An Empirical Study of Real-World WebAssembly Binaries

Security, Languages, Use Cases

Aaron Hilbig
aaron@hilbigpost.de
University of Stuttgart
Germany

Daniel Lehmann
mail@dlehmann.eu
University of Stuttgart
Germany

Michael Pradel
michael@binaervarianz.de
University of Stuttgart
Germany

ABSTRACT

WebAssembly has emerged as a low-level language for the web and beyond. Despite its popularity in different domains, little is known about WebAssembly binaries that occur in the wild. This paper presents a comprehensive empirical study of 8,461 unique WebAssembly binaries gathered from a wide range of sources, including source code repositories, package managers, and live websites. We study the security properties, source languages, and use cases of the binaries and how they influence the security of the WebAssembly ecosystem. Our findings update some previously held assumptions about real-world WebAssembly and highlight problems that call for future research. For example, we show that vulnerabilities that propagate from insecure source languages potentially affect a wide range of binaries (e.g., two thirds of the binaries are compiled from memory unsafe languages, such as C and C++) and that 21% of all binaries import potentially dangerous APIs from their host environment. We also show that cryptomining, which once accounted for the majority of all WebAssembly code, has been marginalized (less than 1% of all binaries found on the web) and gives way to a diverse set of use cases. Finally, 29% of all binaries on the web are minified, calling for techniques to decompile and reverse engineer WebAssembly. Overall, our results show that WebAssembly has left its infancy and is growing up into a language that powers a diverse ecosystem, with new challenges and opportunities for security researchers and practitioners. Besides these insights, we also share the dataset underlying our study, which is 58 times larger than the largest previously reported benchmark.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

ACM Reference Format:
Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3442381.3450138

1 INTRODUCTION

WebAssembly is a fast, compact, low-level byte code language originally intended for client-side execution in web browsers. It is widely supported and available in 93% of all global browser installations as of February 2021.[1] Beyond client-side web applications, WebAssembly is also running on Node.js and even stand-alone runtimes.

Despite its growing popularity, the WebAssembly ecosystem is severely understudied. To date, little is know about how the language is used, for what purposes, and how this affects the security of WebAssembly-based applications. In particular, we are interested in the following research questions:

RQ1: Source languages and tools. WebAssembly is a compilation target, and in principle any programming language can be compiled to it. What languages are actually compiled to WebAssembly, how much do they contribute to the overall population, and what tools are used to produce the binaries? Answering these questions is relevant for understanding the impact of issues that specific source languages may have and for guiding future work toward source languages and toolchains prevalent in practice.

RQ2: Vulnerabilities propagated from insecure source languages. Recent work has shown that memory vulnerabilities in insecure source languages, such as C and C++, may be exploited in WebAssembly binaries, sometimes even more easily than in native binaries [18]. How large is the attack surface offered by real-world WebAssembly binaries compiled from insecure languages, e.g., in terms of dangerous APIs these binaries import from JavaScript or in terms of vulnerable memory allocators they ship? Answering this question will increase our understanding of the threat posed by vulnerabilities compiled to the web.

RQ3: Cryptomining. Previous results show [24], and recent work assumes [15, 25, 34, 43], that WebAssembly is frequently used for cryptojacking, i.e., cryptomining performed in the browser of an unsuspecting client. Is cryptomining still an important threat today?

RQ4: Use cases. As a general purpose language, WebAssembly can serve many purposes in web applications and beyond. What are the typical use cases of WebAssembly? Given that the language is becoming more widely adopted, it is important to understand what its use cases are and how this affects the security of the web.

RQ5: Minification and names. The ability to understand WebAssembly binaries, e.g., for auditing third-party code or for reverse engineering malware, depends on whether binaries contain meaningful names for program elements, e.g., functions. Do real-world WebAssembly binaries contain meaningful names or are they obfuscated through minification?

Answering these and other questions requires a set of WebAssembly binaries that is (i) representative for how WebAssembly is used in

WWW '21, April 19–23, 2021, Ljubljana, Slovenia
© 2021 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC-BY 4.0 License.
ACM ISBN 978-1-4503-8312-7/21/04.
https://doi.org/10.1145/3442381.3450138

[1] https://caniuse.com/?search=WebAssembly

the wild and (ii) large enough to cover the diversity of real-world WebAssembly usage. Currently, no such set of binaries exists.

The closest existing work is by Musch et al. [24], who report on a study of WebAssembly usage in the top one million websites. While inspiring, their study falls short in two respects. First, it has been performed at a point in time when WebAssembly was still in its infancy, with usage biased to early adopters, e.g., cryptominers, and a single toolchain dominating the ecosystem. Since then, many changes have happened, including higher browser adoption, alternative compilers that have become available, the shutdown of Coinhive (a common cryptomining platform) [42], and the realization that vulnerabilities in insecure source languages can also be exploited in WebAssembly [18]. Second, the methodology proposed by Musch et al. [24] focuses only on binaries found on the web, and only on those that are executed when just visiting a website. By only looking into client-side web applications, WebAssembly on other platforms is disregarded, e.g., on Node.js, browser extensions, and stand-alone WebAssembly runtime engines.

This paper presents a comprehensive empirical study of real-world WebAssembly binaries. The core of our work is *WasmBench*, a diverse set of 8,461 unique binaries gathered from a variety of sources, including querying source code repositories and package managers, searching the HTTP Archive, and crawling the web. The binaries found through our methodology show that considering only a single one of these data sources would miss a significant fraction of the WebAssembly ecosystem. While we obviously cannot guarantee to cover all kinds of real-world WebAssembly usages, WasmBench provides not only a 58× larger benchmark, but also a more diverse set of WebAssembly binaries, than the largest previously reported benchmark [24].

We use WasmBench to address the above research questions through a combination of manual inspection, custom static analysis tools, and statistical analyses. Our findings include:

- Real-world WebAssembly binaries are compiled from a variety of source languages, including systems programming languages, such as C, C++, Rust, and Go, higher level languages, such as AssemblyScript (a variant of TypeScript), and some rather unexpected languages, such as COBOL and Kotlin.
- The majority of binaries is compiled from memory-unsafe languages, from which vulnerabilities may propagate into WebAssembly binaries [18].
- 65% of all binaries and 44% of all functions in them use the "unmanaged stack", a portion of linear memory that is unprotected by the virtual machine and that can be exploited by attackers.
- 21% of all binaries import potentially dangerous APIs from their host environment, e.g., the infamous `eval`, APIs to modify the DOM from JavaScript, or system call-like APIs to interact with the network and file system on platforms outside the browser. An attacker compromising a binary may abuse such APIs to trigger unexpected behavior.
- Contrary to earlier findings [24], cryptomining has dropped significantly in relevance, comprising only 1% of all binaries. Instead, we find applications with up to many millions of instructions that cover diverse use cases, including visualization, interactive shells for programming languages, media players, game engines, data compression, and natural language processing.
- 28.8% of all binaries on the web are minified, calling for future work on decompiling and reverse engineering WebAssembly, to ensure that security analysts can understand web applications despite the presence of low level components.

```
int increment(int x)
{
    return x + 1;
}
```

(a) C Source Code.

```
(func (param i32)
      (result i32)
   local.get 0
   i32.const 1
   i32.add
)
```

(b) WebAssembly text format.

```
[...] // header with
      // type info
20 00 // local.get 0
41 01 // i32.const 1
6a    // i32.add
0b    // end
```

(c) WebAssembly binary format.

Figure 1: Example of a function compiled to WebAssembly.

Overall, our findings show that WebAssembly is "growing up", which leads to a larger and much more diverse ecosystem than in its early days. From a security perspective, this diversity has several implications. First, the fact that there are now many legitimate applications, and proportionally much fewer malicious ones, shifts the focus from detecting *malicious* code to handling *vulnerable* code. Second, the large fraction of binaries that originate from "insecure" source languages, in particular C and C++, shows the risk that their problems, e.g., memory vulnerabilities, will now propagate to the web. Mitigations against such memory vulnerabilities [18] are becoming an important goal to keep the web safe. Third, the many different compilation toolchains and their variants, e.g., in terms of memory allocators compiled into the binaries, create a potentially large attack surface. Automated tools to analyze and improve the security of WebAssembly binaries are needed.

