Review of selected modern cryptojacking detection techniques

(Przegląd wybranych współczesnych technik wykrywania cryptojackingu)

Krzysztof Chmiel

Praca inżynierska

Promotor: dr Paweł Rajba

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

12 czerwca 2024

Abstract

This work covers a review of a few modern detectors, aiming to counter the threat of cryptojacking. It spread across the internet in the recent years, due to prevalence of cryptocurrencies and increasing value of their markets. It led to the development of more and more sophisticated techniques to take advantage of 'illicit mining'. 'Illicit', because in order not to worry about high consumption of computational resources associated with mining, victims' hardware is being exploited, instead of actor's own, minimizing the cost of the process. In the typical scenario user visits a website, which mining payloads have been attached to and they leverage users computers through their browsers. Consequently, many solutions have been introduced, all sharing one characteristic - black-listing detection. It worked well at the beginning, but couldn't keep up with the evolution of payloads and their constantly expanded evasion capabilities in the long run. Therefore, to catch up, in the recent years researchers dropped the old approach and have been working to invent more advanced mechanisms to detect cryptojacking. In this work the efficiency of these results is evaluated. At the beginning the terminology and essential concepts are introduced. Then each detector is described, covering its setting up process, features and evasion techniques. At the end improvements are proposed.

Praca ta koncentruje się na analizie wybranych, współczesnych wykrywaczy "cryptojackingu", ataku który rozpowszechnił się w ostatnich latach, z powodu dużej popularności kryptowalut oraz ich stale rosnącej wartości na giełdach. Spowodowało to rozwój coraz to bardziej zaawansowanych technik "nielegalnego kopania". "Nielegalnego", ponieważ aby proces był opłacalny i przestępcy nie musieli martwić się wysokim zużyciem zasobów przy kopaniu, aktorzy nie używają własnego sprzętu, wykorzystują do tego komputery niczego nieświadomych użytkowników. Najczęściej atak odbywa się przez strony internetowe, do zawartości których dołączany jest złośliwy kod, który wykonuje się w przeglądarce użytkownika po odwiedzeniu takiej witryny. Aby można się było jakoś obronić powstało wiele rozwiązań, które niestety miały jeden wspólny mankament - opierały się na idei listy zakazanych słów tzw. "black-listing". O ile na początku to wystarczało, to wraz z rozwojem technik omijania stosowanych przez aktorów okazało się, że metoda ta jest niewystarczająca. W związku z tym, w ostatich latach naukowcy pracowali nad rozwiązaniami, które dorównywałyby wyrafinowaniem stronie atakującej. Rezultaty tejże pracy oraz idąca za tym skuteczność w walce z złośliwym oprogramowaniem zostaną poddane ocenie w tej pracy. Na początku wprowadzona zostanie niezbędna terminologia oraz koncepty. Następnie każdy z detektorów zostanie dokładnie opisany, włączając jego właciwości oraz jak go odpalić, obejść i poprawić.

Contents

Chapter 1

Introduction

1.1 Motivation

The idea to evaluate the efficiency of modern cryptojacking detection techniques is related to the previous work [1], which focused on similar area, because it involved cryptomining as well, however it assessed the detectors in a form of browser extensions only. The methods to be evaluated in this work, are based on the recent Systematization of Knowledge (SoK) paper [2], summarizing the current levels of knowledge and comparing results of various solutions in one work. There is a table in this SoK paper:

TABLE 3. THE LIST OF OPEN-SOURCE CRYPTOJACKING MALWARE DETECTION IMPLEMENTATIONS.

Ref	Implementation Link	Description	Last Update
CMTracker [119]	https://github.com/deluser8/cmtracker	code	Sep 21, 2018
Minesweeper [112]	https://github.com/vusec/minesweeper	data and code	Mar 17, 2020
OUTGUARD [116]	https://github.com/teamnsrg/outguard	data and code	Sep 6, 2019
SEISMIC [122]	https://github.com/wenhao1006/SEISMIC	code	Sep 10, 2019
Retro Blacklist [143]	https://github.com/retrocryptomining/	data and code	Jul 16, 2020

which shows out of all the analysed projects, the ones which are open-sourced, so potentially could be assessed.

1.2 Overview

The following, opening chapter will cover the basic concepts, definitions and technologies, as without them the understanding of the main part of this work may be limited. Beginning from Chapter 2, the primary scope of this work is covered. Five existing solutions ([3], [4], [5], [6] and [7]) published similarly in time, around 2019 accurate to within one year, are compared. All of them have one thing in com-

mon, they try to address an existing problem of cryptojacking and lack of adequate solutions, matching the sophistication of the malicious adversaries' programs.

1.3 Cryptocurrencies

Since the 2009 release of Bitcoin, a lot has changed in the world of cryptocurrencies, people got familiar with the existence of alternative currencies, but what exactly are these cryptocurrencies?

1.3.1 Overview

Cryptocurrency is a digital currency designed to work as a medium of exchange through a peer-to-peer computer network that is not reliant on any central authority, such as a government or bank, to uphold or maintain it. It is possible because of the technology cryptocurrencies are based on, called `blockchain`, which is a distributed ledger, working as a public financial transaction database. In other words, all the information about who sent the "money" to who, and how much or who is the owner of X amount of the currency, is inscribed as a record in this ledger and cannot be modified (cannot be updated, only appended). Blockchain consists of the blocks, which include this data and these blocks are linked one to another - making a chain. It is the cryptographic hashes that link them. Each of valid blocks contains a solution to the proof-of-work puzzle. It means a calculated hash proving, the computational heavy work has been made made in order to evaluate it. These hashes contain the information about what is the previous block in the chain, the information about transactions in the current block and wallet address (to make sure the rightful owner only can transfer the funds). It is essential to understand that it is fast and easy to check the correctness of the calculated hash, but it takes a lot of time to actually calculate it. The complexity of proof-of-work algorithms prevent malicious adversaries from adding fake blocks to the ledger, as it takes resources and time to correctly compute the new block's properties, including the information about the previous block. Having said that, the blockchain is maintained by miners. They collect the transaction data from the peer-to-peer network of miners, validate it and insert into the blockchain. When a miner successfully add a block, meaning the essential parameters turned out to be correct, the network rewards it with a piece of currency (prize for extending the blockchain, meaning there is more space to include new data). Later the miner shares it with the user who actually resolved the puzzle and calculated the hash correctly. This is called mining - adding new blocks and getting compensated for it.

1.3.2 Monero

At the beginning of this section Bitcoin was mentioned, as a precursor of cryptocurrencies. Later many more currencies have been successfully created and still are, but one in particular is interesting and stands out - Monero. The reason being, Monero is more private than its counterparts and it is essentially impossible to track down the owner of the funds or the author of the transaction, making payments with its use anonymous. There is a lot of people benefiting greatly from this feature e.g. darknet drug dealers or any other vendors selling illegal goods over the internet. In case of this work, it will be the only currency I took into consideration, because of its other very desirable feature - equality. What it means, is that the way Monero's proof-of-work algorithm is designed is to not give an upper hand while mining on ASICs - customized integrated circuits, made specifically to mine cryptocurrencies. In other words, Monero mining is optimized for general-purpose CPUs and it is unfeasible to use super-powerful mining machines, as they don't make the process more profitable or faster. The proof-of-work that Monero used throughout the years up until the end of 2019 was Cryptonight. It is the algorithm reoccurring from the beginning till the very end of this work, as topics covered here are based on the papers from the cusp of an eras (around 2018-2019), before Monero switched to RandomX permanently.

Disclaimer: As of 2024, the browser-mining is not as popular as it used to be 6 years ago, because Monero uses RandomX (not Cryptonight anymore) as its main proof-of-work algorithm and it cannot be implemented with web technologies. However, it is still possible to use Cryptonight in the mining payloads, hence the topic of this work is still relevant.

1.3.3 Cryptonight

1. The first stage of the Cryptonight algorithm is all about hashing the initial data with SHA-3, which is Keccak algorithm. First 32 bytes of the output serves as an AES key and expand to 10 round keys. Bytes 64 to 191 are split into 8 blocks of 16 bytes, each of which is encrypted in 10 AES rounds with the expanded keys. The result, a 128-byte block, is used to initialize a scratchpad placed in the memory (cache level L2 or L3, depending on the scratchpad size and CPU cache sizes) through several AES rounds of encryption. *[Note: There are a few different variants of the algorithm. Most often the difference is in the size of the scratchpad used, but it will not really matter that much in this work's considerations.]*

2. Second stage is where the computational heavy work has to be done. At the beginning two variables are created from the XOR-ed bytes 0–31 and 32–63 of Keccak's final state. Later the main loop is repeated 524,288 times and it consists of a sequence of reads and writes from and to the scratchpad.

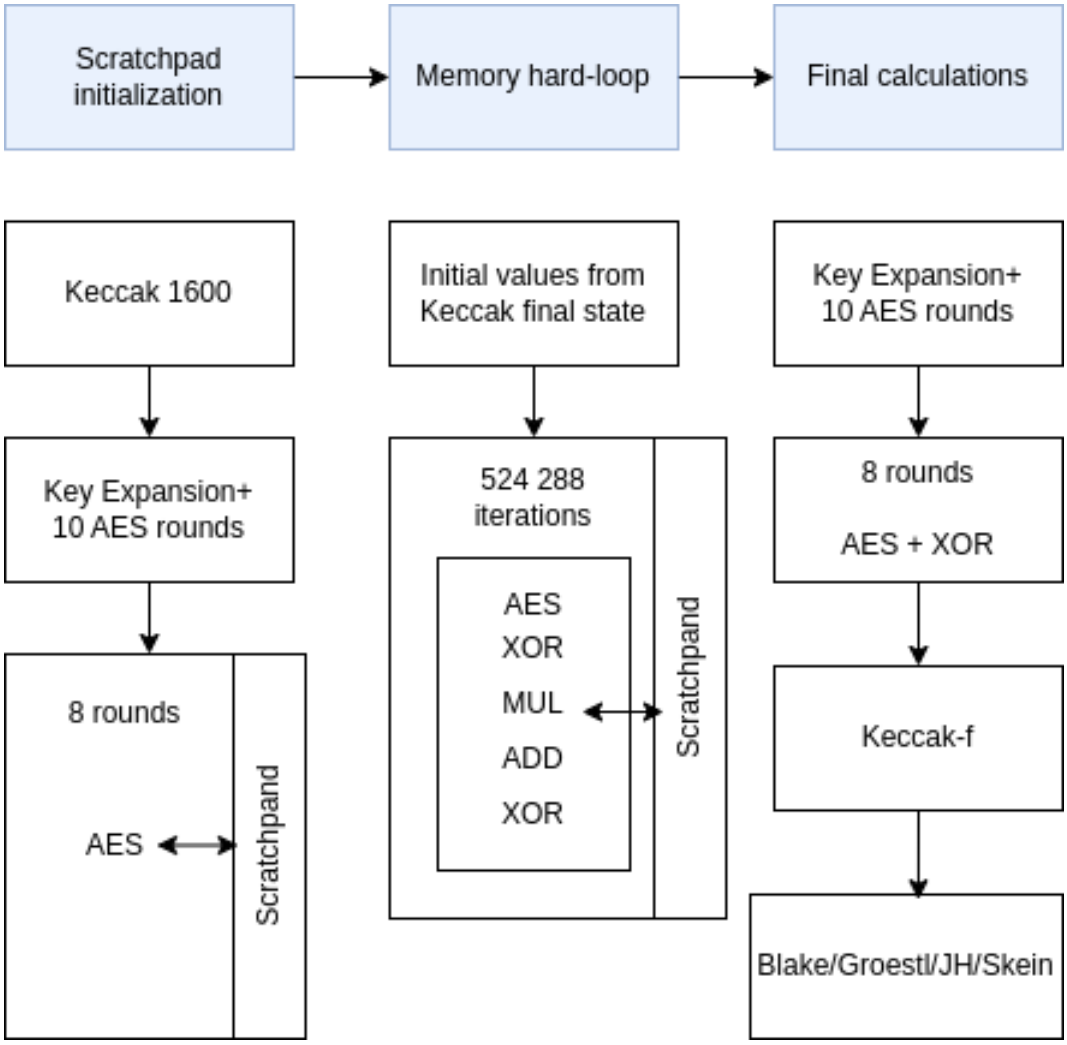


Figure 1.1: Cryptonight (based on Monero source code and Minesweeper paper)

3. Finally the last step begins with the expansion of bytes 32–63 from the initial Keccak's final state into an AES-256 key. Bytes 64-191 are used in a sequence of operations that consists of an XOR with 128 scratchpad bytes and an AES encryption with the expanded key. The result is hashed with Keccak-f (which stands for Keccak permutation). Eventually the lower 2 bits of the final state are then used to select a final hashing algorithm to calculate the result. There are 4 options: Blake, Groestl, JH and Skein. (All four of them were SHA-3 competition finalists, but lost to Keccak. The competition was organised by American NIST (National Institute of Standards and Technology), to create a new hashing standard. It happened back in 2012.)

