Towards Automatic Detection of Exploitable Vulnerabilities in WebAssembly Modules of Real-World Websites

DANIEL MAXIMIANO MATOS, Instituto Superior Técnico, Portugal

WebAssembly (Wasm) is a binary code format that is a compilation target for high-level languages, that allows them to run on a browser with near-native performance. Besides its multiple advantages due to its compact format, WebAssembly allows for the presence of vulnerabilities on Web programs, just like in *x86* binaries. Due to being a relatively recent technology, its presence in real-world websites was not yet properly studied, but it has the tendency to be more and more prevalent on the Web, as well as its vulnerabilities.

In this work, we present WebAF, a scalable web crawling framework enabling us to obtain webpages with WebAssembly submodules. It can collect a representative dataset of WebAssembly binaries that are useful for testing automated exploits on the found vulnerabilities. We also present WebAssault, our automatic exploitation framework, and show that it can generate simple proof-of-concept exploits.

1 INTRODUCTION

Due to JavaScript's poor performance when it comes to web-based applications, new languages have emerged to overcome this problem. Emerging languages also include new features and can give stronger safety guarantees. Some alternatives that have been developed to overcome this performance problem are Asm.js [1] and Google Native Client [2]. As of now, the most popular is WebAssembly [3], a compilation target for languages like C and Rust, which is a byte-code language intended for a portable virtual machine, *i.e.* uses a virtual Instruction Set Architecture (ISA). Also known as Wasm, WebAssembly can run with a near-native performance which makes it very attractive to port existing applications to it. Due to WebAssembly's benefits and popularity, the other two mentioned alternatives have been deprecated and are no longer maintained [4, 5].

Although WebAssembly code is sandboxed and features a stack-based virtual machine that is type-checked after every instruction, security vulnerabilities are still common and can sometimes be ported from the source code originally written in unsafe languages like C or C++ to the WebAssembly binary. Because WebAssembly is a virtual ISA, it can run on multiple platforms such as web browsers or server-side web containers. This means that undetected security vulnerabilities can transition to WebAssembly, e.g., buffer overflows can cripple many applications that depend on WebAssembly code running across various platforms. As a consequence of this, several tools have been developed in order to detect the possible vulnerabilities that can occur in this low-level language, being Wasmati [6] and WASP [7] the most relevant ones for a part of our work.

To better identify the possible vulnerabilities that the byte-code language may inherit from its source, the strategy used in other languages was also applied in Wasmati and WASP: recurring to *static analysis* and *symbolic execution*, respectively, to detect bugs in the code. On the static analysis side, we have Microsoft PREfast [8] that analyses C and C++ source files, JSLint [9] to check on JavaScript code, and RIPS [10] which checks the interactions between sources, sinks, and sanitizations in PHP files. Along the same vein as these tools, Wasmati employs static analysis to detect vulnerabilities in WebAssembly binaries. However, although these techniques are very efficient, they can generate a lot of false positives, requiring a great deal of manual effort to validate if the vulnerabilities flagged by the tool are true or false.

On the symbolic execution side, we have tools like Manticore [11] for smart contracts and binaries, WANA [12] for WebAssembly, and KLEE [13] for C/C++ and Rust, which are all symbolic execution engines that find bugs in their specific area. Likewise, WASP is a robust concolic execution tool for WebAssembly. The potential of these tools is that they can accurately prove the presence of a given vulnerability by generating an input that can trigger the vulnerable code path. However, a central drawback of symbolic execution, in general, is scalability and performance. In general, tools of this nature have to make simplifications in the code exploration algorithms in order to deal with path explosion problems. This is necessary because symbolic execution takes significant time to execute, and programs generally have many execution paths that need to be explored.

In work, we aim to investigate the possibility of combining the best of both worlds in order to improve vulnerability detection capabilities for WebAssembly code. In theory, we can prune out the false positives generated by Wasmati by manually checking if the detected vulnerabilities are exploitable or not. However, testing the exploitability of vulnerabilities may not be achievable by hand because they might be numerous, and exploits are time-consuming to write. To speed up this process, our idea is to employ automatic exploit generation, as first proposed in [14]. With this approach, we strive to simultaneously identify vulnerabilities and test their exploitability by software in a fully automated fashion. This way, developers can prioritize the vulnerabilities that must be patched first.

Toward this end, however, we need to obtain a representative dataset of WebAssembly binaries so that we can evaluate the effectiveness of our technique in finding security vulnerabilities. Unfortunately, as of the time of this writing, no such dataset has yet been collected by the research community. Moreover, there is no prior study focused only on investigating how prevalent Wasm binaries have become on the Internet. As a result, we cannot be sure how common WebAssembly vulnerabilities are in real-world web applications: if they are present on a small number of websites or if tech giants are also affected. Therefore, to test the viability of our vulnerability detection technique we must first collect such a dataset which in itself requires the development of a scalable web crawling framework for fetching webpages containing subcomponents written in WebAssembly.

1.1 Objectives

In light of the context clarified above, this thesis has three main goals:

- Obtain an up-to-date dataset of real-world webpages, containing HTML, JavaScript, and WebAssembly, allowing us

Author's address: Daniel Maximiano Matos, Instituto Superior Técnico, Portugal.

to perform multiple types of analyses and queries on the obtained data.
- Characterize the obtained dataset according to the presence of WebAssembly and JavaScript data types being used and shared between the two languages.
- Generate proof-of-concept exploits for WebAssembly vulnerabilities in the dataset, showing the viability of static analysis and symbolic execution as a precursor to automatic exploit generation.

1.2 Contributions

The contributions of this work are the following:

- A web analysis framework capable of navigating the web and scanning any Wasm binaries for security vulnerabilities they could contain.
- An up-to-date and representative web dataset.
- Proof-of-concept exploits generated by combining Wasmati and WASP.

2 RELATED WORK

In this section, we discuss the related work. We start by taking a look at some of the efforts to detect vulnerabilities in WebAssembly binaries. Then, we present some of the existing tools and methods that have been used for automatic exploit generation. Finally, we review some of the existing web crawlers and web datasets used in previous studies.

2.1 Vulnerability Scanning Tools for WebAssembly Programs

Apart from Wasmati and WASP, we survey other tools that have been developed to detect vulnerabilities specifically for WebAssembly and to help developers avoid mistakes when writing their code.