In summary, this paper contributes:

- The first comprehensive study of WebAssembly binaries gathered from multiple sources, including client-side web applications, package managers, and source code repositories;
- A combination of automated program analyses, manual inspection, and statistical analysis to answer research questions about the security, source languages, and use cases of WebAssembly;
- Empirical evidence and insights about security-related properties of real-world WebAssembly, some of which update earlier findings and many of which call for future work on mitigation techniques and analysis tools;
- By far the largest benchmark of WebAssembly binaries, which we make available as a basis for other studies and as a benchmark for future tools: https://github.com/sola-st/WasmBench .

2 BACKGROUND

File formats and modules. WebAssembly is a low-level bytecode language, executed on a stack-based virtual machine. Figure 1 shows a small code example in C, the corresponding code in WebAssembly's `.wat` text format, and the same code in WebAssembly's `.wasm`

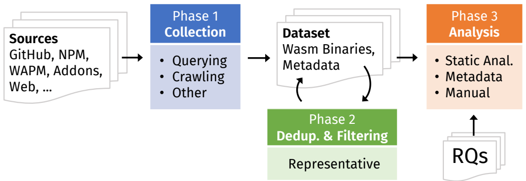


Figure 2: Overview of the phases of our methodology.

binary format. The latter is usually used to distribute WebAssembly. Instructions are arranged into functions, and functions are arranged into modules, which correspond to files. We use the terms "module" and "binary" interchangeably. Each module is divided into multiple sections, including `import` and `export` sections that declare functions shared with the environment, a `code` section that defines functions and their bodies, and a `data` section that initializes memory. Functions, types, and variables are referenced by indices.

Host environments. WebAssembly runs within a host environment, such as the browser, Node.js, or a standalone WebAssembly runtime like wasmer or wasmtime. The host environment instantiates the WebAssembly binary and can provide imported functions to the module. For example, in a client-side web application, JavaScript code can instantiate binaries through the `WebAssembly.instantiate` and `WebAssembly.Module` APIs.

Linear memory. Each module instance has a *linear memory* section, which can be thought of as an array of consecutive bytes. It stores all data handled by the module, except for locals and globals (which can only be of four primitive types), including all dynamically allocated data structures. The `i32` datatype is used for pointers into linear memory. The linear memory has two implications. First, programs that want to associate non-primitive data with functions typically do so through a so-called "unmanaged stack", which simply is a region in the linear memory used to represent the function stack. This stack is called "unmanaged" because it is not protected by the virtual machine and under full control of the program. As a result, memory-unsafe behavior from source languages, e.g., C and C++, can also affect WebAssembly binaries. Second, non-trivial applications often have their own memory allocator compiled into the binary. Both the unmanaged stack and custom memory allocators may allow attackers to exploit a WebAssembly binary [18].

Compilers, tools, and runtimes. Various compilers target WebAssembly, e.g., the Emscripten compiler for C and C++, the Rust compiler, the AssemblyScript compiler, or the Asterius compiler for Haskell. Several compilers sometimes share a common framework, e.g., Emscripten and the Rust compiler are based on LLVM, while the AssemblyScript compiler and Asterius are based on Binaryen. Because the bare WebAssembly language does not provide any built-in library, compilers often combine the compiled code with a runtime environment. For example, the Emscripten runtime environment and the WebAssembly system interface (WASI) both offer a set of low-level system calls, e.g., to support file I/O and networking.

3 METHODOLOGY

Our methodology is split into three phases (Figure 2). In the collection phase, we obtain a large set of WebAssembly binaries from a variety of sources. We select sources to cover WebAssembly in different contexts and at different stages of deployment. Sections 3.1 to 3.4 present how we collect WebAssembly binaries from source code repositories, package managers that distribute deployed software, archived and live websites, and through manual search, respectively. Figure 3 gives on overview of the different sources we collect binaries from. Alongside each binary, we also collect metadata, e.g., on which website a binary was found. All activities related to collecting binaries were done between April and September 2020. Overall, the collection phase results in 51,148 binaries, including duplicates and binaries that are not representative for real-world usages of WebAssembly. Section 3.5 presents the filtering phase, where we filter and deduplicate these binaries into a set of 8,461 unique binaries that serve as the basis for our study. Finally, the third phase analyzes the set of binaries through a combination of static code analysis, analysis of metadata associated with the binaries, and manual inspection. We present the analysis phase along with its results in Section 4.

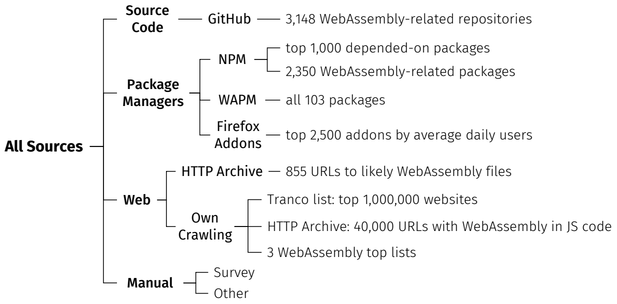

All Sources
Source Code
GitHub
3,148 WebAssembly-related repositories
Package Managers
NPM
top 1,000 depended-on packages
2,350 WebAssembly-related packages
WAPM
all 103 packages
Firefox Addons
top 2,500 addons by average daily users
Web
HTTP Archive
855 URLs to likely WebAssembly files
Own Crawling
Tranco list: top 1,000,000 websites
HTTP Archive: 40,000 URLs with WebAssembly in JS code
3 WebAssembly top lists
Manual
Survey
Other

Figure 3: Sources from which we collect binaries.

To the best of our knowledge, no prior work has gathered WebAssembly binaries from such a diverse set of sources. As a result, the number of binaries we obtain is 58 times larger than the largest set studied so far (147 unique binaries) [24]. Our experimental results (Section 4.2) show that the sources we consider WebAssembly binaries from complement each other, i.e., considering all of them is crucial to obtain a representative dataset.

3.1 Collecting Binaries from Repositories

Our first method for collecting binaries looks into source code repositories. Even though WebAssembly is a binary format, developers often store binaries into source code repositories, e.g., to ease the installation of a project or to include third-party libraries. To gather such binaries, we clone all public repositories that are in the top 1,000 results of four queries to the GitHub search API:

- Repositories where "wasm" or "WebAssembly" is in the repository name or description (i.e., two queries).
- Repositories that are tagged with "WebAssembly" as one of the used programming languages.
- Repositories tagged with the topic "WebAssembly".

Overall, the queries result in 3,148 repositories, which we clone, and then search for files ending in `.wasm`.

3.2 Collecting Binaries from Package Managers

Once developers deploy a WebAssembly-based application, it is often made available through a package manager. We consider three software ecosystems that use WebAssembly.

3.2.1 npm Packages. The Node Package Manager (npm) distributes JavaScript code, some of which relies on WebAssembly modules. Packages distributed via npm are typically used in server-side applications with Node.js or on the client side. To find npm packages that contain WebAssembly binaries, we gather two sets of packages. First, from the full registry file of npm we compute the top 1,000 most depended-upon packages. Second, we query npm for all packages that match at least one of the keywords "wasm" and "WebAssembly", which yields 2,350 packages. We install these packages and their transitive dependencies, and then search the resulting 7,198 packages for `.wasm` files.

3.2.2 wapm Packages. The WebAssembly Package Manager (wapm) specializes on distributing WebAssembly code. Most of the wapm packages are intended to run standalone WebAssembly runtime engines. Unlike for npm, we can afford to analyze all 103 available packages. We install all packages and again extract all `.wasm` files.

3.2.3 Firefox Browser Add-ons. Browser extensions, traditionally implemented in JavaScript, nowadays can also make use of WebAssembly code. To gather binaries used in browser extensions, we download the top 2,500 Firefox add-ons from addons.mozilla.org, as measured by average daily users. We then unpack the extensions' XPI archives, and search again for `.wasm` files.

3.3 Collecting Binaries from Websites

Collecting WebAssembly binaries from the web involves several challenges. First, as the web is too big to be searched in its entirety, finding suitable starting points for exploring it is crucial. Second, even when visiting a WebAssembly-powered website, it is nontrivial to identify and collect WebAssembly binaries from it. Some sites embed WebAssembly modules into JavaScript source code, e.g., as `base64`-encoded strings that are decoded and instantiated at runtime. For such sites, we must detect WebAssembly modules when they are executed. Other websites spawn requests for WebAssembly modules but never execute them during our collection process, e.g., because execution relies on specific user inputs. A purely dynamic methodology would miss such binaries.