1.4 Browser model

1.4.1 CDP

Reading this work one must be aware that the detectors described later in the document, in most cases, rely on advanced features of the browser. To be exact one specific browser - Google Chrome. It allows the user to interact with the runtime, through remote debugging interface. The communication happens by default on port `9222` as a local TCP connection. There are a few implementations of this interface, but the prominent one is called `chrome-remote-interface` and could be operated in Node environment. Under the hood it uses `Chrome DevTools Protocol` (in this work often shortened to `CDP`), which is an in-built Chromium feature to instrument, inspect, debug and profile the browser. The default UI of `CDP` (called Chrome DevTools) everyone should be already familiar with, because it is known in most browsers as `developer tools`.

What is important to introduce here, is what is the structure of the `CDP`. Having said that, instrumentation is divided into a number of **domains** (Page, Debugger, Network, Tracing, Profiler etc.). Each domain defines a number of methods it supports and events it generates. Both methods and events are serialized JSON objects of a fixed structure (however the structure may change over the years). Operations performed with use of `CDP` domains will take a crucial role in this work.

1.4.2 Other Web APIs

Having `CDP` covered, there are two other interfaces, which I believe have to be mentioned in the introductory chapter, to make sure the following contents will be understood correctly.

- **WebSocket** - makes it possible to open a two-way interactive communication session between the user's browser and a server. With this API, the user can send messages to a server and receive event-driven responses. It takes a significant role in how victim's browser communicate with the mining pool (through proxy server) to send and receive jobs and respective results. (more about it in the section `1.6.2`)

- **WebWorker** - is a simple mean for web content to run scripts in background threads. The worker thread can perform tasks without interfering with the user interface (therefore it doesn't block the DOM). Once created, a worker can send messages to the JS code that created it by posting messages to an event handler specified by that code and vice versa.

1.5 WebAssembly

1.5.1 Overview

There could be no efficient mining in the browsers, if it was not for WASM. Introduced in 2017, it is a fairly new web technology, which allows any `C` or `C++` application to be easily converted to a binary instruction format and executed in the browser at almost native speed, as it is statically typed and doesn't have to be parsed and analysed to be re-optimised by the JIT engine at the runtime once again (it happens only once during compilation). This lack of significant overhead, is what makes WASM so attractive in terms of mining. Having native code performance in browser is essential for malicious adversaries, when planning cryptojacking campaigns and their profitability. WebAssembly modules are used as a compiled version of proof-of-work algorithm library (Cryptonight) and hashing function inside the module (with all its dependencies) is executed every time there is a need to solve the job sent by the mining pool.

1.5.2 Language structure

One of the interesting features of WASM is its ability to be represented as a text format - WAT. Both binary and text formats are very easy to switch between as the authors of WebAssembly created `WABT` (The WebAssembly Binary Toolkit), which is equipped with `wasm2wat` and `wat2wasm` tools, which enable the user to change the binary format to the human readable code and back. This feature is important to know of, because it will be used in this work, by some detectors to analyse the scripts and their maliciousness.

Having said that, this work is not a place to understand WASM fully, however it feels like some basics of the language semantics should be covered.

- Each module begins with keyword `(module ...)` and all of the code ends up inside the brackets (in place of `...`).

- The next first few lines describe function types sections, which means number and type of arguments function takes and non-obligatorily a return value, meaning if there is any - `(type (;0;) (func (param ...))`. These arguments usually are the data types such as: `i32, i64 or f32, f64`, meaning 32-bit or 64-bit integers and floats. Many functions can be of the same type.

- Later, the next lines describe imports section - `(import ...)`. This means the data that will be imported from the JS file, which initialize the WASM module. It can be either a predefined memory structure, a table of data, functions or just global variables.

- Once the imports are defined, it is the time to define functions. Each begins with the `func` keyword - `(func <name> (type ...) ...)`. In the next line local variables are defined - `(local ...)` (just as global variables mentioned earlier are defined with the keyword `(global ...)`). To later use them in the function body `local.get` could be used and to assign `local.set`.
- Function body is full of various operations, just as in the regular x86 Assembly. There are e.g.`i32.const` keywords introducing constants or e.g. `i32.xor` performing xor operations or `call ...` calling other functions. More could be read in the Text Format documentation [63].
- At the end of the file global variables are initialized (the ones earlier were imported from JS). Usually they use `(mut ...)` keyword, to be of a mutable type.
- Following them, exports sections are laid out. Each begins with `(export ...)` specific keyword and usually describes function, which will be exported to JS and will be available to be used from there. (e.g. cryptonight hashing function in mining payloads).
- Last few lines of the WASM module regard `(data ...)` and `(elem ...)` sections. With the first one we can just write the string contents into global memory. Data sections allow a string of bytes to be written at a given offset at instantiation time and are similar to the `.data` sections in native executable formats. The latter on the other hand allows to initialize regions of tables (instead of linear memory as `data`). The tables that are defined with `(table <size> ...)` keywords earlier of course.

1.5.3 Emscripten

The compiler's toolkit, which is widely adopted as the main WebAssembly technology is called `Emscripten`. In many ways it reminds of `gcc`, as for instance it introduces compilation flags such as `-O1, -O2 or -O3`, which optimizes the code while compiling. What is worth mentioning, is that just as it is in `C/C++` programs, `Emscripten` can unroll loops. What it means, is that whenever the compiler finds out, the program would be executed faster and the overhead in code size is acceptable, it can remove the `loop` keyword and the conditional jump to the beginning of the loop, at the end of each iteration, and replicate loop body as many times as it has to. Some other interesting point is that the compiler can produce both standalone WASM modules, which can operate independently, but can also generate a `.wasm` file and a JS glue code. The glue code is automatically produced and works as a "middleware" between the user's application and WebAssembly code. `Emscripten` is also capable of optimising the output, if the compiled code is meant to be run in the worker thread and tons of other options that are for sure interesting, but are not be mentioned here (see [28]).

1.6 Cryptojacking

1.6.1 Mining vs illicit mining

In-browser mining is not illegal. Obtaining digital currencies by calculating proof-of-work puzzles is fine, doing so on other people's computers is acceptable as well. The turning point is in one detail - user consent. If a website doesn't want to bother its user with ads, it can throttle the miner to let's say 10%, which shouldn't be even noticed by the user and earn money thorough computed hashes, it later forwards to the pool and gets rewarded. The only condition such a page has to fulfil, is to inform user about mining taking place. It could be either a pop-up window or any other visible form of communication. This is what differs legal cryptomining from cryptojacking.

1.6.2 Architecture of mining - threat model

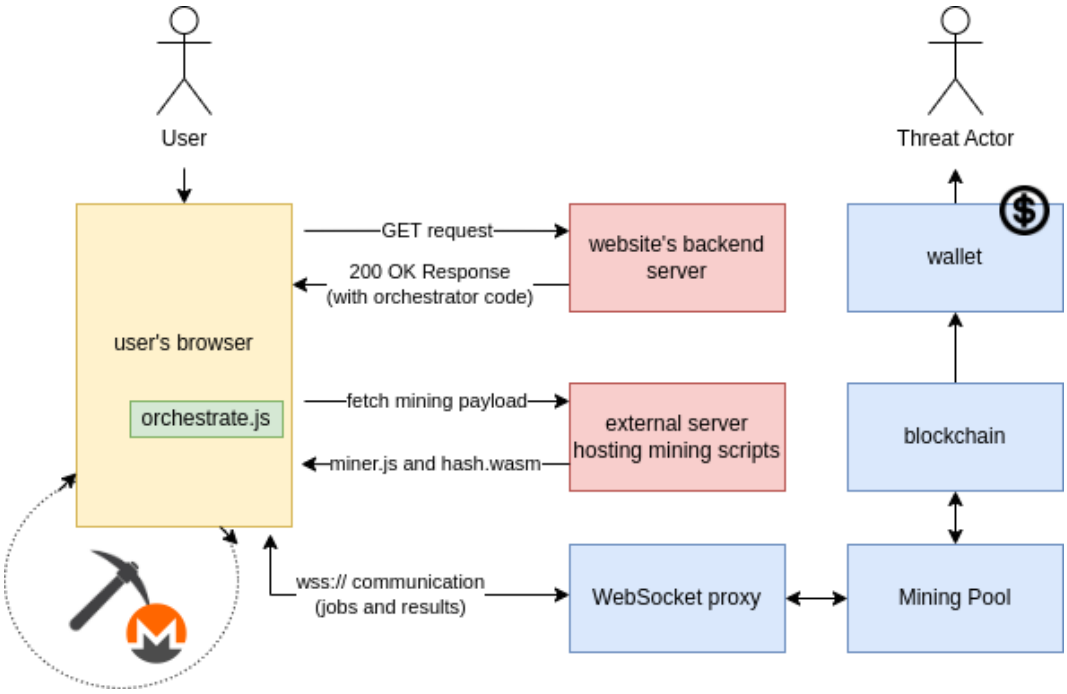


Figure 1.2: Cryptojacking flow overview

Before the main part of this work will be covered, it is essential to understand how the typical cryptojacking attack scenario looks like.

1. At the beginning user visits a malicious website. The browser, as a client sends an HTTP request to the server and gets the contents, which usually involves

HTML, CSS, JS files and static content like images - these might be fetched from CDNs.

2. Among these files or inside one of them there might be hidden a miner or instead a very short orchestrator code. The latter option (also presented in the Figure 1.2) is more popular, as it is easier to hide a short snippet and the rightful owner of the website (if it got hacked) might not even be aware, their web application is serving malicious contents.

3. If the miner's code was not sent in the previous step, the orchestrator pulls it from the external server controlled by the actor. It usually means both JS script and WASM module, where JS is a wrapper initializing the WebAssembly functions to perform hashing operations.

4. For as long as the user is active on the website, their hardware is exploited mining Monero by calculating proof-of-work hashes. The victim's browser knows what to evaluate, because it keeps an open, duplex connection with proxy server, via web sockets, and receives the jobs (these hashes to calculate), which it resolves and sends back.

5. Proxy server handles later the communication with mining pool. It all happens via Stratum protocol, which is a JSON-RPC type of channel. It means that the messages sent to and from the mining pool, are structured as JSON objects and the methods and keywords are well-defined by the Stratum itself. It makes the communication with mining pools universal, meaning cryptocurrency-agnostic.

6. In the end the mining pool rewards the mining entity, for the hard work it has done to extend the blockchain by another block. The pool knows what is the miner's share and adequate prize (minus the commission fee the pool imposes). However starting the mining process it was the malicious actor who provided their wallet address in the mining payload, so the reward will be sent to them, even though it was the victim's hardware computing.

1.6.3 Sources of payloads

There is a question one could ask, how does it happen, that websites are affected by mining scripts. The most straightforward answer is that they are deployed as such or get hacked, usually by some unpatched vulnerabilities and modified to serve malicious content, although originally they were legit and benign websites. This is for sure correct, but there are some other possibilities, less obvious ones. Here are a few that I believe are worth bearing in mind:

- **third party libraries** - in the code of the libraries used commonly there might be a hidden payload

- **ads** - malicious adversaries sometimes are skilled and capable enough, not to affect a single website, but instead hide their payloads in the contents of the advertisements. If such a trick works out undetected, even the biggest ad platforms and markets, such as DoubleClick may unconsciously become a distributor of malicious content and serve it to many more customers than the website from typical scenario could cause.
- **public Wi-Fi hotspots** - another source of mining scripts might be a compromised or set up as a trap access point, which would add hazardous contents to any website user visits, it is even more dangerous than the previous scenarios, because it affects all user actions.
- **hacked routers** - the same story could be said about hacked routers. They can become a man in the middle between the user's computer and external world and append unwanted, abusive code anytime.

1.7 Environment

Another important factor in analysis of hazardous websites and their mining payloads is the safety of the researcher. In the field of malware analysis there is a lot of well-known techniques and general knowledge, not to harm ourselves when interacting with dangerous content. Cryptojacking definitely falls into this category, so it would be advised to run a virtual machine, preferably well sandboxed and etc. Bearing the risk in mind, I have decided to wipe out the disk, deleting the operating system and all the data irretrievably and proceeded with fresh Ubuntu 22.04 LTS installation, performing all the experiments on bare-metal. The main reasons were:

- **comfort** - I could afford to let anything happen to the machine, including firmware backdoors, as the machine will not be used anymore after the experiments
- **ease of set-up** - needing to work with the tools such as e.g. `CDP` and `perf`, which are highly dependent on the kernel and broadly speaking on the low-level interaction with the OS, the results may be more precise, when the additional layer of virtualization is not in place.

1.8 Previous attempts

Finally, a word should be said about the previous attempts to tackle cryptojacking. All of the methods known beforehand are based on the black-listing approach. There could be different forms the final tool adopt e.g. browser extensions or standalone applications, but all the detection attempts were as good as a list of forbidden phrases, patterns or expressions. The ineffectiveness of this approach in the long

run is something that was already proved before [1] and in this work I want to focus solely on the modern and creative solutions to the underlying issue of cryptojacking and check how efficient and promising they really are.

Chapter 2

Minesweeper

2.1 Overview

Having the generic concepts grounds covered now it's finally the time to focus on what is the essential topic of this work. The first cryptojacking detector that I have been scrutinizing is called Minesweeper [3]. The solution was originally presented during the "ACM SIGSAC Conference on Computer & Communications Security" in October 2018. The date is important here, as it has been already more than 5 years since then and the time that passed will play a significant role in how the results presented by the original authors compare to my own research.