Wasabi. Wasabi [15] is a framework developed for dynamic analysis of WebAssembly programs through binary instrumentation. It allows us to develop analysis functions in JavaScript that will be called by the instrumented binary through the usage of hooks which will receive these functions as a callback. It uses on-demand "monomorphization" to handle polymorphic instructions, in order to avoid wasting memory by storing a lot of code that is not going to be used. It is useful for call graph analysis, dynamic taint analysis, crypto miner detection, and memory access tracing, as well as studying branch and instruction coverage and profiling basic blocks and instructions.

Wassail. In [16], the authors' proposed Wassail, a static taint analysis tool that uses compositional analysis. It works by creating a call graph of the module and then identifying the Strongly Connected Components (SCCs) of the generated graph. It generates an information flow summary of a function based on how the information propagates from the function's parameters and global variables to the function return value and globals after the execution. Next, it analyses the SCCs in topological order so that when a function is analyzed it will either use a function that has already been fully analyzed or another function in the same SCC and, in the latter case, it is rescheduled for analysis. In the end, we obtain the resume of the module.

Fuzzm. Fuzzm [17] is a grey box fuzzer, based on Google's AFL fuzzer[1] that works directly with WebAssembly binaries. In order to identify stack overflows, it instruments every function in the program with code that inserts a canary, on function entry, onto the current stack frame in linear memory, and that checks its integrity upon exit. For the heap, a chunk is surrounded by canaries that are going to be checked when it gets deallocated, which may not be ideal since overflows can occur without being detected if a chunk is not freed. These canaries work as oracles in order to guide fuzzing toward exploring all paths in a program and finding crashing inputs. The program is also instrumented such that all return instructions are rewritten to a jump to the code that validates the canary.

2.2 Automatic Exploit Generation

Detecting bugs in programs is very important to prevent unwanted behavior in them, especially security-critical ones. There are many tools whose goal is to find them. However, a program can contain a huge number of bugs, and deciding which ones to fix first may be a very complicated issue.

In order to fix this issue, Automatic Exploit Generation (AEG) [14] was proposed in 2011 as a way to automatically detect bugs, determine which ones are exploitable, and then generate an exploit that spawns a shell. However, this type of automatic analysis is not currently available for WebAssembly, being mostly applied to x86 binaries. AEG analyses the source code of a program and uses KLEE [13] as the backend for symbolic execution, with some modifications that implement their techniques and heuristics in order to find exploitable bugs more quickly and pruning from the space state inputs that are not large enough to cause an overflow. After finding a bug that crashes the program, the binary is instrumented and stops when the vulnerable function is called so that the stack can be analyzed in order to obtain the location of the return address and the location of the buffer containing the shellcode generated by AEG. These values are then passed to STP[2], a constraint solver, in order to solve the exploit constraints and concretize the input if the given constraints are satisfiable.

2.3 Web Crawling Systems

A web crawler, sometimes known as a web spider, is a computer software that searches the Internet by accessing websites. Crawling is primarily used to create an index of the websites visited by the tool. Web scraping can be performed by web crawlers because it is well interconnected with the crawling activity since it consists in extracting data from the accessed sites. Web scraping is widely performed by search engines, like Google [18] or Bing [19] since they need to store the contents of the pages to fulfill the queries made by the users. Due to this, both activities are almost always done simultaneously.

We now present some of the current web crawlers, their architectures, and their limitations.

curl. Curl [20] is a command line tool for transferring data. It supports multiple protocols, like HTTP, HTTPS, FTP, GOPHER, and SMTP. This tool allows a user to override the User-Agent parameter

[1] https://github.com/google/AFL
[2] https://github.com/stp/stp

it uses so that websites that block the default one think the request was made by a browser, for example.

Selenium. With Selenium [21], we are able to automate web crawling using a browser since it supports using all the major web browsers. Selenium is a browser automation framework that exposes an interface that can handle all browsers in a generic manner. Specific features and configurations of each browser can also be used, however, using them makes a solution less portable when changing testing with other web browsers. Since it uses a browser, it is able to handle all the requests made by JavaScript and load all the contents of the web page. This tool has an API available for a big number of languages, allowing it to be embraced by many other projects without much effort. It provides an interface to execute JavaScript commands and also to interact with the DOM and alert boxes.

OpenWPM. In [22], the authors propose a crawling framework, OpenWPM, which was developed to study 1 million websites in order to identify online tracking (either stateful or stateless) sites, among other privacy issues on the Internet. It supports the running of multiple browser instances providing higher speed than single browser solutions, however, has a more complex architecture. This tool is composed of three pieces: a task manager which distributes commands to the browser managers; browser managers which automate web browsers through Selenium [21]; and finally the data aggregator which instruments the requests of the browser. This is a very promising approach since it uses a leader process to distribute the work to the working process.

2.4 Web Datasets and Related Studies

Through the usage of web crawling techniques and tools like the ones we presented in Section 2.3, many datasets have emerged with multiple goals in mind. Also, many studies have been performed on the web and on the datasets generated by crawlers. Although these datasets may exist in big numbers, some are incomplete, do not represent the web nowadays, or are not suitable for all types of studies.

We now present some web page datasets and studies performed by the scientific community, as well as some of their limitations:

Minified and Obfuscated Code. In [23], the authors analyze minified and obfuscated JavaScript code. This code, due to the processes that have passed, is very hard to read by humans, so it is easy to hide malicious programs this way. It contained scripts from the top 1 hundred thousand websites of the Majestic Million service[3].

OpenWPM. The authors of OpenWPM [22] developed their tool to measure online tracking on the top 1 million websites of Alexa[4]. Since this study was performed in January 2016, WebAssembly was not even a minimum viable product, so it was not present on the Web or was in a very small amount, due to it only starting to gain traction in the last years. However, with the rise of Wasm, we can see more diverse ways of fingerprinting and tracking.

[3]https://de.majestic.com/reports/majestic-million
[4]https://alexa.com

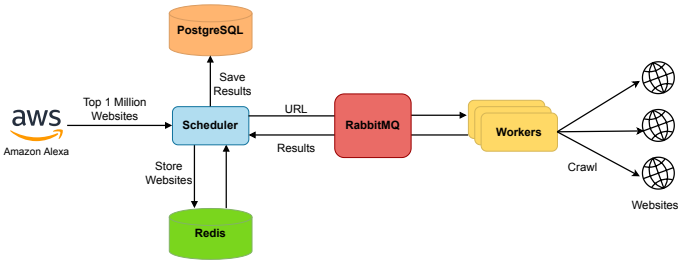


Fig. 1. Crawler high-level architecture

WasmBench. In [24], the authors present us with a dataset more focused on WebAssembly, in order to evaluate the usage of these binaries, their source languages and to test them for vulnerabilities. They have gathered them from multiple sources: crawling, source code repositories, package managers, and live websites (manually).