We address the first challenge, finding good websites as starting points, through two techniques. On the one hand, we can build on results from the HTTP Archive for finding sites known to contain WebAssembly binaries (Section 3.3.1). On the other hand, for our own crawling, we systematically start from potentially WebAssembly-related seed URLs (Section 3.3.2). To address the second challenge of detecting WebAssembly binaries during crawling, we analyze all websites through a combination of static and dynamic detection techniques.

3.3.1 Direct Downloads Guided by HTTP Archive. The *HTTP Archive* project[2] regularly crawls the web and makes the requests and responses available. Starting from URLs obtained from the Chrome User Experience Report , the project currently covers over 5 million top-level domains, monthly. We here focus on websites crawled using the desktop version of Google Chrome, which we access via Google's BigQuery database system.

We search the responses stored in the HTTP Archive tables for likely WebAssembly binaries and then directly download the corresponding files. To this end, we query two tables, from months May and June 2020, which contain information about all requests made while crawling the websites, and the corresponding responses. These tables, called `summary_requests` are 434.4 GB and 476.7 GB large. We filter all requests in the tables by the MIME type of the requested resource, keeping only those commonly used to serve WebAssembly, such as `application/wasm` and `application/octet-stream`, and where `.wasm` appears in the URL. These queries result in a set of 855 URLs. We download files from each of these URLs using `wget` and keep all that start with `\0asm`, WebAssembly's magic number.

3.3.2 Web Crawling. The HTTP Archive-guided search covers a wide range of top-level domains, but it may miss WebAssembly binaries on websites not covered by crawling a generic list of websites and binaries that one cannot identify based on their MIME type. To collect additional binaries, we also perform our own web crawling. There are three components to our crawling: the seed list, the crawling algorithm, and methods for detecting WebAssembly.

Seed lists. Any kind of web crawling requires a *seed list* of URLs to start from. We consider three seed lists, one generic list of popular websites and two lists targeted specifically at WebAssembly:

- *Top one million websites.* As a generic set of websites to explore, we start crawling from the one million most popular websites on the Tranco list [28], a top list more resilient to manipulation.
- *"WebAssembly" in JavaScript files.* WebAssembly binaries on websites must be executed by some surrounding JavaScript code, e.g., by calling `WebAssembly.instantiate`. To identify websites with such JavaScript code, we query a table provided by the HTTP Archive that stores the full bodies of all HTTP responses up to some size. We search this table, which has a total size of 9.32 TB, with Google BigQuery for all JavaScript responses that contain `WebAssembly` and add the URLs of the corresponding websites to our seed list, which results in about 40,000 URLs.
- *WebAssembly top lists.* As the most targeted seed list, we start crawling from three hand-curated lists of noteworthy WebAssembly-related websites. These websites cover projects using WebAssembly[3], tools and demos[4], and WebAssembly-based games[5].

Crawling algorithm. Given a seed list, our crawler visits each URL on the list and recursively follows links on the visited websites. The crawler visits each URL, with up to one retry. If the website is loading successfully, the crawler waits until either the "DOM content loaded" event is fired and all network connections have become idle, or until a 30-second timeout occurs. The crawler collects all WebAssembly binaries loaded or executed in this time (details below). For each visited website, the crawler extracts more URLs to explore from the `href` attribute of all `<a>`-tags on the site.

To control the amount of sites to visit, the crawler is configured with two parameters: the recursion depth *d*, which bounds how

[2] https://httparchive.org/

[3] https://madewithwebassembly.com/

[4] https://github.com/mbasso/awesome-wasm

[5] https://www.webassemblygames.com/

many links away from the seed URLs to explore, and the exploration breadth b, which bounds how many links to follow on each explored site. If a site has more than b links, the crawler picks b of them at random. For the first two seed lists, we set $d = b = 2$, i.e., the crawler visits at most seven sites per URL in the seed list. Because the third seed list is the most focused one, we explore it more thoroughly with $d = 7$ and $b = 3$, and repeat the exploration with 16 separate crawler instances. We chose those parameters based on preliminary experiments, to find most binaries in a given time budget.

Identifying WebAssembly binaries. For each website visited by the crawler, we use a combination of two techniques to identify WebAssembly binaries on the site. Our first detection mechanism intercepts the website's traffic using a local proxy that inspects the headers and contents of all requests and responses. To identify WebAssembly modules, we check if the content-type header matches `application/wasm` or `application/octet-stream`, or if the URL contains `.wasm`, and then ensure that the response payload starts with the proper magic number. If this is the case, we store the loaded file as a WebAssembly binary. The key advantage of this detection mechanism is that it detects WebAssembly modules even if they are not executed during the crawler's visit of the website. The second detection mechanism tracks calls to APIs used for instantiating WebAssembly modules, as proposed in prior work [24]. We transparently overwrite built-in JavaScript functions, such as `WebAssembly.instantiate`, and analyze its invocations. In contrast to the first detection mechanism, this mechanism can detect WebAssembly binaries that occur inline in JavaScript code, if executed.

3.4 Collecting Binaries Manually

In addition to automatically collecting WebAssembly binaries, we also gather a small number of binaries manually. On the one hand, we collect binaries through manual interaction with the web in daily browsing between April and September 2020. On the other hand, we asked WebAssembly developers on reddit.com/r/WebAssembly in June 2020 for binaries they are willing to share. As discussed in the results, these two manual collection methods complement our automatically collected binaries with otherwise missed examples.

3.5 Deduplication and Filtering

After collecting binaries and associated metadata from the aforementioned sources, we remove duplicates and filter binaries that are not representative of real-world applications. To deduplicate binaries, we compare files based on their SHA256 hash and remove identical files. Unless mentioned otherwise, our study focuses in the deduplicated dataset. In addition to deduplication, we remove binaries that are non-representative of real-world applications, because they fall into at least one of the following categories. Binaries that occur multiple times, e.g., across different sources, are only removed if all occurrences of it were filtered out.

- *Generated binary variants:* Some GitHub repositories contain binaries generated by research tools, e.g., to fuzz-test WebAssembly implementations, to superoptimize WebAssembly code [4], or to perform code diversification [2]. Since these tools turn a single binary into many, only slightly different variants, we remove the generated variants. We identify those variants by filename (e.g., `*.opt.wasm`) and path (e.g., binaries in `afl_out/`).
- *Test suites:* On GitHub and in some npm packages, we find binaries that are used as test inputs for WebAssembly-related tools, e.g., parsers, compilers, and virtual machines. One large portion are binaries from the official specification test suite, which often test only a single instruction or language construct. We identify them by typical repositories and paths (e.g., files in `spectest/`).
- *Tutorial projects:* Many npm projects and some GitHub repositories are instances of users following WebAssembly tutorials for particular tool chains.[6] Those binaries are small and all very similar. We identify them based on common binary names (e.g., `hello_world_bg.wasm`) and project names (e.g., `test-wasm@0.0.1`).
- *Small and invalid binaries:* Finally, we remove binaries that contain ten or fewer instructions, and binaries that cannot be validated by the reference WebAssembly binary toolkit (WABT), even with all language extensions enabled.

4 RESULTS

Based on our WasmBench dataset of real-world WebAssembly binaries, we address the research questions (RQs) described in the introduction. For each RQ, we detail the analyses performed on the dataset, the direct results, and then interpret those to obtain *insights*, i.e., take-away points, often with a focus on security.

4.1 Implementation and Experimental Setup

The crawler is implemented based on Puppeteer and Puppeteer Cluster, two Node.js libraries for controlling instances of the Chromium browser, here version 83.0.4103.0. The static analyses described in the following are implemented in several Rust programs to statically extract relevant features, such as instructions, names, etc. from the binaries, complemented by Python scripts that perform the final analyses. For parsing binaries, we use the wasmparser library, a project by the Bytecode Alliance. All experiments were run on an Ubuntu 18.04 machine with two Intel Xeon CPUs at 2.2 GHz running 48 threads, equipped with 256 GB of memory. The crawling was performed in chunks of 50,000 websites, where each chunk took about 7 hours to finish, with a total of about 10 days for all crawling. The static analyses usually finish within several minutes for the entire dataset. Our entire dataset and the implementation are available for others to build on at https://github.com/sola-st/WasmBench.