Having said that, the authors of this tool proposed a never-seen-before approaches to detect illicit mining. The first focuses on the hardware resources exploitation, which undoubtedly go hand in hand with cryptomining per se. The idea is to monitor, using well-known tools, the computer and spot the number of cache operations, which in case of cryptomining should be incomparably, to any other resource heavy in-browser activity, higher. The other innovative feature that Minesweeper in equipped with is the cryptographic operations count detector, which allows to decompile the WebAssembly payloads and by static analysis distinguish the ones performing cryptomining under the hood, from these which are benign.

2.2 Setting up the environment

2.2.1 Basic tools

Minesweeper's source code is publicly available due to the courtesy of the authors, who did not mind sharing their work with the community. However, as I mentioned before, it is been over 5 years since the tool was developed, hence some of the technologies changed and it is not that straightforward to run it. For instance the use of `python2`, which is the main programming language used in the project. Most of the

operating systems released nowadays don't have it preinstalled anymore, even the standard apt repository on Debian based systems doesn't offer it. To get the closest setup to what the authors had though, I installed: `python2, pip2, nodejs, nvm, wabt and perf`. Python2 is necessary to run the Minesweeper, `pip2`, as a package manager for Python is needed to satisfy all the dependencies within the language. The authors provide the source code with Bash script called `python_requirements.sh`, which helps automatize the process. Node.js was essential for one of the tool's profiling features, that cannot be run in any other language than Javascript. To make the node environment as close to the original as possible, I used `nvm`, which allows the user to get any version of node quickly and effortlessly. My choice was: `nvm install v10.19.0`, which is dated to be from February 2020. Node requirements are satisifed with in a similar manner to the ones in Python. `npm_requirements_install.sh` script uses the in-built node environment's package manager called `npm` to do so. On the other hand the WebAssembly Binary Toolkit, known simply as `wabt`, I downloaded from the official WebAssembly's GitHub repository, as a .tar.gz compressed archive and extracted it. It is one of the core dependencies from the Minesweeper's runtime point of view. Finally the `perf` utility, to monitor privileged kernel operations, had to be installed. This software is highly dependent on OS kernel version, as it has to be compatible with what the user's computer is currently running. Hence there was really no simple way to downgrade it to what was originally used by Minesweeper's researchers. To get it, there need to be three Ubuntu apt packages satisfied. This command: `sudo apt-get install linux-tools-common linux-tools-generic linux-tools-` `` `uname -r` `` pulls them from the repository, bearing in mind that the last package name depends on the resolution of the `uname -r` command, which prints the kernel release. Once `perf` is installed, it should be enabled for unprivileged user by opening `/proc/sys/kernel/perf_event_paranoid` processor directory file and putting there the value of `-1`.

2.2.2 Custom browser build

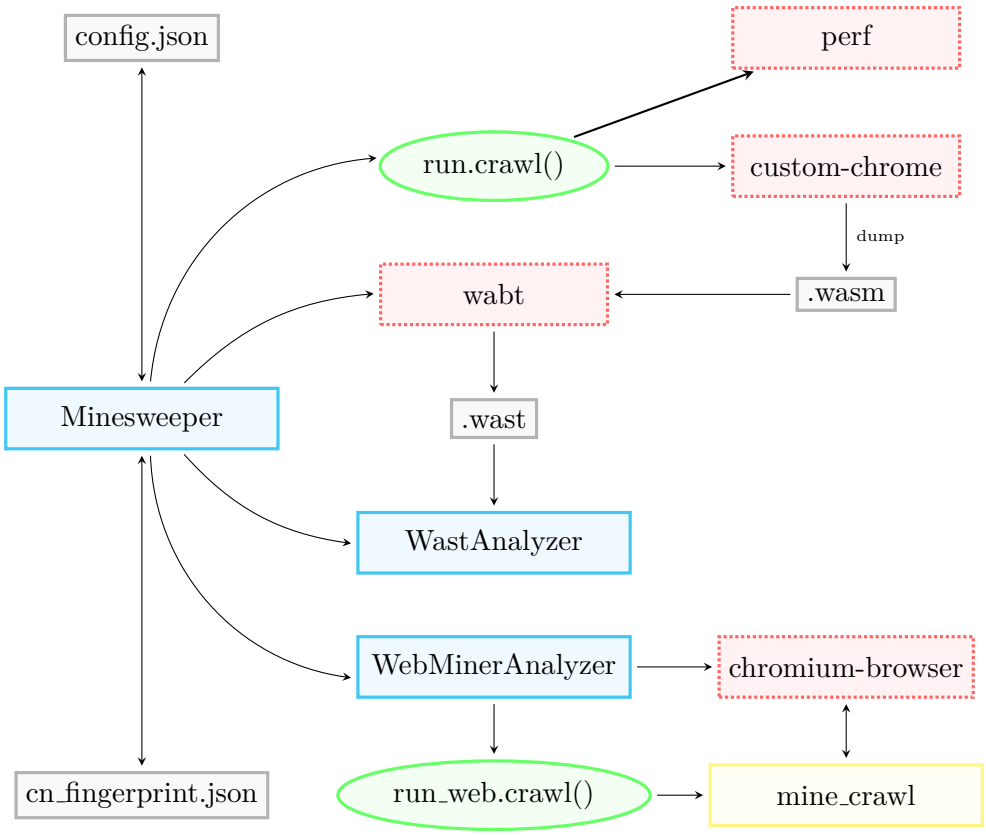
In the previous section the dependencies needed to run Minesweeper were covered. The way to install them is in all cases easy enough, to just mention them, without and further instructions. Nevertheless there is one more preliminary step, which needs to be fulfilled to be able to use Minesweeper. In one of the phases of the program's runtime, the subprocess is called to execute Chrome browser and dump the WebAssembly file. Publicly available Google Chrome is not sufficient to perform this action. A user needs to get a debug build of the browser to be able to download the mentioned `.wasm`. The authors compiled Chrome on their own, from sources, and provide users willing to use Minesweeper with the ready-to-use build. However after all these years, the operating systems and their programs are not compatible anymore with the 2018 version of the software. I have tried many different approaches and did a lot of debugging, unsuccessfully trying to fix the underlying issue of incompatibility. In the end I had to build Chrome on my own.

The instructions on how to compile the browser in debug mode are mentioned in the Chromium repository [17]. It takes a lot of time (over 10 hours) and computer resources, even on the powerful servers equipped with over 100GBs of RAM, that I used. However once it's done, custom, debug build can be run and all the efforts lead to the ability of using the necessary command line switch [80], more known as runtime command line flag called `--js-flags="--dump-wasm-module --dump-wasm-module -path=/path/to/file"`. Bear in mind, that this switch is not documented anywhere and even in the Chromium source code repository it is tough to find any traces of its existence. By this I mean the flag `--dump-wasm-module`, which is a subflag run by the V8 JavaScript engine inside `--js-flags` switch - this one being documented properly.

2.3 Source code and features analysis

In this section the overall architecture of the Minesweeper and its features is covered. Before the potential evasion techniques will be presented, a deeper understanding of how the program really works is needed.

2.3.1 Code structure



Structure of the code. Python code is blue, JS is yellow, runners are green,

external programs are red and files are gray.

2.3.2 Runtime and program flow

To run Minesweeper one can simply execute `python2 minesweeper.py -t <URL>`. The high level abstraction of what the program does is, that the user provides it with the link to the website and it checks whether the web application that the end-user would be interacting with is trying to abuse their computer resources by drive-by mining. If one takes a closer look into the source code, the overall flow of the program looks like this:

1. From the command line interface the `minesweeper.py` with the given URL is executed.

2. The program parses the arguments provided by the user and in case any of the flags or their values are wrong, it exits with the proper error message.

3. Once the checks are passed successfully, the program proceeds to the first stage of its runtime, called '**Stage 1: Website Analysis**'.

4. In this stage the Python2 library - 'subprocess' - is used. It executes the system command, which is pulled from the configuration file. There is a separate `config.json` that is used globally by Minesweeper, it helps organizing the code and when needed delivers the environmental variables' values or paths the to system commands. In this case it provides the main function with the path to the debug Chrome build, which was described in the 2.2.2 section. The browser opens a new window (or if the flag –headless is passed in the code, it does so unnoticeably) and opens a tab in which the given, at the beginning, URL is loaded. If it spots in the upcoming network traffic, that one of the GET requests from the server pulled the `.wasm` file, it downloads it to the specified in the configuration file directory with the name of the MD5 hash hexdigest of the target URL. The WebAssembly file's name is not the original one either and it is on the Chrome's side to work it out. Of course here an assumption that such a file is indeed loaded is made, if no WebAssemly was found, the program skips Stages 2 and 3 and moves to the Stage 4 immediately.

5. In the meantime, when the custom browser is running and the website is loaded and kept opened, another sub-process is forked. It executes a `perf` utility. It is a kernel-based subsystem for Linux that provides a framework for performance analysis. Minesweeper uses one of the tools called `perf stat` and to be exact `perf stat -a -e LLC-loads,LLC-stores,L1-dcache-loads,L1-dcache-stores sleep 5`. It monitors the whole system (because of `-a` flag) for a given amount of time, in this case for 5 seconds and counts the OS-wide number of loads and stores to the L1 and L3 level caches. Which means number of reading

and writing operations to the fast access memory. According the the research done by the authors of Minesweeper [3], in the situation in which Monero mining is involved, these numbers should exceed any other computational heavy, in-browser activity e.g. JS-gaming, WASM-gaming or Video playing.

6. When the `perf stat` process ends with success the program moves to the second phase called '**Stage 2: Wasm Analysis**'.

7. Now when the `.wasm` file is downloaded, another external tool is used. Again, from the configuration file, the exact location of the `wabt` is pulled and it decompiles the WebAssembly Binary Format into WebAssembly Text Format. Having this human readable file Minesweeper proceeds with its analysis. There is a separate Python file, representing an independent tool within the Minesweeper project called `wast_analyse.py`. It performs all the parsing and finding out what most probably is the purpose of the `.wasm` that the website loaded. In this case its job is to try to answer the fundamental question 'Does this .wasm have something to do with cryptomining?'.

8. At first `wast_analyse.py` parses the WebAssembly Text Format file, to find all the functions. The function definition always starts with a keyword '`(func`', so the regular expression is used to discover all of the starting points. Each function starts exactly after the previous one ends, so it is not tough to find, where the body of each of them is either. They are saved in the dictionary, which structure looks approximately like this:

```
#  Functions structure
functions:
{
 fun1: {'code': <whole_function_body_code>, 'i32.mul': 0,
 'i32.xor' : 0, .... <all_the_operations>},

 fun2: {'code': <whole_function_body_code>, 'i32.mul': 0,
 'i32.xor' : 0, .... <all_the_operations>},

 fun3: {'code': <whole_function_body_code>, 'i32.mul': 0,
 'i32.xor' : 0, .... <all_the_operations>}
 ...
}
```

Later the program computes the cgraph, which is a call tree of of all functions and their 'calling list'. This means: "what functions does each function calls and how many times". The structure of such a dictionary looks approximately like this:

```
#  Cgraph structure
cgraph:
{
 fun1: {'calllist': ['call funX', 'call funY'], 'call funX': 1,
 'call funY': 3},
```

```
 fun2: {'calllist': ['call funZ'], 'call funZ': 2,},

 fun3: {'calllist': ['call funJ', 'call funK', 'call funL'],
 'call funJ': 1, 'call funK': 4, 'call funL' : 1}
 ...
}
```

Once the `wast_analyse.py` knows what are the functions and which of them call the others inside their body, it moves to counting operations. This step means finding out what each of the functions does, and to be precise what are the WASM instructions used by it. The `functions` structure is already prepared, so the program only increments the counter, in the dictionary, of each instruction it finds. What happens next, is the process of loop counting. As mentioned in the chapter 1.5.3, there are two types of loops - normal and unrolled.

8.1 `Normal loops` - are easier to handle, because there is a keyword '`loop`' in the WASM, pointing out to the beginning of the loop and there is another one called '`end`', marking the end of it. Loops are counted based on the occurrences of these keywords. However, there are some issues, because: 1. loops can be nested (the ending of the first one is later than the beginning of the second one), 2. 'if statements' use the same keyword to indicate their end. Wast Analyzer resolves these by keeping track of 'how many of them started vs how many already ended'. This means that the loop ends only if all the earlier 'ifs' where matched with their respective endings (their counter should be zero) and the outer loop ends only when all the nested ones have been matched. The counting function executes itself recursively, whenever nesting occurs.

8.2 `Unrolled loops` - since the compiler can unfold some of the loops, there was a need to check whether any type of repetitive behaviour appears within the functions. Thus Web Analyzer uses its own Sequence Manager to identify snippets of code that evince repeatability. It does so by extracting the first (most left) operation of the line in the function and checking if there are code snippets not longer than `max_len` length, repeatedly occurring one after the other more than `min_seq` times. If so, the loop was unrolled. Sequence Manager is configured to flag such a code as a loop, if the maximum length of each snippet is not longer than 25 operations and minimum number of times it reoccurs is 4 in a row. It checks every single line as a potential starting point of the sequence.