3 FRAMEWORK DESIGN

This section presents our approach to generating a dataset of web pages that will later be analyzed. To accomplish this, we developed our analysis framework named WebAF. We start by giving an overview of the framework, its division, and how its parts interact. Lastly, we present each of its parts and the challenges of their implementation.

3.1 Framework Overview

WebAF is the framework that we developed in order to generate and analyze a part of the web, seeking for websites featuring WebAssembly code and analyzing them. Our framework is composed of two main parts: a **crawler** and a **scraper**, each with a different role. The crawler will simply access the websites and store information about them, e.g., if they contain WebAssembly or make use of the WebAssembly JavaScript keyword. On the other hand, the task of retrieving the actual contents is left to the scraper, as well as the task of analyzing the web page contents.

In terms of their respective architectures, the crawler and the scraper are very independent of each other. The only point they have in common is the PostgreSQL database which is where the scraper obtains its data to process – the websites signaled by the crawler. The crawler functions in a microservices architecture, allowing it to scan a wider range of websites faster. We decided that the scraper would not need such a complicated architecture since it would scan only a smaller subset. The crawler and the scraper can be run in parallel, but we need to keep feeding the new data to the scraper because the crawler stores data in PostgreSQL, and the scraper queries MongoDB. Also, if following this approach, we need to be careful not to export the same data twice from PostgreSQL, because this will affect the results we will obtain in the end. Next, we present these two components in detail.

3.2 Crawler

Our web crawler uses Puppeteer [25] at its core and navigates through the web obtaining and making a quick analysis of the received data.

As we can see from Figure 1, our tool is loaded with the data provided by Amazon Alexa (which has recently been shut down by Amazon [26], however, its API is still available [27]) – the websites ranking based on traffic. It contained the top 1 million most accessed websites on the Internet, so it would be a good source for analyzing patterns in them. Those websites may be following the most recent trends to keep their users engaged and achieve better performance.

This list is then passed to the **Scheduler** component, which selects a batch to load to a cache database (**Redis** [28]). By default, this value is set to 1000 websites. The scheduler then retrieves an URL from the cache and puts it into a RabbitMQ [29] queue to be then consumed and accessed by a **Worker** node. If the number of inserted URLs in the queue is different from the number of worker nodes, it will retrieve more until this condition is met. Then, it will wait until a response is emitted from one worker. When this happens, it parses the response from the worker and stores it into **Postgres** [30], our persistent storage database. Lastly, it proceeds to get another URL from Redis and the cycle repeats itself until the batch reaches its end, in which case it will load the next batch, or there are no more links and the scheduler waits for the remaining responses and exits.

When a worker node is started, it first bootstraps Mitmproxy[31] with the available plugins and then starts the Puppeteer[25] component, also loading the plugins accessible. Then, it waits for an URL to be available so it can start working. As soon as one becomes available, the crawler accesses it, and when a response is emitted by the server, the proxy calls the plugins in order to perform the needed actions on the page. Then, the Puppeteer modules come into place to override anything that may be needed – in our case, the WebAssembly plugin overrides attempts to create or run a WebAssembly module. Only after these steps the response reaches the browser and is then executed.

After getting to the browser, we collect all the anchors that are present in the website and randomly select a subset to be analyzed next (the number of elements of this subset is four by default but can be altered by changing an environment variable). The worker node then proceeds to queue the selected subset on Puppeteer, to later be accessed. This process continues until we reach the maximum depth from the original website (by default the maximum depth is one website, but can also be changed) in which case we will not select more URLs and proceed to crawl URLs that are on the same level or on the levels above. In the end, the worker generates a response with all the URLs that served as an entry point for all the others, with the accessed URLs, and with the results signaled by each of the plugins in case they have made any work. This response is put into the result queue and is then handled by the Scheduler. Finally, we remove the URL from the job queue to prevent other nodes from performing repeated work. If a node does not perform this operation, after a while the URL will reappear in the queue to be processed, indicating that a worker node has failed and its work was lost.

Given the microservices architecture, our web crawler is an excellent candidate to be scaled in multiple machines, like in a Kubernetes (K8s) [32] cluster. Using this technology, we can launch containers at will without concern about which machine should they be deployed to and how they will communicate with other containers that may be placed in different physical devices after the cluster has already been set up. Since we had our own infrastructure with multiple powerful machines, we wanted to host a Kubernetes cluster there, so we could be able to run our tool.

The first step was to deploy virtual instances on the physical machines, so we started to build our Vagrant config with one virtual machine for each host, with all the CPUs of the host except four and with all of its RAM except 4GB.

The next step was to then deploy a Kubernetes cluster using these machines. A Kubernetes cluster has two types of machines:

- **Control plane.** Instance which controls the whole cluster, namely, scheduling of the containers, detecting and responding to events such as new deployments being made or resources being deallocated, among others [33].
- **Node**[5]**.** Instance where the containers are run and report events to the control plane. This type of instance runs on every machine of a cluster meaning a device in the cluster can both be a control plane and a worker node.

In addition, because we are using our own infrastructure – which is not a private cloud – and the need to deploy a production-grade cluster, we had to set up our cluster using `kubeadm` [34], a tool that facilitates the bootstrapping of a Kubernetes cluster.

For the container runtime, we decided to go with Docker [35] because it is the most widely used platform for running containers both for users and companies [36] and is easy to set up and install. After installing Docker, all that was needed to do was to install `kubeadm` [34], `kubelet` [37] and `kubectl` [38].

Lastly, we were missing the networking solution which is needed for the nodes and pods to communicate with each other (and consequently pods and containers) and to be accessible from the outside. We went ahead with Calico [39] because it has high performance, is open-source, and is already very mature - suitable for production.

At this point, all the requirements are met to proceed with the installation. After initializing the controller node, we must issue a command on the other nodes we want to join to the cluster. In order for this to work, the machines need to communicate with each other, so in the Vagrantfile we need to create a new interface for each one of them because by default they are behind a NAT and do not communicate.

