set of fixed function names and a stack-based profiler to detect busy functions. In contrast, Konoth et al. [19] propose a novel detection based on the identification of cryptographic primitives inside the Wasm code. Similarly, Wang et al. [38] detect miners by observing bytecode instruction counts, while Rodriguez and Posegga [29], on the other hand, use API monitors and machine learning.

Unauthorized mining of cryptocurrencies, however, is not limited to web scenarios. For example, Huang et al. [15] present a study on malware families and botnets that use Bitcoin mining on compromised computers. Similarly,

Ali et al. [3] investigate botnets that mine alternative currencies, such as Dogecoin, due to the rising difficulty of profitably generating Bitcoins. To detect illegitimate mining activities, either through compromised machines or malicious users, Tahir et al. [35] propose *MineGuard*, a hypervisor-based tool that identifies mining operations through CPU and GPU monitoring.

From a more general point of view, cryptocurrency mining is a form of *parasitic computing*, a type of attack first proposed by Barabási et al. [5]. As an example of this attack, the authors present a sophisticated scheme that tricks network nodes into solving computational problems by engaging them in standard communication. Moreover, Rodriguez and Posegga [28] present an alternative method for abusing web technology that enables building a rogue storage network. Unlike cryptojacking, these attack scenarios are mainly of theoretical nature, and the authors do not provide evidence of any occurrence in the wild.

7 Conclusion

With this study, we provide the first comprehensive view on the use of WebAssembly in the wild. Although we are investigating a rather novel technology, our empirical investigation shows that an increasing number of websites already deploy functionality in the form of Wasm modules. In particular, we find that 1 out of 600 websites in the Alexa 1 million ranking use WebAssembly, one-third of which even spend significantly more time on it than executing JavaScript. The remaining two-thirds, in turn, perform very few computations only. We credit this imbalance to the current prevalence of web-based cryptomining using WebAssembly, which drains a considerable amount of energy every day.

Moreover, by introducing an entire new execution environment, WebAssembly also opens the door for novel obfuscation strategies. The existence of multiple languages, that interact with each other, renders effective malware analysis extremely difficult. This holds true for static, dynamic, as well as manual analysis likewise. While these obfuscation strategies can be carried out to an extreme by heavily intertwining WebAssembly with JavaScript code, in our recordings we have only seen rather moderate ways of obfuscating program code thus far.

This, however, suggests that we are currently only seeing the tip of the iceberg of a new generation of malware obfuscations on the Web. In consequence, incorporating the analysis of WebAssembly code hence is going to be of essence for effective future defense mechanisms.

Acknowledgments

The authors gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF) under the project VAMOS (FKZ 16KIS0534) and FIDI (FKZ 16KIS0786K), and funding from the state of Lower Saxony under the project Mobilise.

Bibliography

[1] AdGuard Research. Cryptocurrency mining affects over 500 million people. And they have no idea it is happening. Website `https://adguard.com/en/blog/crypto-mining-fever/`, Oct. 2017.

[2] Adobe Corporate Communications. Flash & the future of interactive content. `https://theblog.adobe.com/adobe-flash-update/`, 2017.

[3] S. T. Ali, D. Clarke, and P. McCorry. Bitcoin: Perils of an unregulated global p2p currency. In *Security Protocols XXIII*, pages 283–293. Springer, 2015.

[4] ASM.js. Frequently asked questions. Website `http://asmjs.org/faq.html`, Feb. 2019.

[5] A.-L. Barabási, V. W. Freeh, H. Jeong, and J. B. Brockman. Parasitic computing. *Nature*, 412:894–897, 2001.

[6] ChromeDevTools. Chrome DevTools Protocol Viewer. Website `https://chromedevtools.github.io/devtools-protocol/`, May 2018.

[7] Chromium Blog. Goodbye pnacl, hello webassembly! Website `https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html`, May 2017.

[8] L. Clark. What makes webassembly fast? Website `https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/`, Feb. 2017.

[9] CoinHive Documentation. JavaScript Miner. Website `https://coinhive.com/documentation/miner`, Feb. 2019.

[10] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proc. of the International World Wide Web Conference (WWW)*, 2010.

[11] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *Proc. of USENIX Security Symposium*, 2011.

[12] S. Eskandari, A. Leoutsarakos, T. Mursch, and J. Clark. A first look at browser-based cryptojacking. In *Proc. of IEEE Security and Privacy on the Blockchain Workshop*, 2018.