At the end the analyzer groups its findings into predefined categories, based on how well the WASM arithmetic operations found inside functions and loops match the cryptographic behaviour. These are used to return a verdict in the next Minesweeper's stage.

9. When the WebAssembly is fully profiled and Minesweeper already knows what

is inside it, it is time to analyse the output and decide whether any mining activity has been recognized. It means the third stage is about to happen - '**Stage 3: Crypto primitives detection**'.

10. This stage is not intricate. Based on the output files of `wast_analyse.py` the program iterates through the results, in a form of Python dictionaries, and gives its verdict, distinguishing two forms of suspected activity:

    10.1 the first one being generic cryptography, which means that even though the `.wasm` does not evince Cryptonight algorithm, it most likely performs some undefined kind of cryptographic evaluations. To be qualified as 'generic crypto', the `.wasm` module has to have more than 5 functions in which there are some cryptographic operations happening in loops and at least 11 loops (of both kinds) overall in the whole code that contain these operations as well. The WebAssembly operation to be considered cryptographic has to be one of these instructions:

    ```
    [i32.xor, i32.shl, i32.shr_u, i32.shr_s, i32.rotl, i32.rotr,
    i64.xor, i64.shl,i64.shr_s, i64.shr_u,i64.rotl, i64.rotr]
    ```

    which means either a XOR, shift left, shift right, rotate left or rotate right. Whenever the condition is met the Minesweeper will inform the user in its final output, that it recognized suspicious generic cryptography.

    10.2 the other is called Cryptonight algorithm primitives identification.The authors manually analyzed hundreds of websites' WASM payloads and calculated how many crytographic operations should be performed for each of the Cryptonight's primitives, and as it was described in the section 1.3.3, this algorithm consists of five fundamental ones. Minesweeper is hence equipped with `cn_fingerprint.json` file, which defines for each of the primitives, the number of operations that are expected, in order to claim that the Cryptonight has been detected. Of course as different variants of the mining payloads may slightly vary, Minesweeper is designed to establish how many of the primitives has been recognized and what's is the difference score in the operations count between the analyzed file and the fingerprint. The threshold is, that at least 70% of the WebAssembly instructions types have to be the same as in the fingerprint and the variance in the count of them has to be less than the number of instruction types. This means, that the initial manual work allowed to define patterns, which recognized in the WASM code, detect the presence of Cryptonight, based on its design.

11. These first three stages cover what is in this work considered to be never-seen-before. The last phase of the program is there to profile the input website once more and try to evaluate its malice, using some well-known techniques. Having said that, in the end the WebMinerAnalyzer as a fourth stage is executed - '**Stage 4: Website profile**'.

12. Even though the disruptive part of Minesweeper took place in the previous phases, the authors designed the last stage to add something novel to what has been already known in the world of cryptojacking detectors. The WebMinerAnalyzer tool is divided into sections, initially it repeats what the first stage of the Minesweeper did with the custome-chrome, this time though, normal build is enough to crawl and dump all the elements the website uses. Starting with DOM `.html`, through JS scripts, Web Socket activities, CPU usage, Web Workers, received/sent requests, ending on e.g. cookies. To achieve it however, there is another component needed. It is the JavaScript code utilizing the power of Chrome DevTools Protocol (CDP), described in the section 1.4.1. The way it works is that the WebMinerAnalyzer executes `chromium-browser --headless --no-sandbox --remote-debugging-port=9222 &`. The last flag allows TCP connections on local port to interact with in-built browser's CDP. It is `mine_crawl.js` job to do it and make use of predefined protocol's domains to gather all the data. What the Node runtime collected is later analyzed back in WebMinerAnalyzer. The program detects:

    - Web Workers' code by looking into JS code and searching for any occurrences of the words listed in the blacklist patterns
    - WebAssembly, based on blacklisted hexadecimal WASM header - `b'\x00\x61\x73\x6d'`
    - Orchestrator's code, based on the blacklisted JS keywords for each of the mining services. The services discovered by the authors are:

      ```
      coinhive, cryptonoter, nfwebminer, jsecoin, webmine,
      cryptoloot, cryptominer, coinimp, deepminer, monerise,
      coincube, grindcash, coinhave, kuku, cpufun, minr,
      ricewithchicken, connection, mineralt, dryptonight,
      blakcrypto
      ```

    - Number of Web Workers running in the background
    - CPU usage
    - Web Sockets communication and the occurrence of Stratum protocol in it or Web Sockets keys
    - Addresses of pool proxies or public mining pools and attempts to login in to them - wallet identifier presence

    The last phase is detection. WebmMinerAnalyzer decides whether the data it collected and discovery process based on blacklisting patterns it made, is considered cryptomining or not. There are several conditions, which if met produce a positive output, meaning mining has been recognized. These are:

    - (Web Workers' mining payload `OR` keys `OR` any of the mining services types presence) `AND` open Web Socket communication `AND` positive number of Web Workers

- Stratum communication presence
- Public mining pool usage
- Detected login attempt to pool

13. At the very end Minesweeper prints the results of all of the three methods it used to determine mining.

2.4 Potential evasion techniques

The previous section covered what the Minesweeper does and how. Once we are familiar with it, now it is the time to try to trick it into believing that the malicious payload it analyses is not responsible for mining.

2.4.1 Generic cryptography - unrolled loops

In the second stage of its runtime Minesweeper counts loops. It detects 'generic cryptography' behaviour, if there is a sufficient number of functions and loops (of both types), containing particular WASM instructions.

```
1 if stats['gen_crypto']['f_count'] > 5 and \
2    stats['gen_crypto']['loop_count'] \
3    + stats['gen_crypto']['loop_unr_count'] > 10:
4     gen_crypto = True
```

Listing 2.1: Generic crypto detection verdict

There is probably no way one could hide the occurrence of functions, as the semantics requires the keyword usage. Reducing the number of them to be lesser than 6 wouldn't probably work either, as it would require modifying how the Monero mining works or at least re-writing the existing source code of Cryptonight algorithm library. As this idea sounds barely possible from the very beginning, what about the other condition, loops. As one can see, the requirement is that the overall number of them is more than 10, but the proportion between normal ones and unrolled is not important. In the section 1.5.3 it was mentioned that the Emscripten compiler is equipped with the optimization flags. The highest level of optimization one can have compiling `C/C++` to WASM is `-O3`, meaning the building process should take longer, but the produced output code will be well-optimised. As we know, loops, as they are in the normal from, may sometimes be expensive for the processor, because they require backwards jumps. There is a great chance that the optimised code will have some of them unrolled, because it gives speed for the price of the code size (`-O3` flag doesn't care much about the size, there are other ones that do). The question is how many could be unrolled.

I could only experiment with different publicly available Cryptonight libraries and hope than one of them compiled will produce a payload similar to the samples.

All I hoped for was to be able to compare how produced WASM files behave having different levels of optimisation and try to find out what could be the flag used by cryptojacking actors. The comparison is based on `loop` keywords, since the analysis focuses on the loops.

Cryptonight compiled with different optimisation	Normal loops count	Well-optimised mining payloads	Other payloads
`emcc -O0`	127	Number of samples: 25	Number of samples: 9
`emcc -O1`	100	Avg number of loops: 63.08	Avg number of loops: 159.11
`emcc -O2`	59	Median: 63	Median: 164
`emcc -O3`	49	Std dev: 9.09	Std dev: 33.26

Table 2.1: Comparison between Cryptonight implementations compiled by myself, with different optimisation flags and payloads collected by the Minesweeper authors (available in minesweeper/wasm_samples/miner_samples/). It helps in figuring out what could have been the flag originally used by malicious actors, when they crafted each of these payloads back in the day. I divided them into two groups, as number of loops significantly differed between various modules and and clear distinction into better and worse optimised payloads could have been made. Notice: The table presents all 'normal loops' (the ones with loop keyword), not just the ones in which cryptographic operations happen.

Results from the Table 2.1 are based on the compilations of the library that gave the most similar outputs to the malicious payloads, which were taken from the `wasm_samples/miner_samples/` directory provided in the Minesweeper. From the looks of it, most of the mining payloads found in the wild in 2018 by the authors of Minesweeper were compiled using either `-O2` or `-O3` flag. This means, that there is not much room for improvement in terms of enforcing greater optimisation, however as we know from the runtime analysis in the section 2.3.2, not all loops are counted. Minesweeper only counts these, that are containing 'forbidden' cryptographic instructions. In all of the cases, which are presented in the Table 2.1, less than 50% of normal loops is counted by Minesweeper, as some loops are not 'suspected' to perform hashing due to lack of certain instructions. The number of valid ones fluctuates around 25 for different payloads that were tested.

If a malicious actor is determined enough to make thousands of dollars and stay undetected, they could unroll around 15 of these normal loops, picking the shortest ones and the count would drop below 11 - number that is needed to detect 'generic cryptography'. Disclaimer is that in this work I didn't try to unroll them myself.

It may look that the the problem of loop counting solved, but we can't forget about the loops that the compiler already unrolled by itself. If they contain any cryptographic operations, they are counted by the separate counter as well. The idea that comes to mind is, what if it was possible to reduce their number to zero. To do so, the Sequence Manager would have to go through the code and return that none repetitive snippets were found. If only there was an option to add random instructions in-between the original code, breaking the loop pattern, it could work. Here the `nop` instruction in WebAssembly comes with help. This instruction, as it is in the normal Assembly - does nothing. That's good, because it wouldn't break the program and more importantly it is cheap, so it won't waste processor's cycles. Having this powerful tool that could trick Sequence Manager, it has to be figured out, how to add these nops between valid code. If every X lines Y nops would be added, there is a great chance it would mean nothing, because they would be treated as additional part of the sequences. The solution is to randomly insert some number of nops every 6 lines and therefore disrupt the repetitiveness. I wrote a script that does it, abiding all of the WASM rules, which are:

- the beginning of the file, before the first function has to be unmodified
- the end of the file, after the last function's definition has to be kept as it was
- there can be no instructions between the end of one function and the beginning of the next one
- it is not allowed to add instructions between the function's first line and the second one, because they are both considered as a definition, only later the body of the function begins.

```
#!/bin/bash

set -e

# Before running this script decompile *.wasm to good.wast using wabt's wasm2wat

beg="temp.wast"
ending="aux.wast"
i=1

# Take the beginning of the file
while read p;
do
  i=$(($i+1))
  echo $p >> $beg
  if [[ $(echo $p | awk '{print $1}') == '(func' ]] ; then break; fi
done < good.wast

# Take the end of the file
tac good.wast > this.wast
```

```
j=1
while read p;
do
  j=$(($j+1))
  echo $p >> $ending
  if [[ $(echo $p | rev | cut -c1) != ')' ]] ; then break; fi
done < this.wast

# Modify the body of the wasm between the beginning and the end

arr[0]="nop"
arr[1]="nop\nnop"
arr[2]="nop\nnop\nnop"

lines=0
tail -n +$i good.wast | head -n -$j > thiss.wast

# This is a lot faster, but adds nops where it's forbidden - between
    functions
# while read p;
# do
#   rand=$[ $RANDOM % 3]
#   printf '%s\n' "${p}" >> $beg
#   !((lines++ % 3)) && echo -e $"${arr[$rand]}" >> $beg
# done < thiss.wast

# This one is fairly slow, however there is no need to do any manual
    job to fix the code and get it compiled
while mapfile -t -n 2 p && ((${#p[@]}));
do
  rand=$[ $RANDOM % 3]
  printf '%s\n' "${p[0]}" >> $beg
  lines=$((lines+1))
  if [[ $(echo ${p[0]} | awk '{print $1}') == '(func' ]] || \
  [[ $(echo ${p[1]} | awk '{print $1}') == '(func' ]];
  then
    lines=$((lines+1));
    printf '%s\n' "${p[1]}" >> $beg;
    continue;
  fi
  !((lines++ % 3)) && echo -e $"${arr[$rand]}" >> $beg
  printf '%s\n' "${p[1]}" >> $beg;
done < thiss.wast

rm thiss.wast
tac $ending >> $beg
rm $ending
rm this.wast

# Compile temp.wast back to *.wasm using wabt's wat2wasm
```

Listing 2.2: Script adding nops to the WASM

This script resolves the problem of any manual modifications and prepare ready-to-use file, which as it turns out indeed tricks Minesweeper to believe that there are 0 unrolled loops in the modified file it analyses. This simplifies the problem down to normal loops only.

2.4.2 Cryptonight primitives - abundant operations

The Cryptonight primitives detection technique could be potentially evaded by repeating the trick with nops, however this time it would have to be other instructions - the ones that are considered cryptographic. Having these operations would of course produce much more CPU overhead than the nops do and, as the authors rightly pointed out in their paper, make it unprofitable to run mining campaigns. Another argument would be, that what if it is not Monero and Cryptonight that some mining services would use. In this case, again, the authors are aware of the issue and because Minesweeper is not a production-ready commercial software, it would be rather unfair to blame the tool for not handling these situations. Extending a dictionary of different proof-of-work algorithms for new fingerprints shouldn't be problematic. As of 2024 for instance, Monero is not using Cryptonight anymore, it switched to RandomX.