We had our small cluster deployed in one virtual machine, so we needed to make a quick test on the crawler. It was deployed using our own Docker images and some Helm [40] charts. Helm charts are basically packages that we can easily install in our cluster, like containers that are very easy to configure. For the external technologies used by the crawler, we relied on Bitnami's [41] Helm charts as they are very well documented besides having some modifications over the original images. For our docker images, we needed to have a private docker registry and to accomplish this, we decided to simply host a docker registry as a container on the master node. After setting it up and pushing the images there, we were able to deploy the crawler on the Kubernetes cluster and make a quick run to check everything worked properly.

The solution we found to connect multiple virtual machines from different physical machines was to use the bridged mode [42] for the virtual machines, which assigns one IP to each virtual machine as if it was a physical node in the network.

[5]To avoid ambiguity, this type of node will be called worker node from now on

After solving all network-related issues, we still needed to manage how a cluster with multiple masters was set up. After reading the Kubernetes documentation [43], which mentioned the use of a load balancer, we chose to deploy a virtual machine that would only serve this purpose. We decided to go with HA Proxy [44] since it is very mature, is open-source, and has an excellent performance. Also, with a load balancer, we could be able to easily check if a master node is down and take action to bring it back up and running.

To sum up, our final deployment was composed of three Kubernetes masters, twenty Kubernetes worker nodes, and one load balancer, corresponding to a virtual machine each with a public IP. The VM deployment was configured with Vagrant and their provisioning with Ansible, using roles. The number of virtual machines and other parameters can be configured to adjust to the available devices. After the machines were ready, we just deployed our application in the cluster with a hundred crawler worker pods and started crawling the web.

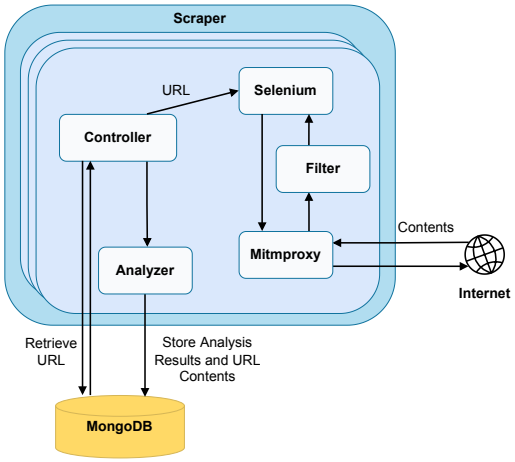


Fig. 2. Scraper low-level architecture

3.3 Scraper

Now we will present the scraper we used to collect the data that was marked by our crawler, and discuss some of the challenges we faced during its implementation.

Our scraper is a very simple tool that will work with the results of the crawler. It is developed in Python 3 [45] and uses MongoDB [46] as its storage. The data gets exported from Postgres using `psql` [47] with the `ROW_TO_JSON` function, present in this Database Management System, and is imported into MongoDB through `mongoimport` [48].

The scraper can be decomposed in the items we can see in Figure 2:

- **Controller.** Controls which websites to visit and passes the results to the Analyzer.
- **Selenium [21].** The browser controller which accesses the websites.
- **Mitmproxy [31].** Serves the same function as it did in the crawler: obtain all the requests and responses, allowing us to analyze them through the use of plugins.
- **Filter.** Is a Mitmproxy plugin that analyzes the responses obtained by the scraper.
- **Analyzer.** Perform an analysis of HTML, JavaScript, and WebAssembly (described in Section 4).

The scraper is initialized with multiple threads (the number can be configured on startup) and each one of them will scan a different website. The controller will start a Mitmproxy instance with the Filter plugin and set up its Selenium browser instance to make requests through the proxy. Then, it will contact the database, get one link to visit, and load it into the browser, to access it. When the responses are obtained, they are passed to the Analyzer, which will filter and accept only valid HTML, JavaScript, and WebAssembly. These responses are stored, hashed to only save repeated responses once, and then passed to the Selenium browser so that it can process the page and make any new requests.

When all is done, the Controller continues its job and forwards the findings to the Analyzer, which will perform a series of static analyses on the obtained files. Finally, both the responses and the results of the analyses are sent to the database where they are going to be stored and later processed.

All accesses to the database are mediated by a wrapper that we defined and they are synchronized, in order to avoid repeating saved requests. This wrapper is shared among all threads, and this is how we can guarantee that each thread gets a different URL to analyze.

We decided to simplify this part of our framework since it will only visit a subset of the URLs the crawler has accessed. The scraper runs on a single machine and can take advantage of multiple cores.

The biggest challenge in the implementation of this tool was whether to first obtain all the results and then analyze them, or analyze them first and then store them. The first approach allowed us to scan more websites more quickly but when it came to analyzing their contents, it became too slow, possibly due to retrieving big responses from MongoDB, from multiple collections. The whole process of retrieving the contents to analyze and then storing the results was just too slow, so we decided to put the analysis step in between accessing the website and storing its contents - the second approach. This showed itself the best way to tackle analysis since we already had the contents in memory and did not need to fetch them again from the database. It takes a little more to access the same amount of websites as the first approach but we can obtain the results right away and the overall progress is not that slow in comparison.

One of the other challenges we had to overcome when implementing the scrapper was how to handle the failures of Mitmproxy, because it fails silently, meaning that the Selenium browser would not receive an error, just an empty response for every request. The way we handled this was to kill the current Mitmproxy instance, close the browser instance and start the proxy in a new port. This would only affect one of the threads, leaving the others intact, since each one has a different instance of the proxy.

Finally, to handle failures from a thread (in case an exception occurred and we could not finish the request), we did not want its website to be forgotten and its results missed, so we implemented a cache in the system with the Least-Recently Used policy. The cache has a number of entries equal to the number of threads running.

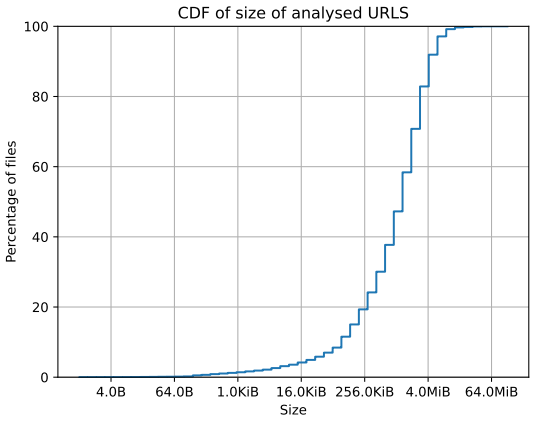
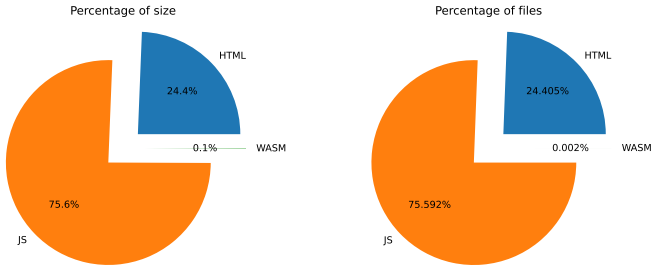


Fig. 3. Distribution of sizes of the analyzed websites.