4.2 Overview of Dataset

Table 1 gives an overview of our dataset. For each source, the table shows how many binaries we found, and how many remain after deduplication and filtering. The last column shows how many binaries are found only via a single source, illustrating the importance of particular collection methods for obtaining a diverse dataset.

Sources. The largest contribution to the dataset are the GitHub repositories and packages from npm. Given that WebAssembly binaries on websites or in arbitrary packages are still relatively scarce, selecting repositories and packages related to WebAssembly

[6] For example, the *rustwasm* book: https://rustwasm.github.io/docs/book/.

Table 1: Contribution of different sources to the dataset.

Source (see Figure 3)	Found Binaries			
	Total	Unique	Filtered	Only
GitHub, search: wasm	44,218	21,117	6,830	6,641
NPM, top dependend-upon	14	8	8	3
NPM, search: wasm	3,488	2,452	1,163	1,036
WAPM, all	122	113	108	81
Firefox add-ons, top by users	29	17	17	15
Web, HTTP Archive	261	141	141	54
Web, crawling, with seed list:	2,923	432	412	298
HTTP Archive	2,046	268	254	128
Tranco top websites	769	167	164	86
WebAssembly top lists	108	89	76	43
Manual	93	89	88	57
All Sources	51,148	23,413	8,461	

is an effective way of finding binaries. At the same time, the sources where we do not query for WebAssembly specifically (i.e., most of the web crawling, the top packages from npm, and Firefox addons) still show that WebAssembly binaries are found in the wild in popular projects used by millions of users. For the web, we also see that all our four sources are essential for finding a diverse set of WebAssembly binaries. Only crawling the top one million websites would miss at least 225 unique binaries. The fact that the seed lists from HTTP Archive and the WebAssembly top lists are much smaller than the list of the top one million websites, yet the number of found binaries are similar or even higher, shows that a targeted seed list for crawling is key to finding otherwise missed binaries.

Insight 1. All methods we use to collect binaries contribute in a non-negligible way. Combining different sources and collection techniques is crucial for obtaining a representative dataset.

Filtering. Deduplication and filtering non-representative binaries (Section 3.5) significantly reduces the dataset (Table 2). In particular, one repository that contains generated variants of input binaries is important to filter, as it otherwise accounts for over 8,000 unique binaries. From the second category, test suites, we also see that test binaries are commonly reused across many projects (26,138 occurrences, but only 5,283 unique binaries), and that most of them are from the official specification test suite. The last filter removes binaries from a few, very similar tutorials, including more than 500 binaries from projects called `hello-wasm`.

Binary sizes. As a first proxy for the diversity of the collected binaries, we look into their sizes and instruction count. Figure 4 shows the histogram and cumulative distribution of binary sizes in bytes. The distribution of the number of instructions is similar in shape and omitted for space reasons. While there are many small binaries, there is a long and heavy tail towards larger sizes. Two thirds of the binaries are larger than 20 KB and have more than 8,700 instructions. The median binary is 37.1 KB large and has 14,885 instructions. The largest binaries are a WebAssembly port of TiDB[7], a distributed SQL database written in Go, with 75.1 MB and 16.9M instructions, respectively, found as a wapm package;

Table 2: Binaries filtered out due to different criteria.

Filter	Removed Binaries	
	Total	Unique
Generated binary variants, of those:	9,279	8,048
in CROW [2] repository	8,025	7,987
Test suites and files, of those:	26,138	5,283
variants of WebAssembly spec suite	25,133	4,931
Invalid WebAssembly binaries	13,593	3,513
Small binaries: <10 instructions	10,146	1,881
Tutorial projects, of those:	848	633
`hello-wasm` projects	695	506
All Filters	37,808	14,952

opencascade.js[8] (65.8 MB, 22.2M instructions), a WebAssembly port of an open source C++ CAD library, found on npm and GitHub; and finally the Clang compiler, itself compiled to WebAssembly (46.7 MB, 12.6M instructions), found on wapm and GitHub.

Insight 2. Complex, real-world applications with millions of lines of code are compiled to WebAssembly.

4.3 RQ1: Source Languages and Tools

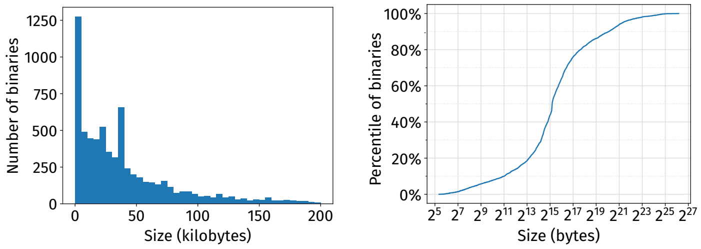
Given WebAssembly's goal of being a universal bytecode, we study which languages are compiled to it in practice, and which toolchains are used for to it.

Analysis. It is non-trivial to infer from a binary which source language and compiler has produced it. We rely on several complementary methods. First, we check the `producers` section, where some toolchains explicitly encode the source language(s) a program is compiled from. Second, our analysis searches for *characteristic function names* that appear in the `import` section, the `export` section, or the optional `name` section. For example, `_ZdaPv` is the name-mangled `delete` operator of C++; or `runtime.gostring` is a Go runtime library function. Overall, we identify characteristic function names for C++, C, Rust, Go, AssemblyScript, Kotlin, and FStar. Third, the analysis searches for *characteristic strings* among all text sequences of >3 ASCII characters in the `data` section. For example, being core types, `Result::unwrap` and `Option::unwrap` frequently appear in error messages of the Rust standard library. We identify characteristic strings for C++, Rust, Matlab, and COBOL. Fourth, if none of the above work, we analyze *sibling files* of binaries collected from code repositories and package managers. Sibling file here means a file in the same directory that shares the file name except for the extension. We take into account extensions for C, C++, Rust, Go, AssemblyScript/TypeScript, the WebAssembly text format (`.wat`/`.wast`), and several smaller languages. Finally, for some source code repositories and packages with multiple unidentified binaries, we *manually inspect* source code, build scripts, and binaries. For each of the automated methods above, we manually inspect binaries and the predictions to confirm that our heuristics are precise. For binaries where multiple methods identify the source language, we confirm that the predictions are consistent.

Results. Figure 5a shows the inferred source languages. We see that almost two thirds (64.2%) of the binaries are compiled from C,

[7] https://github.com/pingcap/tidb

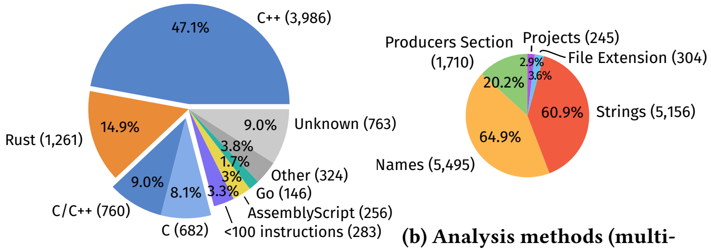
[8] https://github.com/donalffons/opencascade.js



(a) Histogram of the lower 80% of the sizes of the binaries.

(b) Cumulative distribution of the binary sizes (in $\log_2$ scale).

Figure 4: Distribution of binary sizes.



(a) Source languages.

(b) Analysis methods (multiple can apply per binary).

Figure 5: Source languages and methods for inferring them.

C++, or a combination of both. Given that these are memory-unsafe languages, plagued with decades of vulnerabilities [41] and that WebAssembly binaries are not automatically safe from exploitation [18], this result is highly worrying.

Insight 3. Almost two-thirds of all collected binaries are compiled from memory-unsafe source languages. These results and those in the next section call for techniques to analyze and ensure WebAssembly's binary security.

Rust comes in second place with 14.9% of all binaries, followed by AssemblyScript (3%) and Go (1.7%) as source languages with official WebAssembly support. Finally, there is a longer tail of other languages, often used in single projects: Matlab[9] (0.69%), FStar[10] (0.33%), CHIP-8[11] (0.26%), several binaries compiled from toy languages, and even a single instance of COBOL. A small portion of binaries (1.1%) is translated directly from the WebAssembly text format, i.e., likely to be written by hand. Finally, for 3.3% of all binaries we could not assign a source language, but since they contain less than 100 instructions, they are also likely to be written manually.