There is however one potential scenario to deceive Minesweepr's Cryptonight detection. It would require Cryptonight modifications though, but someone who is running a broad campaign and has an access to the C source code files, before they end up being compiled to WASM could do it. The way the tool is designed, is to spot the primitives, including the hashing functions used in the last phase of Cryptonight's proof-of-work. In this phase, one of the three SHA-3 finalists is chosen randomly. These are Blake, Skein and Groestl. Minesweeper knows what are the fingerprints for each of them and can detect all of them with ease. The problem is, that it is not exactly three algorithms only. There is a fourth one and it was there back in 2018 as well (at least in Monero's case).

```
1 static void (*const extra_hashes[4])(const void *, size_t, char *) =
2 {
3     hash_extra_blake, hash_extra_groestl, hash_extra_jh,
    hash_extra_skein
4 };
```

Listing 2.3: Functions to be choose from to calcuate the last phase of Cryptonight

Minesweeper doesn't know about JH hashing algorithm. If the malicious mining code would be modified to calculate only JH at the end, the detection would be unsuccessful. Furthermore to avoid being detected the last step of proof-of-work calculations could be moved to the proxy servers. Victim's computer would calculate the main part of Cryptonight, the most resource-heavy one with a lot of AES iterations and and rounds and the extra hash at the end would be performed somewhere else. It would remove 3 (in reality 4) out of 5 (in reality 6) primitives and

make recognizing Cryptonight harder. But that's just speculations.

2.5 Possible improvements

Bearing in mind Minesweeper is not a commercially distributed software with continuous development, but just a Proof-of-Concept tool, there are issues that one shouldn't be critical about. Although in this work, I have tried to come up with constructive criticism examples of what could be considered, if Minesweeper would be taken into production environment. Of course the room for development is almost unlimited, that is why the improvements in this section apply only to the features that are already in the code.

2.5.1 Loops count and ratio

Based on the typical results that the `C` to WASM compiler produces, the program could compare the average ratio between normal and unrolled loops in mining payloads. Speaking of Cryptonight alone, the same way its primitives fingerprint was prepared, one could draft a fingerprint of a typical number of loops the algorithm has and to expand this idea even more, monitor the relation between them. If for instance the number of unrolled loops is zero, this might mean someone tried to tamper with the WASM file to prevent detection. Further steps could be developed later to persist against different evasion techniques.

2.5.2 Counting Web Workers

In the last, fourth stage Minesweeper crawls and collects different types of data regarding the analyzed website. Spotting Stratum protocol in the Web Sockets communication is clever, other blacklisting methods such as spotting the crucial parts of orchestrator's code isn't bad either, these are all similar to what was already known and seem to be susceptible to advanced obfuscation though. The part that stands out right away however and seem to be problematic to evade, if a malicious actor is oriented to maximize their profits, is the count of Web Workers. There is no way to use them, even with the greatest obfuscation, without Chrome DevTools Protocol spotting it. As described in the section 1.4.1 browser's CDP operates on different domains and Target domain is created for every browser's element. Minesweeper is indeed leveraging this fact and properly counts the number of Web Workers. The program acknowledges only the Targets to which the debugger is attached to, as of 2024 it is still enough to do so, because other Targets are created too early in the runtime to be attached to (even with the `Target.setAutoAttach({autoAttach:true,...})`), I have seen different discussions though and it might be changed one day. It would mean that Minesweeper would not only spot Web Workers targets but all other as

well. The authors wanted to prevent that, so whenever the 'page' Target is attached to, it is not counted. However, this is not enough.

```
export enum TargetType {
  PAGE = 'page',
  BACKGROUND_PAGE = 'background_page',
  SERVICE_WORKER = 'service_worker',
  SHARED_WORKER = 'shared_worker',
  BROWSER = 'browser',
  WEBVIEW = 'webview',
  OTHER = 'other',
  /**
   * @internal
   */
  TAB = 'tab',
}
```

Listing 2.4: CDP's Target domains types

In the Listing 2.4 there is a source code snippet from Chromium project and TypeScript's enum in it presents what are the different Targets. Apart from some internal ones like 'dev-tools' Target for instance, the most important ones are listed. The one called `background_page` is interesting. This is a HTML page, that is created for every extension installed in user's browser. The extensions can, but don't have to use it, but regardless one is always provided. If one day Chromium will resolve this race condition of auto-attaching, the number of attached targets may be misleading, depending on how many extensions the user uses and Minesweeper may produce false positives. The improvement here could be to detect not only the presence of 'page' Target, but all the others as well and explicitly count just Web Workers.

2.5.3 Perf stat - test of time

With perf feature there are a couple of issues:

- First of all, Minesweeper is not equipped with any automatic detection of mining based on perf output, all it does is execution of the `perf` and later an actual manual work would have to be performed. This makes this feature incomplete and lack of perf's output analysis in Minesweeper and no sign of its influence of the final results is what could be improved the most in the whole project
- Other than that the perf output parsing in Minesweeper isn't working on a new generation computers, where there are two different processors - cpu_core and cpu_atom. But this isn't that important considering Minesweeper's non-commerciality.
- Moreover, the `perf` ran in the Minesweeper is system wide and its results can be influenced by other processes. Narrowing the monitoring scope just to the

browser would be more accurate.

- Finally, extending the feature to work with other operating systems than just Linux would be another huge improvement.

Having said that, I tried to do manual performance tests, similar to what the authors did. The purpose was to check if abnormal number of L1 and L3 cache operations is unique within one machine or is the number so disproportionately high, it would stand out among other computers' results as well. What it means is, I wanted to check if the cryptomining's store/loads being extreme on the old machine, would be also distinguishable on the newer computer or they would be lost among other resource consuming in-browser operations, such as video playing or gaming, because new machine would have its mining performance peak on completely another level. This was important to check, because Minesweeper wouldn't know how good the computer of the user is and without explicit comparison to other activities it could produce false positives e.g. X million of L1 loads would be considered a threshold for cryptomining activity and on an old computer it would indeed be it, but some powerful, new generation machine would operate on values like X on normal basis and only 10X would be the CPU exhausting mining threshold.

	i7 1st gen - 2010		i7 13th gen - 2023	
Operation	WASM Gaming	Mining	WASM Gaming	Demanding WASM gaming
L1 Load	$\sim 5,200$	$\sim 78,000$	$\sim 4,900$	$\sim 51,200$
L1 Store	$\sim 3,000$	$\sim 19,500$	$\sim 2,900$	$\sim 27,500$
L3 Load	~ 140	~ 750	~ 68	~ 61
L3 Store	~ 180	~ 530	~ 27	~ 14.8

Table 2.2: Perf stat results - number (in millions) of cache operations within 10s. Comparison of 1st and 13th generations of Intel i7 CPUs (with their release dates, 2010 and 2023 respectively) and how they handle WASM gaming and mining in terms of L1 and L3 cache operations. For the 13th gen we observed two types of games. The ones not resource-heavy and demanding ones. The disproportion is so significant, I decided to distinguish them into two columns

Not having an access to the weak machine myself, I have decided to use the results from the Minesweeper's paper, which are based on 1st generation i7 processor. To have the most reliable results, I have ran 7 different Web Assembly games and took the mean, just as it was done in the Minesweeper research. An interesting observation is that during these performance tests games could be divided into 2 groups. Ones that performed expected number of cache operations, which is fairly low and these that were very close to mining, even though both game types were tested on the same website in the same browser [81]. There are two observations here:

- L2 caches reached the capacity to take over some of the L3 responsibilities and the number of read and write operations done on L3 caches is not reliable anymore, so an improvement would be just to monitor L1.

- Regular WASM games work very similarly on both processors, using more or less the same number of operations on L1 caches, regardless of how new or old the processor is. However there are some newly released games, which L1 cache performances are very close to the mining on the old machine. To decide whether this is just because the processor is newer and can perform better or simply there is a trend in WASM online games to make more use of the processor's resources than back in the day, mining tests on the new machine have to be run.

i7 13th gen - 2023	
Operation	Mining
L1 Load	$\sim 565,600$
L1 Store	$\sim 129,000$
L3 Load	~ 51
L3 Store	$\sim 2,296$

Table 2.3: Perf while mining - number (in millions) of cache operations within 10s

The results presented in the Table 2.3 are interesting, because the newer processor turns out to handle way more L1 cache operations, than the old generation could. The numbers are up to 7 times bigger in terms of L1 cache. On the other hand, L3 behaviour is totally different than it was back in the day. The ratio of loads to stores isn't even roughly similar. Looking at this, "L3 store" seems to be the new, most reliable value to check up on.

The overall conclusion here is that, the demanding WASM games could indeed be taken for mining if Minesweeper wouldn't know what kind of processor does it have to deal with. Establishing what is the upper boundary of performance for each processor seems to be necessary, to efficiently detect mining without producing false positive results.

2.6 Summary

To sum up, Minesweeper is for sure a state-of-the-art product. The innovative ideas that the authors used to detect cryptojacking work very well, although there is always a space for improvement. A lot of questions of how could these ideas be really put efficiently into the working environment could be asked and what would

be the form of the final release, that non-technical users would be interacting with to defend themselves, however all these remain unanswered speculations. Apart from that, the authors were very conscious of the upsides and downsides of their software and even sketched out how could the further development look like and what could be improved in the future. If it happens one day, and the world will see a production-ready version of the Minesweeper, I would definitely recommend it, as a high quality detector featured with tough to evade mechanisms.

Chapter 3

Outguard

3.1 Overview

The next cryptomining detector that is covered in this work is called Outguard. Similarly to the previous one, it has been constructed and tested in 2018. The final paper covering its features however, has been published one year later in `Proceedings of the 2019 World Wide Web Conference` in May [4]. Observing the ineffectiveness of, at that time, present solutions to the existing problem of cryptojacking, the authors from *University of Illinois* and *Georgia Institute of Technology* decided to take an approach, that one could describe as similar to Minesweeper, yet very different from it. What it means, is that even though the Outguard relies on `Chrome DevTools Protocol`, just as some features of Minesweeper did, it employs Machine Learning libraries to gently decide on the maliciousness of the analyzed data.

With that being said, Outguard introduces a few state-of-the-art ideas to the world of cryptomining detectors, one being the aforementioned use of ML in the classification phase of whether the visited website is detrimental to its users or not, in a sense of non-consensual mining. However there are others, out of which the prominent ones are the innovative indicators of cryptomining behaviour. These are `PostMessage Events` happening in the browser between renderer process's main thread and Web Workers and `MessageLoop Events` coordinating tasks in each of the worker's threads. More about what it is and how it works, will be covered later.

3.2 Prerequisites

3.2.1 Basic tools

In order to run Outguard there a few tools which presence have to be fulfilled to run the project. First of all `node` and `python2` (or `python3` conversion is very simple

in this case) have to be installed, as the project consists mainly of Javascript and Python code. The Outguard's classifier uses `sklearn` Python library to have out of the box ML methods, as well as `pandas` to read `.csv` extension files easily (to install `pandas` firstly it is required to get its dependency with: `pip3 install Pyarrow`). Since `sklearn` PyPI package is deprecated, it is advised to install through `pip3` the `scikit-learn` package instead, as of 2024 at least. Moreover nowadays `pip3` doesn't like it when the environment is managed externally, so to run everything smoothly it's advised to create virtual Python environment with `python3 -m venv path /to/venv` and then activate it with `source path/to/env/bin/activate` (to exit `deactivate` command is enough). Once we have virtual environment activated, we can use `pip` to satisfy the packages mentioned earlier. Similarly we use `npm` to install `zombie` and `request` Node's packages. The first one provides fast and lightweight headless browser abstraction, the latter helps with sending requests as the name suggests.

Having these ready one is almost ready to run Outguard, with a few fixes within the code to go. Firstly, the cloned repository uses CSV files that aren't provided, so in `outguard_classifier.py`, `feature_set` and `unlabled_data` variables have to be manually set to the paths providing datasets. To "feed" the ML model with training data, the input file has to have both benign and malicious data in it, the simplest way to do it is by combining `benign_set.csv` and `crypto_set.csv` from `labeled/` directory from the source code, by for instance appending one to the other. Furthermore, the training set provided by the authors is not extensive enough, to be able to verify the efficacy of the detector quickly and not big enough to satisfy the input, hence the program doesn't converge and throws a `ConvergenceWarning`. Usually when an optimization algorithm does not converge, it is because the problem is not well-conditioned, perhaps due to a poor scaling of the decision variables [40]. Working on such a dataset and don't minding it, may end up terribly, because when the estimate of the SVM's parameters are not guaranteed to be any good, it may result in the predictions to be false. This is a serious issue, one way to resolve it is to add `dual=False` as an argument in the definition of `LinearSVC`, however for it to works, the number of samples would have to be greater than the number of features. In the provided case it's true, as there are only 7 features and more samples. Another thing to fix is `function generateFilename()` in the `resource_collection.js` file, because each URL there is being treated as if it had a scheme of `http://`, so whenever the URL provided by the user as an CLI argument is `https://` it fails. These days, most of the websites use secure http, so the quickest patch that can be applied is just by adding the `s` in the `replace` function.