(a) Percentage in size. (b) Percentage in files.

Fig. 4. Percentage of each resource in terms of number of files (left) and size (right)

Minimum size	0.0B
Average size	2.3MiB
Maximum size	85.8MiB
Percentile 25	444.4KiB
Percentile 50	1.4MiB
Percentile 75	3.2MiB
Total size	261.8GiB
Total URLs	116101

Table 1. General statistics of the dataset.

Using this approach, we guarantee that a website is only removed after being processed and its results saved. This cache is invisible to the controller and is handled in a wrapper to the database. It is a synchronization point, meaning only one thread at a time is able to request a website, allowing us not to repeat the same work multiple times.

4 DATASET

This section presents the methods that WebAF uses to be able to analyze the data it gathered from the web pages it connected to. We begin by giving an overview of our techniques and explaining in more detail what were the conditions when the dataset was generated. Then, we take a look at the WebAssembly analysis.

4.1 Collected Dataset and Generic Analysis

The scraper described in Section 3.3 performs multiple analyses on the obtained data. The type of analysis depends on the type of content retrieved from the web. Since we are only dealing with code, the generated dataset contains only HTML, WebAssembly, and JavaScript. To the best of our knowledge, this is the first study that investigates the relationship between JavaScript and WebAssembly only focusing on real web pages. This also constitutes the first study to evaluate the prevalence of WebAssembly in the wild. With this evaluation, we want to better characterize the web nowadays, regarding the usage of WebAssembly, its interaction with JavaScript, and how it is loaded, either directly from JavaScript or from HTML.

Since WebAF is separated into two independent parts, we chose to run the crawler and the scraper at two different times, because the available resources on our infrastructure are limited and due to time limitations, since the resources are shared with other researchers and need to be allocated for a well-defined period of time.

Before performing analysis on the specific resources of a web page, we wanted to get some generic insights on the obtained dataset. For that, we chose to study the size of whole websites (excluding images and other resources that were not relevant to our study), as it allows us to perceive how big (or small) the web is on average. It also enables us to evaluate the percentage of each type of resource.

In Figure 3, we can see that 50% of the webpages we visited are around 2MiB or less in size, which is a big number, since we are only measuring code, so the pages may take a little while to load. It also shows us that there are only a small amount of websites that are very small.

We also evaluated the prevalence of each resource (HTML, JavaScript, and WebAssembly) on each website, and obtained both their percentage relative to the size of each content (Figure 4a) and relative to the number of files found (Figure 4b). We can see that besides being very popular in the last years, Wasm only holds a very small percentage both in size (possibly due to being a binary format, so very compact) and also in the number of files. JavaScript holds the position of the most used resource on the web, allowing us to infer that most of the modern web is driven by this powerful language since there are more JavaScript scripts than HTML content. Most of the content nowadays is loaded through JavaScript, not being present in HTML statically.

We can see that we were able to obtain data from a total of 116101 websites, making a total of 261.8GiB[6]. The minimum size of a website we obtained was 0.0B, meaning we did not obtain any data from them, either due to encoding errors processing the returned data or the website did not respond, although it accepted our connection. Taking a look at Figure 3 in conjunction with percentile 25 from Table 1, we observe that there are a lot of websites with very low size, meaning those websites were signaled by the crawler so they have been active in the past. However, since the time of crawling was different from the time of scraping, some of them might not be available at the time of scraping due to Internet dynamism. We can

[6]This is the total amount of data we would store if we did not deduplicate data.

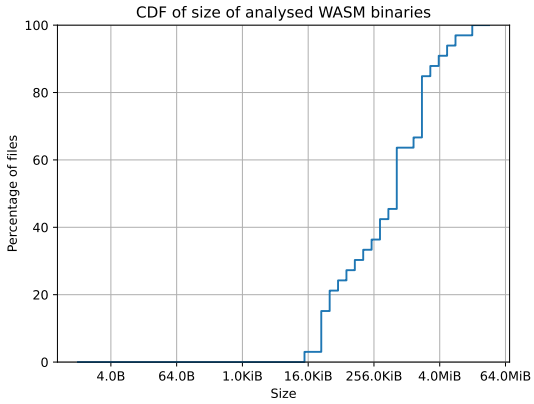


Fig. 5. Distribution of WebAssembly file sizes.

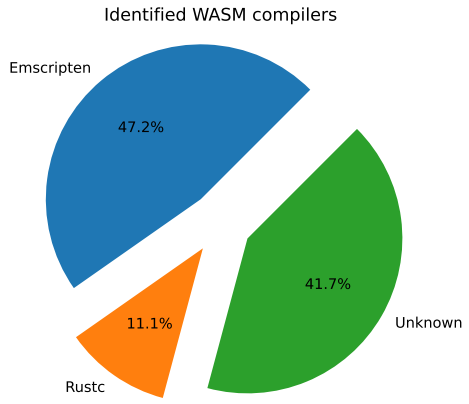


Fig. 6. Compilers of Wasm binaries in the dataset.

Minimum size	0.0B
Average size	1.9MiB
Maximum size	20.7MiB
Percentile 25	41.2KiB
Percentile 50	627.4KiB
Percentile 75	2.3MiB
Total size	70.0MiB
Total unique files	36
Maximum number of reuses	19

Table 2. WebAssembly statistics of the dataset.

see that half of the dataset is 1.4MiB in size, which we consider an amount similar to what we could the real world. Due to the time a website may take to load, developers want to ensure it does not take too much time but also that it transmits all the needed information for it to work properly, this is considered a reasonable value by us. Percentiles 50 and 75 are not too far from each other, meaning that a big range of websites falls into these two values. Given the maximum value of 85.8MiB, we can see it is too far from 3.2MiB, as we would expect since it would import an enormous amount of code and take many seconds (or even minutes) to finish a request, making a user give up on the website quickly. Next, we present our analysis of the websites' subcomponents.