[13] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. In *Proc. of ACM SIGPLAN International Conference on Programming Languages Design and Implementation (PLDI)*, pages 185–200, 2017.

[14] G. Hong, Z. Yang, S. Yang, L. Zhang, Y. Nan, Z. Zhang, M. Yang, Y. Zhang, Z. Qian, and H. Duan. How you get shot in the back: A systematical study about cryptojacking in the real world. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, Oct. 2018.

[15] D. Y. Huang, H. Dharmdasani, S. Meiklejohn, V. Dave, C. Grier, D. McCoy, S. Savage, N. Weaver, A. C. Snoeren, and K. Levchenko. Botcoin: Mone-

tizing stolen cycles. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2014.

[16] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *Proc. of USENIX Security Symposium*, 2013.

[17] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu. J-force: Forced execution on javascript. In *Proc. of the International World Wide Web Conference (WWW)*, 2017.

[18] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proc. of IEEE Symposium on Security and Privacy*, 2012.

[19] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna. An in-depth look into drive-by mining and its defense. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, Oct. 2018.

[20] B. Krebs. Who and What Is Coinhive? Website `https://krebsonsecurity.com/2018/03/who-and-what-is-coinhive`, Mar. 2018.

[21] G. Maisuradze, M. Backes, and C. Rossow. Dachshund: digging for and securing against (non-) blinded constants in jit code. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2017.

[22] J. McConnell. Webassembly support now shipping in all major browsers. Website `https://blog.mozilla.org/blog/2017/11/13/webassembly-in-browsers/`, Nov. 2017.

[23] MDN Web Docs. Proxy. Website `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy`, Feb. 2019.

[24] Microsoft Windows Blogs. A break from the past, part 2: Saying goodbye to activex, vbscript, attachevent. Website `https://blogs.windows.com/msedgedev/2015/05/06/a-break-from-the-past-part-2-saying-goodbye-to-activex-vbscript-attachevent/`, May 2015.

[25] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, May 2009. URL `http://www.bitcoin.org/bitcoin.pdf`.

[26] S. Özkan. CVE Details. `http://www.cvedetails.com`.

[27] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2010.

[28] J. D. P. Rodriguez and J. Posegga. CSP & Co. Can Save Us from a Rogue Cross-Origin Storage Browser Network! But for How Long? In *Proc. of ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2018.

[29] J. D. P. Rodriguez and J. Posegga. Rapid: Resource and api-based detection against in-browser miners. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2018.

[30] A. Rossberg. Webassembly core specification. W3C First Public Working Draft, Feb. 2018. URL `https://www.w3.org/TR/2018/WD-wasm-core-1-20180215`.

[31] J. Rüth, T. Zimmermann, K. Wolsing, and O. Hohlfeld. Digging into browser-based crypto mining. In *Proc. of Internet Measurement Conference (IMC)*, 2018.

[32] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1986.

[33] "Seigen", M. Jameson, T. Nieminen, "Neocortex", and A. M. Juarez. Cryptonight hash function. CryptoNote Standard 008, Mar. 2008. URL `https://cryptonote.org/cns/cns008.txt`.

[34] B. Stock, B. Livshits, and B. Zorn. Kizzle: a signature compiler for detecting exploit kits. In *Proc. of Conference on Dependable Systems and Networks (DSN)*, 2016.

[35] R. Tahir, M. Huzaifa, A. Das, M. Ahmad, C. Gunter, F. Zaffar, M. Caesar, and N. Borisov. Mining on someone else's dime: Mitigating covert mining operations in clouds and enterprises. In *Proc. of International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 287–310, 2017.

[36] N. van Saberhagen. Cryptonote v2.0. Technical report, CryptoNote, Oct. 2013.

[37] W3C WebAssembly Community Group. Webassembly design documents. Website `https://webassembly.org`, Jan. 2019.

[38] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao. Seismic: Secure in-lined script monitors for interrupting cryptojacks. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, 2018.

[39] Wasabi. Dynamic Analyis Framework. Website `http://wasabi.software-lab.org`, Feb. 2019.

[40] C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck. Comprehensive analysis and detection of flash-based malware. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 101–121, July 2016.

[41] W. Xu, F. Zhang, and S. Zhu. Jstill: mostly static detection of obfuscated malicious javascript code. In *Proc. of ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[42] A. Zakai. Why webassembly is faster than asm.js. Website `https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/`, Mar. 2017.