3.2.2 Browser internals

Another chapter of introduction, before we delve into the Outguard and its features is about the browsers and how they internally work. Preserving the high level of abstraction we will try to understand what's the path that the information exchanged

between the main thread and other internal entities has to take. It will be essential to later get further understanding of how Outguard works and why it behaves in a way it does. Without further ado, here is the diagram showing the information cycle, it helps visualise what is going on in the browser, if mining payload is involved:

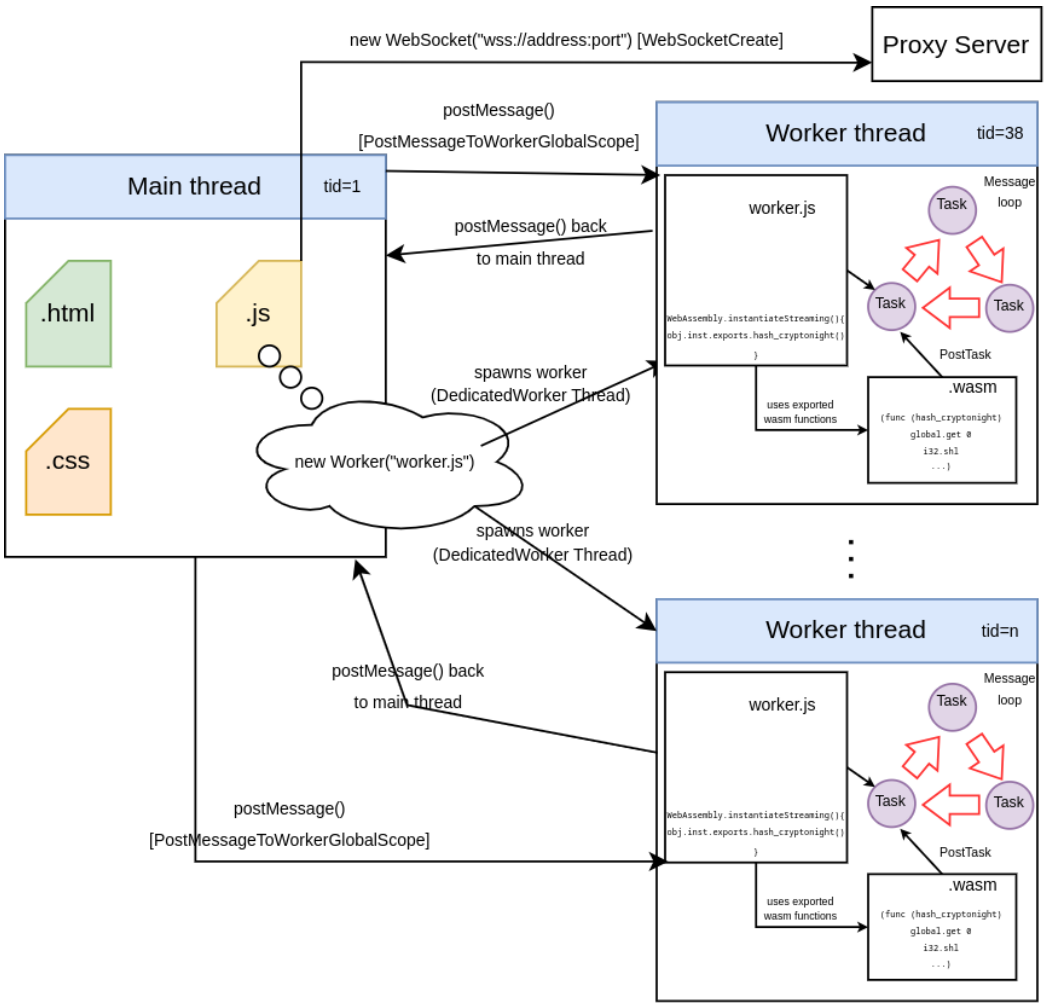


Figure 3.1: Communication between the main thread and web workers

Figure 3.1 shows a few important parts of the communication, which takes place between the main thread and workers threads, which were invoked specifically to compute something, in our case, cryptonight hashes, without blocking the UI. What it means is, that as Javascript is single-threaded, performing a lot of demanding operations would ”clog” the Call Stack, which EventLoop (an abstraction that pushes messages on the Call Stack) queue relies on and freeze the DOM. Anyways, there are a few things one should focus on reading the diagram, because they will be crucial in the next sections of this chapter.

- The communication between the threads. It happens using the events abstraction. There are two functions used in the process: `postMessage()` is for

sending the messages and `onmessage` is a listener that invokes the code in its body, once any message is received. This is important, because as mentioned earlier in the Outguard's overview, monitoring this communication channel will serves as one of the critical features of the detector.

- Secondly, each of the web workers is spawned from the main thread using the in-built, predefined constructor, that passes the JS file to the worker thread, which will be later executed.

- Another highlight is the communication between JS and WebAssembly (WASM). There are two widely used approaches here: with and without the usage of auto-generated by compiler glue-code. What is important though, is that there is no way to execute WASM code without it being spotted, either by a WASM related call frame being pushed on the stack, or by the V8 engine compiling it or the internal `js-to-wasm` function call, for each of the instantiated workers.

  (*Note: Running standalone version of cryptonight hashing would require Web Assembly System Interface (WASI), as there is a need to import* `clock_time_get` *function from WASI's API, similar to POSIX* `clock_gettime` *and this is a feature not provided by default by the browsers yet. However one could modify WASM code, so that WASI dependant functions are executed in worker's code and their returned values passed to WASM as the arguments, or simply mimic the behaviour of the WASI's function and pass as an import*)

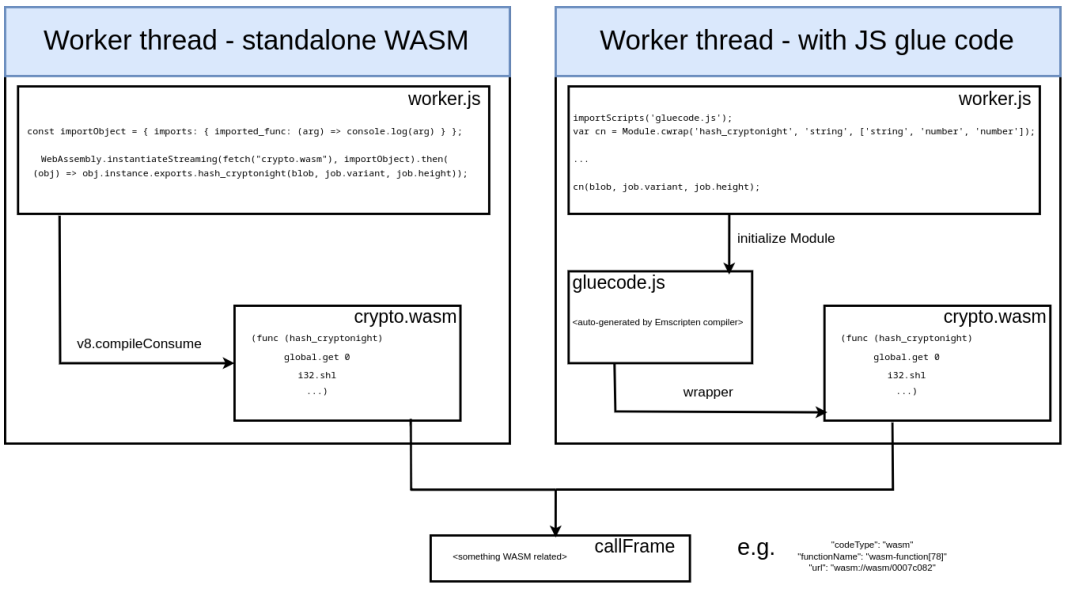


Figure 3.2: Different variants of WASM mining flow

- Next thing is `tids`. Each of the threads has its own unique thread identifier, given by a integer value, usually consecutive numbers.

- To be rewarded, mining browser has to send calculated hashes to the mining pool, it usually happens via proxy, either public or private, probably the latter choice is more frequent, since public proxies are often black-listed by e.g. ad-blockers. The intended way is to use web sockets, imitating the classic Unix system sockets. There is a `WebSocket` constructor initializing the connection.

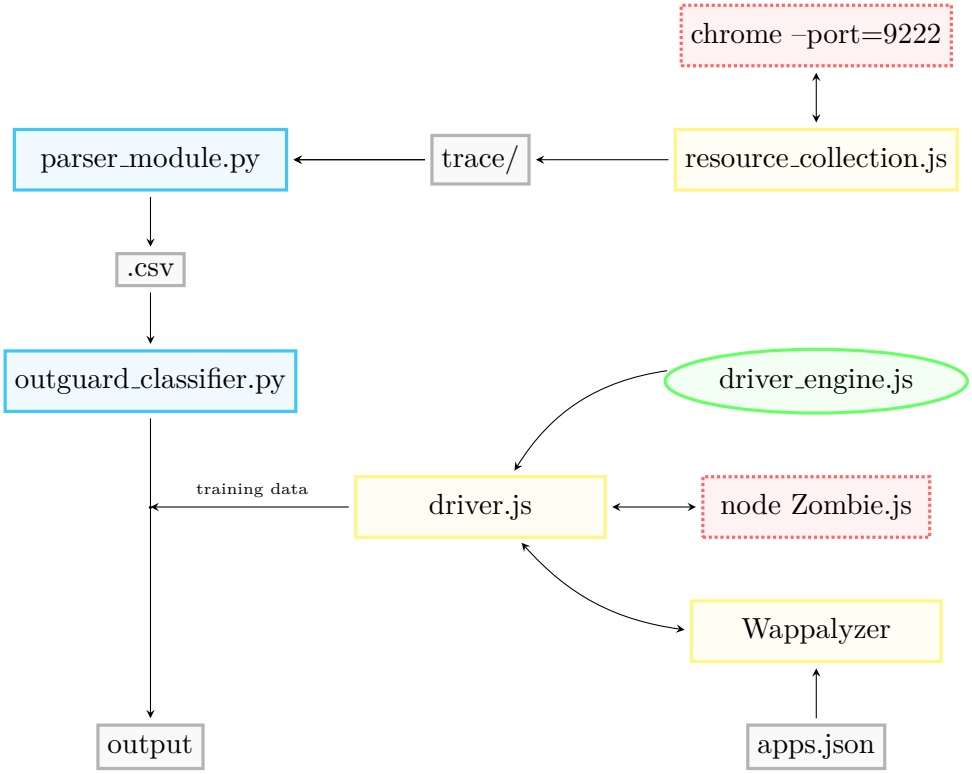
- Finally, the last highlight is the Message Loop abstraction (it has nothing to do with the aforementioned Event Loop, as MessageLoop is only an internal C/C++ code level entity in browsers, not the JS abstraction). There is at most one MessageLoop instance per thread for processing events queued up to be executed (which usually means Tasks submitted via PostTask method, or similar methods like PostDelayedTask or PostNonNestableTask) for the particular thread and fulfilling task management and scheduling of the thread. One of its key roles is to make sure that when the thread object is destroyed, all the tasks that were already queued up to be executed, will indeed be run, before the thread is terminated. There are various types of Message Loops, the `TYPE_DEFAULT` supports tasks and timers, `TYPE_UI` supporting UI events, `TYPE_IO` dealing with asynchronous IO and some others like `TYPE_JAVA` available only on Android and backed up by Java. In case of cryptomining it almost always mean tasks and default loop type. However, there are different types of tasks, when we talk about groups of tasks, we distinguish parallel, sequenced and single threaded ones. The first type prefers all tasks to be executed concurrently at once. The second are executed one at a time in FIFO order, on any thread. The last are just like the second, but on a single thread. Again in case of cryptomining we observe single threaded tasks most often. To sump up, Message Loop coordinates task execution for each of the workers thread and we can think of it not as a separate being, but more of a what the thread does, which is ”wait for an input, process it, repeat” - hence the word loop in the name.

3.2.3 About the ML used in Outguard

This work is not about Machine Learning, so devoting the whole chapter or more of the work to present thoroughly the ideas and mathematical methods of ML used by Outguard is not going to happen. Luckily, this is not needed to understand how the detector works as a whole. It is neither necessary while trying to evade it, so all that matters is that it uses two algorithms - `Random Forest` and `Support Vector Machine`, both of which are good in handling problems related to classification, in which a class label is anticipated for a specific example of input data, just like it is with deciding whether the website should be labeled (or classified) as malicious or benign, based on the already labeled input samples consisting of certain parameters (known as features). These labeled samples are the training data that the model analyses in the learning process.

3.3 Source code and features analysis

3.3.1 Code structure



Structure of the code. Python code is blue, JS is yellow, runners are green, external programs are red and files are gray.