Insight 4. In addition to C/C++, various other languages are compiled to WebAssembly, including languages with garbage collection and heavier runtimes (Go, Matlab). This result matches WebAssembly's goal of serving as a universal bytecode. It also means binary analysis will become more important, since source code is not always available, and even if it is, implementing separate analyses for many languages is impractical.

We also analyze the tools used to produce the binaries. For 20.2% of the binaries, the producers section explicitly mentions them in the `processed-by` field. 10.8% of all binaries explicitly mention being produced by Clang. No binaries mention Emscripten because it does not emit a `producer` section, unlike newer versions of Clang. These results show that Emscripten is no longer the only way to compile C and C++ to WebAssembly. Other tools that appear in the `producer` section are `rustc` (9.5%), `wasm-bindgen`[12] (7.9%), a JavaScript host-code generator, and `walrus` (7.5%)[13], a binary transformation library, and the official Go compiler (0.4%). Since all compilers for Rust, C, and C++ to WebAssembly are based on LLVM, we can also derive that 79.1% of the binaries are produced with the help of LLVM.

Insight 5. Almost 80% of all binaries are compiled with the help of the LLVM toolchain. This implies that security mitigations, such as stack canaries, would have a large effect on the ecosystem if implemented in this toolchain.

Figure 5b shows which of our methods are most effective at inferring the source language. 20.2% of the binaries contain a producers section, from which the source code language can be directly obtained. Characteristic names and strings are also important inference methods, since they apply to 64.9% and 60.9% of binaries, respectively. Overall, our methods infer the source language for 91% (7,698) of the 8,461 unique binaries.

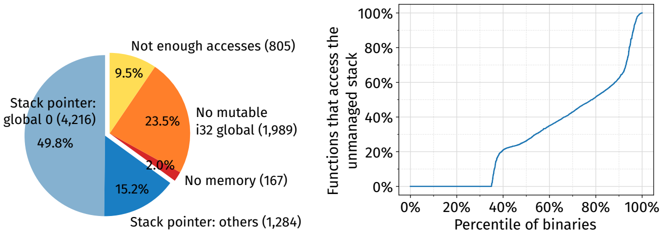
4.4 RQ2: Vulnerabilities Propagated from Source Languages

Recent work shows that memory vulnerabilities in unsafe source languages can propagate to WebAssembly binaries and may sometimes be exploited even more easily than for native binaries [18]. While this prior work evaluates the risks of such attacks on a small set of 26 binaries, most of which are compiled C/C++ benchmarks, it remains unclear to what extent propagated vulnerabilities may affect real-world WebAssembly binaries.

We address this question by studying three important characteristic of binaries that attackers can abuse: (i) uses of the unmanaged stack, i.e., an unprotected representation of the function stack in WebAssembly's linear memory (Section 2), which attackers can use for stack-based buffer overflows and stack overflows (Section 4.4.1); (ii) unsafe memory allocators compiled into a binary, which attackers can abuse as a memory write primitive (Section 4.4.2); and (iii) accesses to potentially dangerous APIs imported from the host environment (Section 4.4.3). For (i) [18] reports results on 26 binaries only. We refine their static analysis and consider a 325× larger dataset. For characteristics (ii) and (iii), this work is the first to systematically evaluate their prevalence in real-world WebAssembly binaries. We study all binaries, irrespective of the source language, because the problem of propagated vulnerabilities may affect all languages with memory-unsafe behavior, in particular C, C++, but also, e.g., Rust, as its `unsafe` keyword is commonly used [7] and can cause memory-safety related vulnerabilities [46].

4.4.1 Usage of the Unmanaged Stack. The so-called "unmanaged stack" is a region within the linear memory of a WebAssembly program that holds, e.g., non-primitive data with function livetime. This design is motivated by the fact that all non-scalar data and all data of which an address is taken cannot be put in WebAssembly's locals or globals, but must instead reside in linear memory. Given

[9] https://github.com/Sable/matwably
[10] https://github.com/FStarLang/kremlin
[11] An 8-bit VM language from 1970s, https://github.com/pepyakin/emchipten
[12] https://github.com/rustwasm/wasm-bindgen
[13] https://github.com/rustwasm/walrus



(a) Proportion of binaries with identified stack pointer, and if not why.

(b) Cumulative distribution of proportion of functions in a binary that use the unmanaged stack.

Figure 6: Unmanaged stack usage in binaries.

that buffer overflows on the unmanaged stack can overwrite across stack frames and even into supposedly "constant" data, this makes the unmanaged stack a more dangerous exploitation target than even in native programs [18]. For this reason, we evaluate how many binaries use it in practice.

Analysis. To analyze the usage of the unmanaged stack, our static analysis performs two steps. First, it tries to identify the stack pointer to determine whether a binary uses an unmanaged stack at all. Out of all global variables, the analysis selects the one that

- has type `i32`, i.e., the type of all pointers,
- is declared mutable, to exclude constants like `STACK_MAX`,
- is the most read and written global, as determined by the number of `global.get` × `global.set` instructions[14], and
- has at least three reads and at least three writes, to avoid false positives in small binaries.

We manually validate that these heuristics identify the stack pointer reliably on randomly sampled binaries. If the analysis cannot identify a stack pointer, it conservatively assumes that the binary does not use an unmanaged stack. Second, once the stack pointer is identified, the analysis counts the number of functions in the binary that access the stack pointer somewhere in their body. Our implementation builds upon the prototype analysis provided by Lehmann et al.[15], but uses a different WebAssembly parser to also handle language extensions and makes the analysis robust enough to run on thousands of real-world binaries.

Results. Figure 6a shows that almost two thirds (65%) of all binaries use the unmanaged stack. While most of them use the global with index 0 as their stack pointer, our heuristics to identify the stack pointer are important, since 15.2% of all binaries use another global variable. For 2% of the binaries, the analysis can clearly determine that they have no unmanaged stack in linear memory, simply because there is no linear memory at all. For 23.5% of the binaries, there is a linear memory section, but no mutable `i32` global that could be a stack pointer. Finally, 9.5% of all binaries have at least one candidate mutable global pointer, but it is not accessed often enough for our analysis to consider it the stack pointer. Interestingly, AssemblyScript programs are in the last category, since its runtime seems to not support stack allocation.

To better understand how much a binary uses the unmanaged stack, Figure 6b shows how many of the functions in a binary access the stack pointer at least once. Consistent with Figure 6a, in 35% of all binaries no function uses the stack pointer because none is present. In the median binary, already 33% of all functions use the stack pointer, and in some binaries almost every function uses the unmanaged stack. On average across all binaries *with* an unmanaged stack, 44% of their functions make use of it.

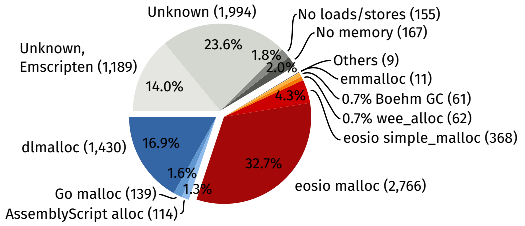


Figure 7: Allocators in binaries, multiple can apply.

Insight 6. Many binaries (65%) and functions in those binaries (44%) use the unmanaged stack, which attackers may abuse for runtime exploitation. This result extends earlier findings [18] to a much larger and more diverse set of real-world binaries.

4.4.2 *Statically Linked Allocators.*

WebAssembly's memory organization is very low-level. Besides the single linear memory section, which can be expanded at runtime with the `memory.grow` instruction, no help with allocating memory is provided by the language. Subdividing the linear memory, e.g., to avoid fragmentation and to reuse space of deallocated objects, needs to be handled by an allocator that is statically linked into the binary. Especially for binaries on the web and for smart contract platforms, code size is an important consideration, so developers can choose a lightweight allocator instead of the default allocator provided by the compiler. Prior work has shown [18] that those smaller allocators can lack important mitigations against heap metadata corruption and yield powerful arbitrary write primitives for an attacker. However, it remains unclear what allocators developers use in practice.

Analysis. To identify allocators in a binary, we rely on similar heuristics as for source language detection (Section 4.3). That is, we first identify allocators by characteristic *function names* in binaries, if those are available. Then, we inspect frequent *strings* in the `data` section of binaries, e.g., for error messages of certain allocators.