4.2 WebAssembly Analysis

The main focus of our study is in WebAssembly binaries, so this analysis is the core of our investigation. We studied the amount of imported and exported functions as it constitutes the main point of interaction of WebAssembly code to the outside, mainly JavaScript code. We also wanted to take a look at the size of the binaries we were able to find.

In Figure 5 we can see that 20% of the total collected WebAssembly files is around 50KiB or less, meaning they are very small, as one would expect from a file in a binary format. We can see two spikes between 256KiB and 4MiB, meaning this is where the biggest number of WebAssembly binaries is located, which we consider being a range close to one of the real world, because a binary that is a lot less than 256KiB probably has very little function and may not be worth its existence, and one bigger than 4MiB is more likely to be a very important program, which handles big chunks of data, most likely a fully-featured application or a library, which are not very common yet.

The number of binaries we found was way below what we were expecting, but still, we believe 36 is a reasonable number for our dataset. The minimum size we observed was 0B, meaning we could not receive and analyze the response from the website. We can see that the average is almost three times the value of percentile 50, meaning that data is not well balanced, so we have fewer files that are bigger, as we saw in the CDF plot. So we also expect percentile 75 to be far from the maximum size we collected since data is very sparse in this area.

Taking a look at the averages of both parameters, they are higher than our expectations, which are around half of the presented values. We did not expect that WebAssembly programs would interact this much with the outside, opening space for the existence of vulnerabilities.

Finally, we also tried to identify the compilers that generated the binaries we collected. We performed this analysis in a very simple way, just to have a very high-level overview of the toolsets used currently. The results are presented in Figure 6 and we can see that Emscripten is the most popular one, as we would expect since most code bases still rely on unsafe languages like C/C++. Followed by it, there is a piece where we could not identify the compiler due to the very naive approach we took. Then, we have Rustc with a non-negligible percentage, allowing us to infer that there is also a small yet significant presence of Rust in the web, making the arising of vulnerabilities more complicated.

5 PROOF-OF-CONCEPT EXPLOITS

This section presents WebAssault, the tool that we envision to automatically create exploits in JavaScript that trigger vulnerabilities in WebAssembly code. It is an initial step towards generating complex exploits for vulnerabilities present in WebAssembly binaries.

We begin with an overview of the design of our tool (Section 6). In Section 6.1, we present some of the challenges we faced. Lastly, we discuss implementation details (Section 6.2) and evaluate the tool with a handmade exploit (Section 6.3).

6 OVERVIEW

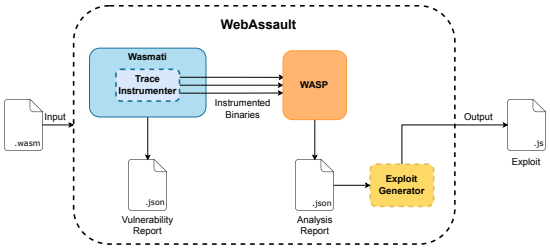


Fig. 7. System architecture.

Figure 7 sketches the architecture of our tool, WebAssault, which aims to generate exploits for WebAssembly vulnerabilities. Dashed lines represent components developed by us whereas the solid ones represent already-existent components. It is composed of multiple stages with different purposes:

(1) *Wasmati:* A WebAssembly binary is fed into Wasmati for it to create a CPG which will be queried for the common vulnerabilities. A report (*vulnerability report*) with the findings is then generated in JSON format showing the function(s) where the bug(s) is(are) present, the type of the vulnerability(ies), and a small description of it(them).

(2) *Trace Instrumenter:* This component receives the WebAssembly binary and for each vulnerability reported, we trim the possible paths that WASP would try in order to prevent it from exploring paths that will not lead to the vulnerability we are testing. We need access to the binary in order to patch it with instructions that guide WASP to the possibly vulnerable path. Multiple instances of instrumented binaries may be generated, depending on the number of vulnerabilities detected by Wasmati.

(3) *WASP:* Executes the instrumented binary provided by the trace instrumented. WASP's concolic execution is guided to execute the program along the path that was selected in the previous step. If the path leads to a real vulnerability, we will get as output a JSON report (*analysis report*) containing the input value(s) of the program that can trigger the vulnerability, and consequently the information needed to generate a simple exploit.

(4) *Exploit Generator:* The last component of our system uses as input the analysis report produced by WASP in order to generate the exploit that triggers this vulnerability in the code. If the vulnerability can be triggered, it outputs an exploit written in JavaScript that when executed will also take advantage of it. Otherwise, it will simply disregard this vulnerability, as WASP could not find an execution path so it will not receive any meaningful input.

6.1 Implementation Challenges

For our tool to fulfill the task of automatically generating exploits, using Wasmati and WASP, meaning, static and dynamic[7] analysis, respectively, we need to implement the modules we pictured as dashed in Figure 7. We aim at generating multiple binaries based on the vulnerabilities Wasmati identifies. The main challenge here is to trim all of the possible paths on a binary, for it to follow only the one intended, as we will explain in Section 6.1.1. Because we are dealing only with the WebAssembly file it may also be hard to find the exact type of arguments that are being passed on to a Wasm library. To handle this issue we need to analyze the code flow in JavaScript, which we discuss in Section 6.1.2.

6.1.1 Trimming Paths in the Concolic Execution. Concolic execution works by exploring all the possible paths a program can go through, using both symbolic and concrete executions. The main challenge of this approach is the well-known path explosion problem. However, in our work, since we have the trace from Wasmati, we want to test only a single path at a time, so we need to cut all the other ones in order to obtain the inputs we need in a timely manner. Also, because WASP extends the WebAssembly syntax, we have support for more instructions that can help us guide concolic execution in the way we desire.

Specifically, our idea to guide WASP's concolic execution is as follows. Since we have access to the CPG of the module we are testing, we are able to get the execution paths by analyzing the control flow graph, allowing us to trim the ones we do not want to test by patching the WebAssembly binary with instructions that will make the path condition automatically *false* in these zones.

To cut a path, we can simply add the instructions `i32.const 0; sym_assume`, because we need the value on top of the stack to be **0** and we also need to tell the concolic execution engine to add this value to the path condition. These instructions are inserted in the branches that we know are not needed for our execution, allowing us to explore the path we need quicker. The path-trimming technique just described will be implemented by WebAssault's Trace Instrumenter.