3.3.2 Runtime and program flow

The entry point to what Outguard has to offer starts by running in the command line `python3 outguard_classifier.py`. Based on the manually provided paths to the `.csv` labeled training data and unlabeled data, the program employs two of the common Machine Learning algorithms to recognize set of features and their respective values and their correlation to the maliciousness of the website's behaviour. But to understand what this means, the comprehension of the previous phases, responsible for data collection is needed. Therefore, the analysis starts from the very beginning:

1. The way Outguard is constructed, it relies on the data previously collected, as without training data, the model has no chance to learn. Thus, the initial crawling stage can be classified not as an Outguard itself, but rather as a helper instrumentation toolkit to introduce early reconnaissance and find sufficient number of cryptojacking websites examples to work with. This process in achieved by `driver_engine.js`, which is there to run `driver.js`, which is

nothing else, but a wrapper, underneath utilizing the `Wappalyzer` open-source tool, capable of recognizing technological stack and libraries used in the process of web application development. The way it works is fairly simple:

1.1 User or automated script runs `node driver_engine.js <URL>`. All it does is, it passes the input to the `driver.js` and runs it. And once it is done it saves the returned output to `output.json` and adds the visited URL to the `checked_websites.txt` file.

1.2 Driver firstly runs its own crawling procedure, fetching recursively all the possible internal links within the website, to find e.g. subdomains (recursion depth is a parameter the user can modify). To imitate the browser, in the whole process, the program runs a node's light-weight framework called `zombie`, which visits provided links and dumps all the relevant data like: HTML, script files, all the JS and headers (scripts are the insides of script HTML tags, and JS are JS variable names). Later the Wapplyzer, based on the collected data recognizes what technologies and libraries has been used to develop the website. It does so, by finding various regular expressions matching keywords or websites associated with given library, its icon image or even characteristic metadata. All the necessary information, patterns and keywords are stored in the `apps.json` file, which is the only source of information for the Wappalyzer, hence it has to be regularly, manually updated. In short, the calling chain looks something like this:

```
analyse() -> crawl() -> fetch() -> visit() ->
Browser instance -> browser.visit() ->
get{headers, html, scripts, js} ->
wappalyzer.analyze(){url, html, meta, scripts, js} ->
wappalyzer.displayapps()
```

1.3 The most important part however, is how does that help with detecting cryptomining? Well, Wappalyzer's `apps.json` file has a separate category for cryptominers, there was (when Outguard was created) 14 different mining libraries profiled and categorized as cryptomining there. Their recognition in the website's sources, categorize website as mining performing one.

2. Once the initial crawling is done and there are some mining websites to work with, now it's the time to analyze the proper part of the Outguard. Just as the Wappalyzer-dependant preliminary crawler visited given websites, the essential Outguard's crawler does that as well - `node resource_collection.js <URL>`. The difference is that this crawler uses a proper browser, not a node module - `google-chrome --remote-debugging-port=9222`, ran with suitable command line switch, to allow runtime communication on port `9222`. This is the same `Chrome DevTools Protocol`, that Minesweeper used, what differs

though, are the CDP domains that are enabled. Outguard's runtime profiling monitors mainly `Debugger`, `Network` and `Tracing` (`Page` is used as a helper domain).

2.1 **Debugger** - it uses two methods/events, to save the source files of all the scripts loaded and executed.

- `` `Debugger.scriptParsed` `` - fired when virtual machine parses script. This event is also fired for all known and uncollected scripts upon enabling debugger.
- `` `Debugger.getScriptSource` `` - Returns source for the script with given id.

2.2 **Network** - equips two methods/events, to save the traffic going in and out.

- `` `Network.requestWillBeSent` `` - fired when page is about to send HTTP request.
- `` `Network.responseReceived` `` - - fired when HTTP response is available.

2.3 **Tracing** - monitors all the browser events happening in the runtime, it starts and stops listening, whenever the user says so, uses buffers to make sure that all the collected data is saved.

- `` `Tracing.start` `` - starts trace events collection.
- `` `Tracing.end` `` - stops trace events collection.
- `` `Tracing.dataCollected` `` - contains a bucket of collected trace events. When tracing is stopped collected events will be sent as a sequence of `dataCollected` events followed by `tracingComplete` event.
- `` `Tracing.tracingComplete` `` - signals that tracing is stopped and there is no trace buffers pending flush, all data were delivered via adequate events - `dataCollected`.

The way Chrome is designed, it allows to choose, which tracing events to gather, in the scope of Outguard's interest there are all of them. As of 2018 it was (from Outguard's code):

```
var TRACE_CATEGORIES = ["-*", "devtools.timeline",
            "disabled-by-default-devtools.timeline",
            "disabled-by-default-devtools.timeline.frame",
            "toplevel", "blink.console",
            "disabled-by-default-devtools.timeline.stack",
            "disabled-by-default-devtools.screenshot",
            "disabled-by-default-v8.cpu_profile",
            "disabled-by-default-v8.cpu_profiler",
            "disabled-by-default-v8.cpu_profiler.hires"];
```

Nowadays in 2024, some categories have changed, there are some new, like: `blink.user_timing, latencyInfo and v8.execute`, but greater majority

remained the same, which is important, from the legitimacy of comparison of outputs perspective.

Note: Even though Outguard monitors these three domains, Network and Debugger don't really matter, further analysis will mainly cover the Tracing part.

3. Once the resource collection is done, there need to be a tool to somehow process this data. Here comes `parser_module.py` responsible for the analysis. It parses all the events that happened under the hood in the browser, finds the most interesting ones and then parses it all to the `.csv` format, so it could be provisioned to ML model.

   Before this part will be covered more thoroughly however, the fundamental question: "What are the indicators Outguard looks for and uses to decide whether visited website was or wasn't calculating Proof of Work hashes for mining purposes?" has to be asked. In the previous sections browser internals were outlined, at that time it may have been not clear why would one bother to know all this and now it is finally the time to reveal it - all the highlighted points are closely related to each of Outguard's detecting features. The authors decided to take initially 12 features, monitor and measure them to see whether there is a distinction in their values between benign and malicious websites. In turned out that 5 of them didn't really work, these are:

   - JS engine execution time - noisy feature, couldn't be trusted.
   - JS compilation time - since cryptojacking libraries are usually well optimised, there was no observable difference
   - Garbage collection - could have been used as an indicator or memory usage, but no distinguishable difference could be spotted
   - Iframe resource loads - mining could be hidden in an iframe, but so could the unwanted ads.
   - CPU usage - this might have been one of the greatest indicators of drive-by mining, if adversaries couldn't throttle it to the point it's impossible to tell the difference between mining and e.g. gaming or video streaming.

   Having said that, the remaining 7 made sense and were useful in spotting cryptojacking successfully. These features are:

   - **presence of hashing algorithms** (based on their names) - out of all the working features this one is probably the easiest to evade, there are a few strings the parser looks for in the Tracing output, renaming the function is enough to evade it, however more about it in the next section.
   - **presence of WASM** - the parser looks for an indicator, that WebAssembly module has been used, as we already know, mining operations are often run in WASM, to achieve greater performance results.

- **number of Web Workers** - the more the workers, the more hashes can be computed, counting them could be very useful in determining the presence of mining.
- **use of Web Sockets** - sockets are fairly common, not only among malicious websites, monitoring them however can always provide more information about the website's behaviour.
- **parallel tasks** - the adversaries injecting malicious mining payloads into websites do this to benefit financially from the operation. It is obvious then that they will want to exploit each of their victims as much as possible. When they spawn X workers, it is almost certain all of these X threads will be calculating proof-of-work hashes. This feature is checking whether indeed each worker perform the same operations, executing the same functions.
- **MessageLoop events** - as it was described earlier, each thread has this mechanism of managing tasks execution. This feature counts how many events related to this execution take place.
- **PostMessage events** - monitoring the communication channel between web workers and main thread and counting number of messages exchanged between the two by summing up the overall duration of these events. As workers calculate the hash, they send it back to the main thread, so it could send it to the proxy. As there are hundreds of hashes evaluated by workers each second, there must be tons of postMessage events.

The question that may come to mind is how does the `parser_module.py` detect these features. There is one, main function called `process()` and in its body all the functions detecting particular features are called. The first one detects the presence of web sockets, it returns a binary result depending whether the `ws` or `wss` prefix has been spotted in the proper Tracing output table's object. The next function is detecting two features - hashing functions and presence of WebAssembly. The first one is based on searching for `Cryptonight wasmwrapper neoscript scrypt` words in Tracing's objects and WASM is looked for by finding call frames passing WebAssembly code (`js-to-wasm`). Third function is executed to decide on the use of web workers. In the Tracing output there are `_metadata` objects with an information `DedicatedWorker thread`. There is one for each of the spawned threads with purpose to serve as the workers. Next function is looking for the events, both MessageLoop and PostMessage ones. The first ones are found among Tracing objects by matching `ScheduleWork` functions ran by `thread_controller_impl.cc`, the latter by finding functions named `PostMessageToWorkerGlobalScope`. Finally, parallel tasks are considered by seeking for function names that were executed by more than one thread, which means there are more than one `tid` values among the Tracing objects that ran them. Later all the parameters returned by these functions

are saved in the `.csv` file. Presence of `WASM, parallel functions, WebSockets and hashing function` is marked binary as either 1 if the described feature took place and 0 otherwise. On the other hand `MessageLoop, PostMessage and web workers` are numerical values of counted matching references from the Tracing objects. As one can see, the detection of all of the features depends on the structure and naming conventions of the data saved in the by Tracing domain of Chrome Devtools Protocol (CDP).

4. In the end, saved data is provided to the `outguard_classifier.py`. There is a Machine Learning model there, which trains on the prepared in advance, labeled data (of both kinds, part of the data is `crypto`, the other is `benign`) and figures out, based on what it learned, whether each of the entries from the other data set, the unlabeled one, displays cryptomining behaviour or not.

3.4 Potential evasion techniques

This section describes the evasion techniques malicious adversary could consider, preparing to stay undetected against Outguard. In consists of both successful attempts and the ones that had a potential initially, but turned out not to be working as expected.

3.4.1 Parallel functions

As it was already outlined, the Outguard detects if the website that's been monitored performs the same functions across different workers threads. The way Tracing output looks for this particular feature is this:

```
{
  "pid": 2781,
  "tid": 33,
  "ts": 23678178944712,
  "ph": "B",
  "cat": "devtools.timeline",
  "name": "FunctionCall",
  "args": {
    "data": {
      "functionName": "CryptonightWASMWrapper.onMessage",
      "scriptId": "16",
      "url": "blob:https://seriesfree.to/9af8e471-74ec-4576-8261-eddc66b9d071",
      "lineNumber": 1,
      "columnNumber": 241577
    }
  },
```

Listing 3.1: Object describing function called on thread number 33

Here the red color highlights the two important things in the object. The first one is the thread identifier, which corresponds to the worker that run the function and the second is the name of the function that the detector recognises among other threads as well. Since usually all the workers are initialized from the same JS code passed from the main thread, all the workers' event listeners `.onMessage` will be spotted and therefore feature presence will be set as `True`. There is however one trick that can help hide the names of the functions and make them all unrecognizable.

I have taken one of the Monero miner's code and changed the way it defines event listeners. What it means, is that from the:

```
onmessage = (event) => {};
```

I have moved to:

```
addEventListener("message", (event) => {});
```

The difference here is that `onMessage` is a very specific `MessageEvent` listener and adds one more layer of abstraction over more generic `Event` listener, that can specify what events it wants to serve. Whenever the change is made, Tracing domain spots the events, but their function name becomes empty - `""`. Outguard cannot distinguish anymore if the same function was called across different threads or not, so the second most important feature of Outguard - parallel tasks - doesn't work anymore and always returns 0.

3.4.2 Detecting hashing functions

The evasion of this feature needs no explanation. There are a few keywords the detector looks for and if the functions are not following the very specific naming convention, the returned value is `False`. The authors are aware that this is not the strong suit of their tool and even emphasise this weakness in the paper. Any attempt of obfuscation makes this features marginalized to nonexistent.

```
1 sigs = ["Cryptonight", "wasmwrapper", "neoscript","scrypt"]
2
3 ...
4
5 if [element for element in sigs if element in d['functionName']] :
6
7     self.hash_function = 1
```

Listing 3.2: Snippet showing how the feature is coded

3.4.3 Workers interfaces

The next feature, that maybe could have been evaded was the detection of web workers. Tracing monitors the workers spawning process. Each time the main thread calls `new Worker("worker.js")`, this object is collected:

```
{
  "pid": 2781,
  "tid": 46,
  "ts": 0,
  "ph": "M",
  "cat": "__metadata",
  "name": "thread_name",
  "args": {
    "name": "DedicatedWorker Thread"
  }
},
```

Listing 3.3: Tracing object describing new worker thread

And then the parser finds it, searching for the highlighted, red-colored phrase. The question is, at what level of abstraction is this event monitored. If the CDP attaches to it on the high-level such as JS constructor, changing the way in which the web worker is spawned could be enough to evade the Tracing, however if the event depends on the low-level C/C++ code, changing the internals of the browser will be very unlikely, almost impossible.

To check it, I have used the solution publicly available on GitHub [37]. The author of this implementation wanted to introduce `Promises` to the communication between main thread and web workers. In order to implement this feature he had to get rid of the already existing solution of `new Worker()` and implement the whole interface on his own, on top of it. This work is not about how to create such a program, all that matters, is that spawning workers becomes: `new WorkerInterface("worker.js")` from now on, because it uses custom interface.