Results. Figure 7 shows our results, grouped into three categories. In blue, we mark default allocators provided by programming languages and compilers, in different shades of red we mark other allocators that we identified, and in gray when we could not identify an allocator. In terms of default allocators, we see that 16.9% of all binaries use *dlmalloc*, the default allocator provided by Emscripten, Clang, and the Rust compiler when targeting WebAssembly. Go and AssemblyScript allocators are present roughly in the proportion of their respective languages.

Among the non-default allocators, two particular ones dominate, being in 32.7% and 4.3% of our binaries. They are both from EOSIO, a smart contract platform that uses WebAssembly as its bytecode. Those contracts can be written in C++ and compiled with Emscripten. However, most of them are not using Emscripten's default allocator. While we did not perform an in-depth security analysis of EOSIO `malloc` and `simple_malloc` we can attest that both are considerably shorter in terms of code and do not feature any

[14] The product of the counts prefers globals which are similarly often read and written. This is true for the stack pointer, but not for other frequently accessed pointers.
[15] https://github.com/sola-st/wasm-binary-security

assertions that would guard against metadata corruption. In our dataset, we also find wee_alloc (62 binaries) and emmalloc (11 binaries), two small allocators for Rust and Emscripten respectively, that were already found to be vulnerable against heap metadata corruption attacks [18]. Other interesting custom allocators are Boehm GC (a mark-and-sweep garbage collector) and gperftools, in several binaries collected from Google domains.

Insight 7. WebAssembly binaries come with a variety of memory allocators, including many custom allocators (38.6%), increasing the risk to include vulnerable allocators. If code size is the motivation to use custom allocators, a more secure alternative could be a memory allocation or garbage collection API provided by the host environment [33].

4.4.3 Imports of Security-Critical APIs from Host Environment. To exploit a WebAssembly binary, an attack proceeds in two steps. The first step is compromising the state or behavior of the WebAssembly binary itself, e.g., by exploiting an unsafe allocator (Section 4.4.2) or a buffer overflow on the unmanaged stack (Section 4.4.1). The second step is actually performing the malicious action to the underlying system. The only way to do so, assuming VM implementations are bug-free and host security[16] is perfect (which they are not [1, 36]), is to call functions imported into the WebAssembly binary from the host environment. For example, an attacker could pass an injected string on the unmanaged stack to an imported function, e.g., JavaScript's eval. To estimate how often WebAssembly binaries use such security-critical host APIs, we thus analyze their imports.

Analysis. We identify imported security-critical APIs based on their import name in the WebAssembly binary. Going by name (instead of implementation) is necessary because, (1) the implementation of an imported function is supplied by the host only at instantiation-time, so it is not available when given only the binary; and (2) there are many host environments, not all of which are using JavaScript. WASI for example, defines imports that can be implemented by different standalone WebAssembly VMs in native code. We thus identify import names for which the host implementation is likely a security-critical function. E.g., the import emscripten_run_script in WebAssembly binaries is typically bound to Emscripten-generated JavaScript code that calls eval. We match imports against 18 patterns in five categories known to be potentially security-critical APIs:

- *Code execution.* Imports like eval, exec, or emscripten_run_script.
- *Network access.* Imports containing XHR, request, http, or fetch.
- *File I/O.* Imports containing file, fd, or path.
- *DOM interaction.* Imports containing document, html, body, or element could manipulate the DOM, which can lead to XSS.
- *Dynamic linking.* dlopen, dlsym, and dlclose allow loading additional code at runtime, which can lead to code injection.

To avoid spurious matches, especially for short patterns like fd, we tokenize import names based on camel-case and non-alphabet characters, and then check for a pattern to occur verbatim in the token sequence. E.g., fd_write matches our file I/O category, but bufdelete does not. We manually inspect matches to ensure they are plausible and remove benign matches otherwise.

Table 3: Imports matching potentially security-critical APIs.

Category	Patterns	Matching		
		Imports	Binaries	%
Code execution	eval, exec, execve emscripten_run_script	383	160	1.9%
Network access	xhr, request, http, fetch	944	172	2.0%
File I/O	file, fd, path	7,532	1,610	19.0%
DOM interaction	document, html, body, element	1,720	212	2.5%
Dynamic linking	dlopen, dlsym, dlclose	352	138	1.6%
At least one		10,468	1,797	21.2%

Results. Table 3 shows the results of our name-based import analysis. The first two columns show the category and patterns we match import names against. In the third column, we count imported functions that match at least one pattern. The last two columns show the number of binaries with at least one matching import, and which fraction of the filtered dataset this corresponds to. We see that imports related to file I/O, the most common category, are present in almost every fifth binary. Interestingly, even though WebAssembly was originally not meant to replace JavaScript, but rather for compute intensive applications, still 212 binaries likely interact with the DOM from WebAssembly, which attackers could use for cross-site scripting. In the last row, we see that overall 21.2% of all binaries import at least one potentially security-critical API.

Insight 8. Many binaries (21.2%) import potentially dangerous APIs from their host environment, which may allow compromised binaries, e.g., to inject arbitrary code or to write to the file system.

4.5 RQ3: Cryptomining

A study of real-world uses of WebAssembly performed in early 2019 [24] reports cryptomining to be one of the most common use cases of WebAssembly on the web. That study found 55.7% of the analyzed websites to use WebAssembly for cryptojacking, i.e., the practice of using a website visitor's hardware resources for mining cryptocurrencies without their consent. Identifying and controlling cryptominers on the web has been the focus of several recent pieces of work [15, 25, 34, 43]. In this research question, we study whether cryptomining is still an important threat today. We address this question in two ways. First, we analyze those binaries we collected from the web for signs of being cryptominers. Second, we directly compare the binaries gathered in earlier work with our dataset.

4.5.1 Analyzing Binaries Found on the Web. To understand the prevalence of cryptomining today, we analyze all binaries collected from the web, i.e., direct downloads guided by HTTP Archive and the results of our own crawling, using VirusTotal. The VirusTotal API allows to upload and scan files with up to 70 independent third-party antivirus scanners and malware detectors, and reports back the number of positive results. Among the 352 analyzed binaries, VirusTotal reports four files to contain malicious content. Three of them are likely to be the same program, as they have similar sizes (68.8 ± 0.7kB) and the same distribution of instructions. These three

[16] Host security: isolation of WebAssembly execution from the underlying host, e.g., ensuring that a WebAssembly program can write only to its designated memory region.

files are detected by 26 or more scanning tools employed by VirusTotal. One of the files is collected from http://monero.cit.net, which further supports the presumption that the binary is a cryptominer, as "Monero" is the name of a common cryptocurreny. Moreover, one of three binaries is identical to a binary we collect also from a GitHub repository called "CryptoNoter"[17], which is an open-source Monero cryptominer. The fourth file reported by VirusTotal is tested positive by only one scanner. Our manual analysis shows that the report for this binary is likely to be a false positive.

4.5.2 Comparison with Dataset by Musch et al. [24]. The previous study is based on 147 unique WebAssembly binaries, which the authors kindly shared with us. The intersection of their dataset with ours contains 23 binaries, i.e., 16% of their dataset and 0.2% of our dataset. To better understand these binaries, we manually examine them and visit the corresponding websites. We find four of the 23 binaries to be suspicious. Two of them are among the files flagged by VirusTotal, as discussed above. For one file from a website that declares itself to be a "blockchain explorer", we could not observe any suspicious activity when visiting the source website, but also could not identify its functionality, and thus declare it to be suspicious. For the last suspicious binary, visiting the corresponding website increases CPU load to 70%. The website offers a service to mine cryptocurrency and openly advertises the fact that one can start mining immediately in the browser. That is, the binary is an example of cryptomining but not cryptojacking.

In summary, we identify only four binaries from our "web" dataset as possible cryptominers (about 1% of the dataset), three of which appeared to be inactive when visiting the website. While our analysis may miss cryptomining binaries, a risk one could reduce through additional analysis techniques beyond those provided via VirusTotal [24, 43], the prevalence of cryptomining seems to have dropped significantly over the past one to two years. This result is also confirmed by a manual analysis of WebAssembly binaries found on the web (Section 4.6.2) and is in line with other reports [42] that cryptomining became less appealing after one of the major cryptomining script providers, Coinhive, shut down. Varlioglu et al. find a 99% decrease in sites using cryptomining among sites that had made use of it before. The low numbers of cryptominers found by our analysis confirms this trend and shows its declining influence on the WebAssembly ecosystem.