6.1.2 Generating Symbolic Inputs from JavaScript Calls. To determine the correct symbolic type of a variable, we should be able to know how it is declared, or in the case of a function parameter, how it is called. Since a high percentage of WebAssembly code is used on the web, it needs to be called by JavaScript code in order to run. Unfortunately, as of today, WASP cannot determine correctly the proper symbolic type to assign to symbolic variables. In some cases, this limitation is due to the fact that they may be a reference to variables in WebAssembly's linear memory (arrays). Since variables are only referenced using an index, it is quite hard to determine their length. Therefore, our solution is to analyze the data type when the function is called in JavaScript, where we can clearly determine the type of each variable being used.

To showcase our approach we consider an example of an array being passed from JavaScript to be sorted by a WebAssembly module. By analyzing only the WebAssembly code, we would not be able to determine the length of the variable that we use as parameter. In contrast, since we have access to the JavaScript calling the function,

[7] By dynamic analysis in this context we mean concolic execution.

we can clearly see that it's type and size. With this new information, we can create a new function in the WebAssembly module, that will create a symbolic array of the corresponding elements and will later be recognized as so by WASP. This new array would then be used as an argument to call the function called by JavaScript, enabling us to more accurately determine if in fact, a bug is present in the code or if it is just a false positive reported by the Wasmati tool. If there are multiple calls to the same function and they use arrays of different sizes, we run WASP multiple times for the same path, but with different symbolic types in order to better test all the possible values we can get.

With this approach, we could inject code in the original WebAssembly binary to explicitly assign symbolic types to variables and then call the function we want to analyze with them.

6.2 Implementation

We decided to start with the instrumentation of the binaries. To implement the instrumenting behavior on Wasmati, we created another visitor which allows us to write Wasm files in text format. We also defined a new data structure that could encode the execution flow paths that can reach the vulnerabilities. This new structure stores a list of CPG nodes in reverse order, representing the path from the vulnerability to all the possible sources that can trigger it. We achieved this by creating a path with just a single node - the one where the vulnerability is identified - and performing a modified Breadth-First Search (BFS) algorithm on it.

This BFS runs from the vulnerable node, in reverse order, until we reach the `Module` node, which is used in the CPG to represent the root of the graph. If a path we were taking did not reach this node, it is discarded. To obtain all the possible paths to the vulnerable node this BFS maintains a queue of paths that need to be expanded and it expands each one of them separately. Whenever we reached a branching point or a function that could be called by JavaScript (*i.e.* exported by the module), we would clone the path and add the source nodes of the branching point to the original path and its copy. One of the nodes would continue to be expanded and the other one would be appended to the queue.

After this, the found execution paths are returned and converted to JSON so that the relevant labels of the nodes can be printed to the vulnerability report generated by Wasmati and allow the user to make a manual inspection if desired.

Finally, we create a directory named `instrumented` where we will place the instrumented WAT files. In this phase, we receive the paths we obtained from all the vulnerabilities and run the previously mentioned visitor on each one of them. The `wat-instrumenter` is fed with the CPG and a path. It starts traversing the graph by the root module, trying to reconstruct a file very similar to the one that generated it, in text format. Whenever it encounters a function node it tests if it is the function where the vulnerability was found, if it was it will trim the paths inside of it, otherwise it will just print the regular function. We followed this approach because other functions in the module can have multiple execution paths and some of them may be useful to our path, but since we could not decide on which ones were and which did not, we took a conservative perspective and allowed them to be fully present. This allows us to decrease the number of false positives WASP could issue due to calculating function outputs that could not be generated by a given function, but since we would have defined them as symbolic, they would be valid for WASP. As stated in Section 6.1.1, we were able to prune branching conditions using the described technique where the path should not be taken.

The instrumentation functionality can be activated for Wasmati using the flag `-instrument`, which will activate the `-native` flag since we need to have the native queries determine vulnerabilities in order to instrument the file accordingly.

We applied the process described earlier on buffer overflow vulnerabilities with static allocation size and we were also able to generate functional WebAssembly binaries. To generate the test case for each of the runs, we used C and then compiled them with WASP-C. However, since WASP-C compiles a file and already fills it with the hooks needed for WASP, we needed to perform a modification if we wanted the file not to receive these modifications because Wasmati would not be able to read the instructions WASP uses. After doing this, we ran Wasmati on the generated file and it gave us the vulnerability we wished for and an instrumented file. After passing this instrumented file through the post-processing WASP needs, we were able to make WASP find an input that would trigger the vulnerability.

6.3 Evaluation

We came up with the very simple buffer overflow where there is a function named `vuln` that receives an argument. In this function, we dynamically allocate a buffer with 256 bytes and then we check if the argument we received was 1337. If it did not correspond to this number, everything is fine, however, if the argument was 1337, then we would write 10000 bytes on a buffer that can only fit 256. We can see that WASP was capable of determining the correct input and so, we could generate an exploit that would be `vuln(1337)`, which would call the Wasm module's function.

Because of all the limitations of WASP and Wasmati, we only managed to generate PoC exploits for WebAssembly, so we are not confident enough to test WebAssault on real-world binaries. However, we showed that it is possible to apply this technique to Wasm using static and dynamic analysis together.

7 CONCLUSIONS

WebAssembly is a new format that is becoming more present in the modern web by running code more efficiently in the browser and also by being very compact. To the best of our knowledge, no study has been performed that evaluated the prevalence of WebAssembly in the wild, so there also does not exist a dataset comprised of only wild Wasm binaries. We also are not aware of any tool capable of creating automatic exploits for vulnerabilities present in WebAssembly binaries.

We evaluated the web by generating a representative dataset that was later analyzed to the relative presence of different file types and other metrics specific to the file type. We saw that besides being popular, Wasm's footprint is almost negligible on the current web, but we hope to see an increase in the coming years. We proposed WebAF as a web analysis framework and verified it could scale very well, allowing for multiple parallel scans to occur.

We propose WebAssault as the first framework capable of generating Proof-of-Concept exploits for WebAssembly vulnerabilities using static and dynamic analysis. We were capable of generating a functional exploit for a handmade vulnerability.