Insight 9. We find WebAssembly-based cryptominers to have significantly dropped in importance compared to the results of an earlier study [24]. This finding motivates security research to shift the focus from malicious WebAssembly to vulnerabilities in WebAssembly binaries.

4.6 RQ4: Use Cases on the Web

Given the decreased prevalence of cryptominers, we study what other use cases WebAssembly has. The following focuses on the web because it is the most prominent target platform of WebAssembly and because websites are complete applications that we can manually analyze with reasonable effort. We address the question in two ways. First, we study binaries that occur across multiple websites, which helps understand libraries and other widely reused components (Section 4.6.1). Second, we inspect a random sample of 100 unique binaries, which helps understand the application domains of WebAssembly (Section 4.6.2). To capture the full picture of WebAssembly use cases, the results are on unfiltered binaries.

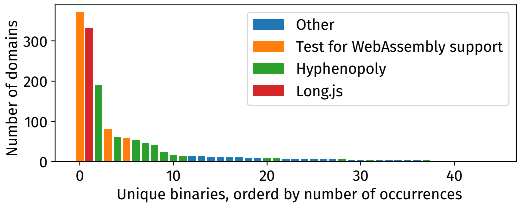


Figure 8: Binaries found on multiple websites.

4.6.1 Binaries Found on Multiple Websites. Out of all 476 unique binaries found on the web, 70 are reused across at least two different top-level domains. Figure 8 shows a histogram of how often binaries occur on multiple domains. The data follows a long-tail distribution, i.e., a few binaries occur on many websites, while many other binaries recur a few times or only once. The top-most widely distributed binary occurs 371 times, i.e., in 28% of all domains where we detect WebAssembly binaries.

To better understand the most recurring binaries, we analyze them through a combination of automated clustering and manual inspection. The automated clustering represents each binary as a set of byte n-grams [21], summarizes the number of n-gram occurrences in a binary into a characteristic vector, and then clusters binaries based on the pairwise cosine similarity of their vectors. We then inspect the top-most binaries in Figure 8, using the clusters to quickly identify variants of the same binary. Our analysis shows the following to be the most widely occurring WebAssembly binaries.

Testing for WebAssembly support. At least 509 domains (38.5% of all domains that use WebAssembly) are serving a WebAssembly binary that tests whether the browser supports WebAssembly at all. We found two variants of such binaries, both of which are rather small: a six instruction binary with a single function called `test` and an eight byte binary that only contains the WebAssembly magic number followed by the language version. Websites serving these test binaries often also serve larger binaries, i.e., they first test whether WebAssembly is supported, and if it is, load a more complex binary. For example, at least 397 domains that serve a test binary also serve the Hyphenopoly library discussed next.

Hyphenopoly. At least 462 domains (34.9% of all domains that use WebAssembly) serve binaries that are part of the *Hyphenopoly.js* JavaScript library, which uses WebAssembly to implement its core functionality. Hyphenopoly is a polyfill that "hyphenates text if the user agent does not support CSS-hyphenation".[18] Our clustering identifies 24 variants of this binary, which are all generated from the same underlying library to support different natural languages.

64-bit integer arithmetic in long.js. At least 331 domains (25% of all domains that use WebAssembly) serve a binary that is part of *long.js*, a JavaScript library for 64-bit integer computations.[19] The library is commonly used in video players.

Draco library for 3D data compression. At least 25 domains (1.8% of all domains that use WebAssembly) serve a binary that belongs to

[17] https://github.com/JayWalker512/CryptoNoter

[18] https://github.com/mnater/Hyphenopoly

[19] https://github.com/dcodeIO/long.js

Table 4: Application domains of 100 randomly sampled, unique WebAssembly binaries found on the web.

Application domain	# Binaries	Application domain	# Binaries
Games	25	Online gambling	2
Text processing	11	Barcodes and QR codes	2
Visualization / Animation	11	Room planning / Furniture	2
Media processing / Player	9	Blogging	2
Demo, e.g., of a programming language	7	Cryptocurrency wallet	2
		Regular expressions	1
Wasm tutorial or test	5	Hashing	1
Chat	3	PDF viewer	1

the *Draco* library, which support compressing and decompressing 3D data.[20] These binaries commonly occur on websites with 3D demos or integrated 3D assets.

Insight 10. The most widely occurring binaries on the web are dynamic tests for WebAssembly support and JavaScript-WebAssembly libraries that perform computation-heavy tasks.

4.6.2 Manual Inspection of a Random Sample. To better understand the long-tail of binaries found on a few or only a single website, we also inspect a random sample of 100 unique binaries found on the web. We exclude binaries detected only via the WebAssembly top lists (Section 3.3.2) to avoid biasing the results toward pre-selected application domains. By inspecting the binaries, the corresponding websites, and how the websites uses the binaries, we identify the purpose of 84 out of the 100 binaries. Table 4 summarizes the application domains that the binaries are used in. The most common domains are games, accounting for a quarter of all binaries. Text processing and applications in visualization and animation are also relatively common, with 11/100 binaries each. The remaining list shows the diversity of application domains WebAssembly is used in, ranging from online demos of programming languages, over support for creating and scanning barcodes to document viewers.

Insight 11. The application domains of WebAssembly binaries on the web reflect the diversity of the web itself, showing that WebAssembly is used in a wide range of applications.

4.7 RQ5: Minification and Names

As a binary format with only a low-level textual representation, WebAssembly binaries cannot be as easily inspected and understood as source code, e.g., in JavaScript. The ability to understand a WebAssembly binary is relevant for auditing third-party code and reverse engineering malware. For example, when a frequently depended-upon npm package contains a WebAssembly binary, the package distribution platform may want to check that it does not perform malicious actions, such as stealing cryptocurrency.[21] Because meaningful names, e.g., for functions, are helpful for understanding code [17], especially in binaries, we study to what extent WebAssembly binaries provide meaningful names.

Analysis. We perform a static analysis to assess two name-related characteristics of binaries. First, the analysis checks whether a binary contains a `names` section. While by default binaries contain names only for imported and exported program elements, the optional `names` section maps *all* function indices to identifiers. Second, the analysis checks whether the names of imported and exported names are minified. Both to save space and to obfuscate the code, compilers may shorten names down to single or two-letter names devoid of information. The analysis considers a WebAssembly binary as minified if it contains more than ten import or export names (to exclude small, potentially hand-written modules), but the average length of those names is ≤ 4 (to account for some imports that are never minified by Emscripten, thus increasing the average).

Results. Our results show an interesting difference between the full dataset and binaries found on the web. In the full data set, many binaries contain a `names` section (19.6%) but only 4.1% of the binaries are minified. In contrast, among all binaries found on the web, only 13.3% contain a `names` section but 28.8% are minified. These results show that for a significant fraction of websites, not only minified JavaScript code [37], but also minified WebAssembly binaries make it harder to understand what code is running on the client side.

Insight 12. Many WebAssembly binaries on the web (28.8%) are minified and do not contain useful names. To help security analysts understand third-party code, future work on decompiling and reverse engineering WebAssembly is needed.

5 RELATED WORK

WebAssembly in general. WebAssembly has been formally defined [10], including a mechanized proof of the soundness of its type system [44]. Since the initial version of the language, several language extensions have been proposed [6, 32, 33].

Cryptomining. Cryptojacking, i.e., websites that use the unsuspecting client's computing resources for mining cryptocurrencies, has been among the first applications of WebAssembly [15, 25, 34]. Several techniques detect and defend against cryptojacking [14, 43]. Section 4.5 studies how prevalent this threat is, showing that its importance has decreased over time. Wang et al. [43] discuss limitations of VirusTotal in identifying cryptomining, which may impact the validity of our results. However, our manual inspection of binaries confirms the low prevalence of cryptominers among today's WebAssembly binaries, which is also supported by Varlioglu et al.'s observations about the decline of cryptojacking [42].

WebAssembly attacks. Beyond cryptomining, other kinds of attacks based on WebAssembly exist. Lehmann et al. show that vulnerabilities that propagate from memory-unsafe source languages may also be exploited in WebAssembly [18]. Others report examples of such attacks [3, 22]. Custom memory allocators are potentially not hardened [5, 22]. Section 4.4 shows that several of the risks reported by prior work affect a wide range of binaries. Another line of attack are malicious WebAssembly binaries, e.g., to escape the browser sandbox [1, 36], attacks that use side channels [9], and attacks based on speculative execution [20].

WebAssembly defenses. A defense against application-level attacks is to enforce security policies on untrusted WebAssembly binaries through taint tracking [8, 40]. WebAssembly also serves

[20] https://github.com/google/draco

[21] https://blog.npmjs.org/post/185397814280/plot-to-steal-cryptocurrency-foiled-by-the-npm

as a technology to implement defenses, e.g., to sandbox libraries executed in a browser [26], to implement formally verified cryptography [30], or to ensure constant-time operations for cryptographic primitives [45]. Our results call for additional mitigations, e.g., to defend against vulnerabilities propagated from source languages, and provide guidelines for developing such techniques, e.g., by showing which toolchains are most commonly used.

Studies of WebAssembly. Musch et al. [24] systematically collect WebAssembly from the web and report cryptomining to be one of its prime use cases. Our work extends their findings in several ways: (i) by considering a wider range of sources to gather binaries, which results in 58× more binaries; (ii) by showing that other applications than cryptomining have become much more prevalent; and (iii) by studying several properties of WebAssembly not considered before, e.g., security properties and toolchains. Other work studies the performance of WebAssembly and compares it to native performance [12], however again on a small set of binaries.

WebAssembly benchmarks. Prior work on WebAssembly often relies on benchmark suites that may not well represent the diversity of real-world WebAssembly binaries, such as PolyBenchC, SciMark, and Ostrich [13], i.e., benchmarks of numerical or scientific computations [10, 11, 19], SPEC CPU, i.e., benchmarks of complex C/C++ programs that are not typically compiled to WebAssembly [12, 18], and small sets of hand-picked applications [18]. This paper instead presents a set of thousands of real-world binaries collected from various sources, which we make available for future research.

Studies of other languages and ecosystems. Beyond WebAssembly, other studies investigate JavaScript and web security in general, including studies of minified and obfuscated code in the web [37], of the use of the `eval` [31], of the communication between websites and embedded frames with 3rd-party content [38], of outdated libraries in the web [16], of trust relationships between websites that include remote libraries and their corresponding library providers [27], of implicit type conversations in JavaScript code [29], of ReDoS vulnerabilities in JavaScript-based web servers [39], of XSS vulnerabilities [23], and of performance issues in JavaScript [35]. Inspired by all that work, this paper fills in important gaps in the existing knowledge about security properties of real-world WebAssembly.

6 CONCLUSION

This paper presents a comprehensive empirical study of security properties, languages, and use cases of a diverse set of real-world WebAssembly binaries. After gathering binaries from several sources, ranging from source code repositories over packages managers to live websites, we analyze them through a combination of static code analysis, manual inspection, and statistical analysis. Our study shows that WebAssembly has grown into a diverse ecosystem with new challenges and opportunities for security researchers and practitioners, e.g., in analyzing vulnerabilities in WebAssembly binaries, in hardening binaries against exploitation, and in helping security analysts reverse engineer binaries. We make the binaries underlying our study, which yields by far the largest benchmark of WebAssembly binaries to date, available to support future work.

ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and Perf4JS projects.

REFERENCES

[1] Georgi Geshev Alex Plaskett, Fabian Beterke. 2018. *Apple Safari – Wasm Section Exploit.*

[2] Javier Cabrera Arteaga, Orestis Floros Malivitsis, Oscar Luis Vera Pérez, Benoit Baudry, and Martin Monperrus. 2020. CROW: Code Diversification for WebAssembly. *arXiv preprint arXiv:2008.07185* (2020).

[3] John Bergbom. 2018. *Memory safety: old vulnerabilities become new with WebAssembly.*

[4] Javier Cabrera Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, and Martin Monperrus. 2020. Superoptimization of WebAssembly bytecode. In *ICPS Companion 2020.* 36–40.

[5] Frank Denis. 2018. *WebAssembly doesn't make unsafe languages safe (yet).*

[6] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. 2019. Position Paper: Progressive Memory Safety for WebAssembly. In *HASP.*

[7] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust used Safely by Software Developers?. In *ICSE.* 246–257.

[8] William Fu, Raymond Lin, and Daniel Inge. 2018. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. *CoRR* abs/1802.01050 (2018).

[9] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. 2018. Drive-By Key-Extraction Cache Attacks from Portable Code. In *ACNS.*

[10] 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. In *PLDI.*

[11] David Herrera, Hanfeng Chen, Erick Lavoie, and Laurie Hendren. 2018. Numerical computing on the web: benchmarking for the future. In *DLS.* ACM, 88–100.

[12] 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 ATC.* 107–120.

[13] Faiz Khan, Vincent Foley-Bourgon, Sujay Kathrotia, Erick Lavoie, and Laurie J. Hendren. 2014. Using JavaScript and WebCL for numerical computations: a comparative study of native and web technologies. In *DLS'14.* ACM, 91–102.

[14] Amin Kharraz, Zane Ma, Paul Murley, Charles Lever, Joshua Mason, Andrew Miller, Nikita Borisov, Manos Antonakakis, and Michael Bailey. 2019. Outguard: Detecting in-browser covert cryptocurrency mining in the wild. In *WWW '19.*

[15] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. 2018. MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense. In *CCS 2018.*

[16] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *NDSS 2017.*

[17] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *ICPC.* 3–12.

[18] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly *(USENIX Security 2020).* 217–234.

[19] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A framework for dynamically analyzing webassembly *(ASPLOS 2019).* 1045–1058.

[20] Giorgi Maisuradze and Christian Rossow. 2018. Ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS.*

[21] Christopher D Manning, Hinrich Schütze, and Prabhakar Raghavan. 2008. *Introduction to information retrieval.* Cambridge university press.

[22] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. 2018. Security Chasms of WASM. NCC Group Whitepaper.

[23] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. *Network and Distributed System Security Symposium (NDSS).*

[24] 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 *(DIMVA 2019).* Springer, 23–42.

[25] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. Thieves in the Browser: Web-based Cryptojacking in the Wild. In *ARES.*

[26] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *USENIX Security.*

[27] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *CCS.* 736–747.

[28] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2018. Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1802.01156* (2018).

[29] Michael Pradel and Koushik Sen. 2015. The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript. In *ECOOP*.

[30] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. 2019. Formally Verified Cryptographic Web Applications in WebAssembly. In *SP*.

[31] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011. Automated construction of JavaScript benchmarks. In *OOPSLA*. 677–694.

[32] Andreas Rossberg. 2019. *Multiple per-module memories for Wasm*.

[33] Andreas Rossberg. 2019. *Proposal for adding basic reference types*.

[34] Jan Rüth, Torsten Zimmermann, Konrad Wolsing, and Oliver Hohlfeld. 2018. Digging into Browser-based Crypto Mining. In *IMC*.

[35] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *ICSE*.

[36] Natalie Silvanovich. 2018. *The Problems and Promise of WebAssembly*.

[37] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. 2019. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *WWW*.

[38] Sooel Son and Vitaly Shmatikov. 2013. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites.. In *NDSS*.

[39] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *USENIX Sec*.

[40] Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. 2018. Taint Tracking for WebAssembly. https://arxiv.org/abs/1807.08349. arXiv:1807.08349 [cs.CR]

[41] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy (SP 2013)*.

[42] Said Varlioglu, Bilal Gonen, Murat Ozer, and Mehmet Bastug. 2020. Is Cryptojacking Dead after Coinhive Shutdown?. In *ICICT*. IEEE, 385–389.

[43] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W Hamlen, and Shuang Hao. 2018. Seismic: Secure in-lined script monitors for interrupting cryptojacks. In *European Symposium on Research in Computer Security*. Springer, 122–142.

[44] Conrad Watt. 2018. Mechanising and verifying the WebAssembly specification. In *CPP 2018*. 53–65.

[45] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem. *POPL* (2019).

[46] Hui Xu, Zhuangbin Chen, Mingshen Sun, and Yangfan Zhou. 2020. Memory-Safety Challenge Considered Solved? An Empirical Study with All Rust CVEs. *arXiv preprint arXiv:2003.03296* (2020).