7.1 Future Work

In the future, we would like to improve the crawler present in our framework, allowing it to be more easily deployed, and study how it could scale to more than one scheduler, enabling it to improve its scalability. Due to time constraints, we could not address all of the challenges we intended to in WebAssault, so, we would like to solve the JavaScript's types analysis challenge. One thing that could be improved in this framework is the Exploit Generator because it is not finished so it is not fully automatic. Since we were only able to test the exploit framework with a very simple example, we would like to test it with more complex data, as well as overcome some of the limitations of WASP and Wasmati, so we could use a wider range of tests and could work with more vulnerabilities. Finally, we would like to automate the whole WebAssault framework, as it requires us to run the programs manually.

REFERENCES

[1] D. Herman, L. Wagner, and A. Zakai, "asm.js Website," http://asmjs.org/spec/latest/, 2014, accessed: 2021-12-15.

[2] Google Inc., "Native Client," https://developer.chrome.com/docs/native-client/, accessed: 2021-12-15.

[3] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[4] Mozilla, "asm.js - Game development," https://developer.mozilla.org/en-US/docs/Games/Tools/asm.js, 2021, accessed: 2021-12-15.

[5] Google Inc., "Goodbye PNaCl, Hello WebAssembly!" https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html, 2017, accessed: 2021-12-15.

[6] P. Lopes, "Discovering security vulnerabilities in webassembly with code property graphs," Master's thesis, Instituto Superior Técnico, https://fenix.tecnico.ulisboa.pt/cursos/meic-a/dissertacao/846778572212698, 2021.

[7] F. Marques, "Robust symbolic execution for webassembly," Master's thesis, Instituto Superior Técnico, https://fenix.tecnico.ulisboa.pt/cursos/meic-a/dissertacao/1128253548922792, 2021.

[8] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fähndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy, "Righting software," *IEEE Software*, 2004.

[9] D. Crockford, "Jslint: The javascript code quality and coverage tool," https://www.jslint.com, accessed: 2021-12-21.

[10] J. Dahse and T. Holz, "Simulation of built-in php features for precise static code analysis," in *NDSS*, 2014.

[11] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, 2019.

[12] D. Wang, B. Jiang, and W. K. Chan, "Wana: Symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection," *arXiv*, 2020.

[13] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*, 2019.

[14] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "Aeg : Automatic exploit generation," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2011.

[15] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2019.

[16] Q. Stievenart and C. D. Roover, "Compositional information flow analysis for webassembly programs," 2020.

[17] D. Lehmann, M. T. Torp, and M. Pradel, "Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly," *arXiv*, 2021.

[18] Google Inc., "Google," https://www.google.com, accessed: 2022-09-02.

[19] Microsoft, "Bing," https://www.bing.com, accessed: 2022-09-02.

[20] Open Source Software, "curl," https://curl.se, accessed: 2022-09-02.

[21] Software Freedom Convervancy, "Selenium," https://www.selenium.dev/, accessed: 2022-08-22.

[22] Englehardt, Steven and Narayanan, Arvind, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016.

[23] Skolka, Philippe and Staicu, Cristian-Alexandru and Pradel, Michael, "Anything to hide? studying minified and obfuscated code in the web," in *The world wide web conference*, 2019, pp. 1735–1746.

[24] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," in *Proceedings of the Web Conference 2021*, 2021.

[25] Google Inc., "Puppeteer | Puppeteer," https://pptr.dev/, accessed: 2022-08-11.

[26] Amazon, "We retired Alexa.com on May 1, 2022," https://support.alexa.com/hc/en-us/articles/4410503838999-We-retired-Alexa-com-on-May-1-2022, accessed: 2022-08-11.

[27] ——, "We will be retiring the Alexa.com APIs on December 15, 2022," https://support.alexa.com/hc/en-us/articles/4411466276375, accessed: 2022-08-11.

[28] Redis Ltd., "Redis," https://redis.io, accessed: 2022-08-11.

[29] VMWare Inc., "Messaging that just works — RabbitMQ," https://rabbitmq.com, accessed: 2022-08-11.

[30] The PostgreSQL Global Development Group, "PostgreSQL: The World's Most Advanced Open Source Relational Database," https://www.postgresql.org, accessed: 2022-08-11.

[31] mitmproxy, "mitmproxy - an interactive HTTPS proxy," https://mitmproxy.org/, accessed: 2022-08-11.

[32] The Linux Foundation, "Production-Grade Container Orchestration," https://kubernetes.io/, accessed: 2022-08-11.

[33] ——, "Kubernetes Components | Kubernetes," https://kubernetes.io/docs/concepts/overview/components/#control-plane-components, accessed: 2022-08-16.

[34] ——, "Installing kubeadm | Kubernetes," https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm, accessed: 2022-08-16.

[35] Docker Inc., "Home - Docker," https://www.docker.com/, accessed: 2022-08-16.

[36] Datadog, "10 Trends in Real-World Container Use | Datadog," https://www.datadoghq.com/container-report/#8, 2021, accessed: 2022-08-16.

[37] The Linux Foundation, "kubelet | Kubernetes," https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet, accessed: 2022-08-16.

[38] ——, "Command line tool (kubectl) | Kubernetes," https://kubernetes.io/docs/reference/kubectl, accessed: 2022-08-16.

[39] Tigera Inc., "About Calico," https://projectcalico.docs.tigera.io/about/about-calico, accessed: 2022-08-16.

[40] The Linux Foundation, "Helm," https://helm.sh, accessed: 2022-08-17.

[41] Bitnami, "Bitnami: Packaged Applications for Any Platform - Cloud, Container, Virtual Machine," https://bitnami.com, accessed: 2022-08-17.

[42] Oracle, "Chapter 6. Virtual Networking," https://www.virtualbox.org/manual/ch06.html#network_bridged, accessed: 2022-08-20.

[43] Google Inc., "Creating Highly Available Clusters with kubeadm | Kubernetes," https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability, accessed: 2022-08-20.

[44] HAProxy, "HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer," https://www.haproxy.org, accessed: 2022-08-20.

[45] Python Software Foundation, "Welcome to Python.org," https://www.python.org, accessed: 2022-08-22.

[46] MongoDB Inc., "MongoDB: The Developer Data Platform | MongoDB | MongoDB," https://www.mongodb.com, accessed: 2022-08-22.

[47] The PostgreSQL Global Development Group, "PostgreSQL: Documentation: 14: psql," https://www.postgresql.org/docs/current/app-psql.html, accessed: 2022-08-22.

[48] MongoDB Inc., "mongoimport — MongoDB Database Tools," https://www.mongodb.com/docs/database-tools/mongoimport, accessed: 2022-08-22.