Running web applications with the new interface confirmed one of the previously stated possibilities. The Tracing is attached to low-level C++ code and modifying the interface **doesn't** work as an evasion technique. This is how the object looked like, when the website spawned the web worker with custom interface:

```
  {
  "args": {
    "name": "DedicatedWorker thread"
  },
  "cat": "__metadata",
  "name": "thread_name",
  "ph": "M",
  "pid": 1351713,
  "tid": 17,
  "ts": 0,
},
```

Listing 3.4: New worker event object while using custom interface

The snippets vary slightly, because Listing 3.3 is from 2018, in the form Outguard authors measured it, Listing 3.4 is from 2024. More about these differences in the

next section, covering possible improvements.

3.4.4 Trying to evade web sockets

Another similar story, to what has been described with the WebWorkers is with the WebSockets. The Tracing object describing its creation is looking like this:

```
{
  "args": {
    "data": {
      "frame": "55156AE5B7729101036D2389762D1726",
      "identifier": 27,
      "stackTrace": [
        {
          "columnNumber": 78,
          "functionName": "openWebSocket",
          "lineNumber": 3,
          "scriptId": "16",
          "url": "http://192.168.0.188/webmr.js"
        }
      ],
      "url": "ws://192.168.0.188:8181/"
    }
  },
  "cat": "devtools.timeline",
  "name": "WebSocketCreate",
  "ph": "I",
  "pid": 1294974,
  "s": "t",
  "tid": 1,
  "ts": 2024928247202,
  "tts": 2998840
},
```

The parser finds it because of `wss://` or `ws://` scheme, present in the highlighted, red-colored string. To evade it, I wanted to try to modify the miner's source code, to spawn WebSockets without the literal use of both sought strings. The idea was to check how in such a situation, the Tracing would react, just like with WebWorkers.

```
class CustomReplacer {
  constructor(value) {
    this.value = value;
  }
  [Symbol.replace](string) {
    return string.replace(this.value, "ss");
  }
}
var conn = "wxx://address:port".replace(new CustomReplacer("xx"));
const socket = new WebSocket(conn);
```

Listing 3.5: Snippet showing how to spawn WebSocket without ws:// and wss://

As one can see, this technique would be sufficient to evade the basic blacklisting method, however it turned out not to be enough to trick the Outguard, or more precisely the CDP and its Tracing domain. The object still consists of the proper web socket URL, nevertheless web sockets are still something that could be evaded, as there are other communication channels and protocols that could use instead.

3.4.5 PostMessage events

Communication via the `postMessage()` events is an inevitable part of how the browser operates. When the connection to the proxy server is established from the main thread and the WASM modules responsible for calculating hashes are in the worker threads, there is no other way, but to exchange the job requests and results, between these two. The Figure 3.3 presents the following situation:

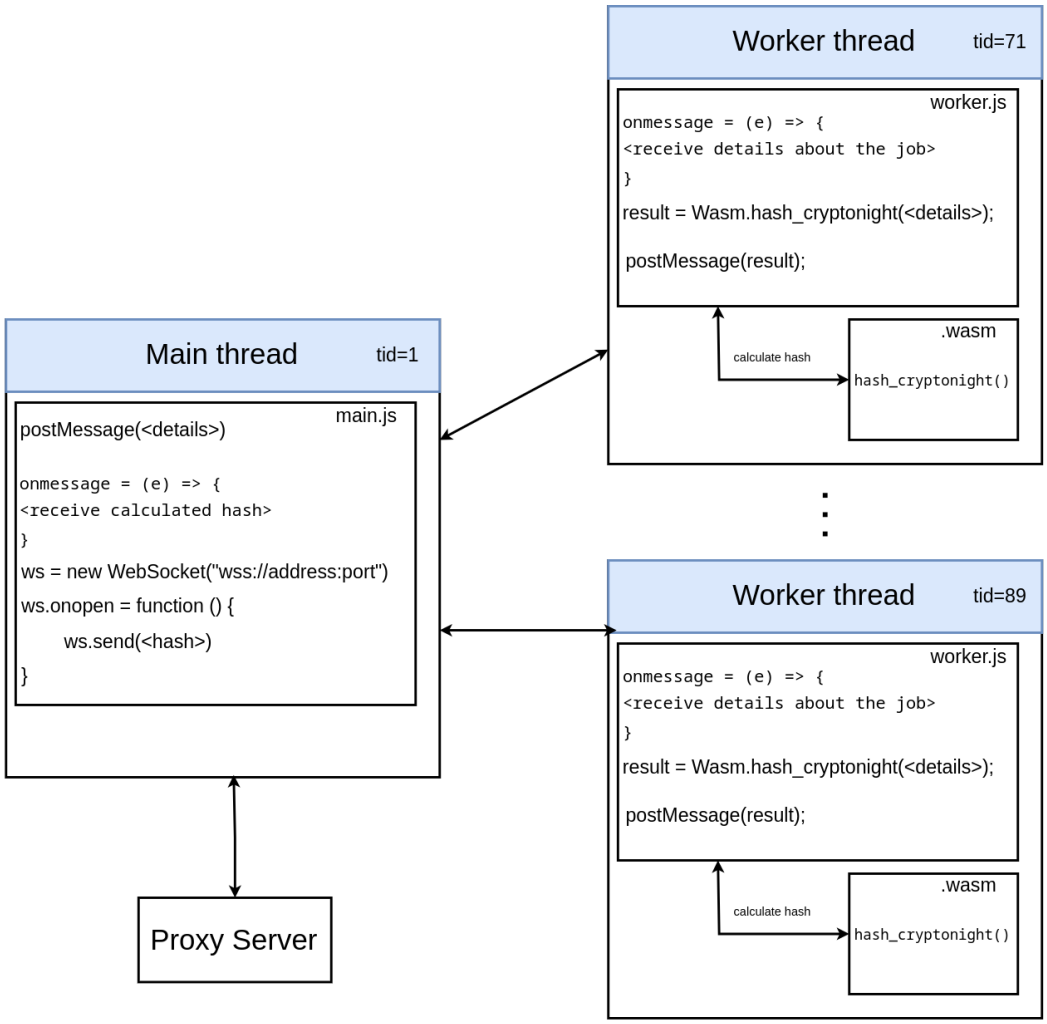


Figure 3.3: Connection to the proxy server is established from the main thread

This variant is probably more popular than the one presented in Figure 3.4 and

helps Outguard heavily in detecting mining. Number of recorded message events is the third most important, in terms of reliability, feature of the detector. To evade it, different architecture presented in the Figure 3.4 has to be used:

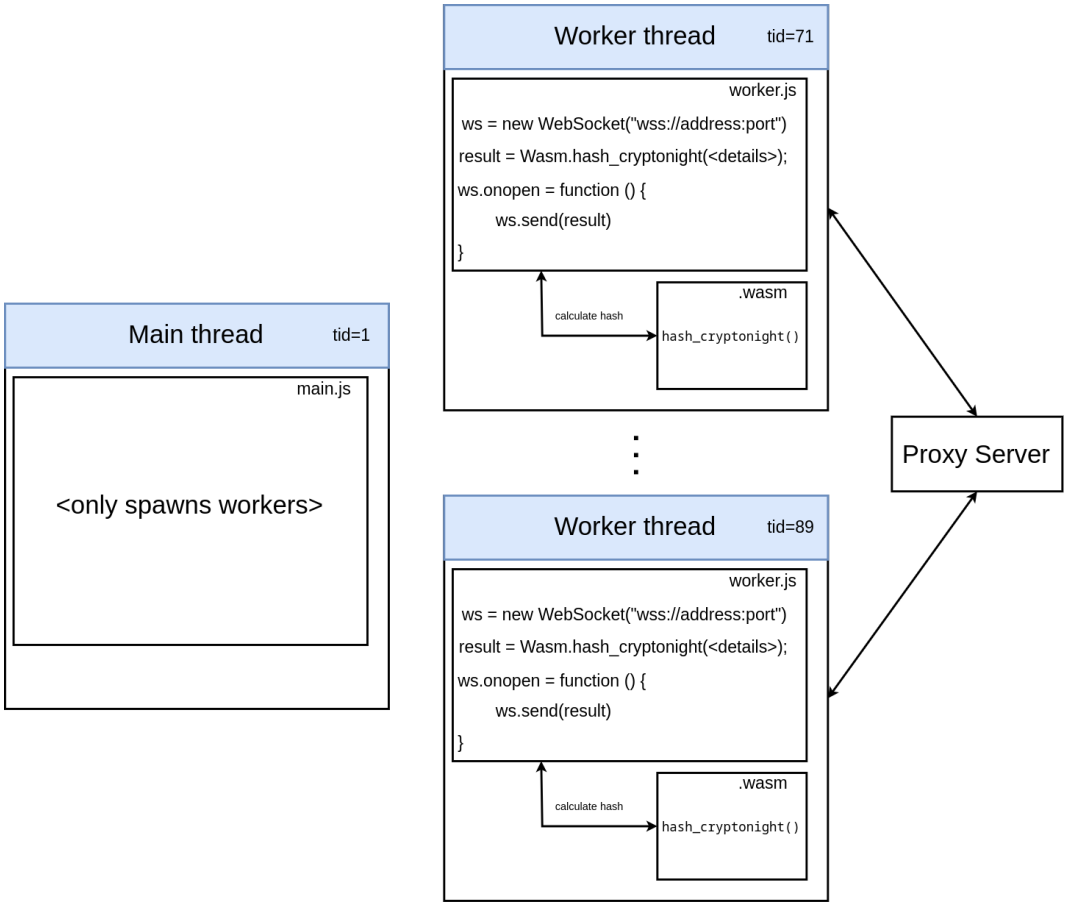


Figure 3.4: Connection to the proxy server are established from the worker threads

Proxy server should be already capable of accepting multiple connections, as there may be many victims or regular users trying to connect from various machines to the same proxy. If not, it is safe to say that each worker could have its own port, reserved for the communication with the proxy, so the server would have to have X ports open and listening for X workers. For the simplicity, but without loss of generality (proxy server is controlled by the adversary, so they can always modify it accordingly) let's assume the proxy server can operate on multiple connections. Now, WebWorkers are designed in a way, that they can create WebSockets, just as the main thread can. If, as the Figure 3.4 outlines, each of the workers opened its own channel to get the job requests and then sent the calculated results back, there would be literally zero message events exchanged between worker threads and the main thread. The responsibility of the main thread is limited to spawning the workers and then letting them do the job and handle the communication with the proxy server, limiting `PostMesssage` events to almost zero.

3.4.6 MessageLoops and WASM

The remaining features are the count of MessageLoop events and WASM presence. The first one is monitored by Tracing on a low-level and cannot be evaded in any known way, the latter differs, depending on which method is used. There is a standalone WASM and WASM with the use of JS glue-code, but there is also another partition: WASM compiled and then instantiated and instantiated streaming. All of these methods however in their own way get traced and looking at the collected data by the CDP, searching for the string `wasm` is enough to establish WebAssembly's presence or lack of it. Outguard doesn't employ this method though, it relies on finding specific pattern in the call frames - `(wasm)` in `js-to-wasm` functions. It works if multiple worker threads are spawned, which is the case in almost all mining cases. The reason why a single worker thread may be not working, is that there is a sort of race condition between spawning worker threads and attaching the Tracing domain. Often the `js-to-wasm` event is not recorded, because it happens before Tracing starts. In such cases, Outguard could return wrong answer to the question whether WASM is present in the website.

3.4.7 Data poisoning

I was aware, that there is not much I can do about tricking the Outguard in the analysis stage, as ML model is not just an `IF` statement one could force to change the boolean value of. The only idea I had is that to receive promising results, I would have to perform data poisoning, which means tampering with the data on the collection stage, so that the model would be fed with incorrect feature values. Generating entries based on which features could be evaded, I got the following results:

- changing web socket presence from `True` to `False`, the detection rate dropped to `0%`.
- modifying `PostMessage` load to values modulo 100, resulted in `20%` of mining profiles to be falsely flagged as benign, dropping this value even further to modulo 10, made the false negative rate increase to `39%`.
- flagging `parallel tasks` or `hashing algorithms` as absent in all unlabeled entries didn't change the recognition at all. Tampering with these features, to poison their values is worth only in conjunction with other parameters.

Bear in mind, I tested it against the model trained on valid data. If the training set was adjusted to the evasion techniques described in this work, the results would be better, but I decided to evaluate it as it is.

3.5 Possible improvements

As it was with Minesweeper, assumption that the tool is not a commercially developed and distributed software, but rather a proof of concept made for academic and research purposes is made. Even so, there are a few things that could be improved.

First of all the need to perform manual work, to make Outguard work properly is something that could be improved, as there is no easy way to determine website's maliciousness while entering it. The idea presented by the authors however, that the analyst job performed by human + the way Outguard works currently, could make an efficient system helping the blacklisting solutions to add new malicious sites to block-list.

There are some details regarding the source code, like:

- the `resource_collection.js` handles http websites only and throws an error for https ones
- there is no way to provide custom port number to visit the website, only default 80 and 443 work
- the user would have to install all npm modules and python requirements manually, a file satisfying the dependencies in more automated manner would be helpful

But these things are all minor and are jut pointed out, in case Outguard would be used one day publicly. There is however one more thing that makes human analysis necessary and should be borne in mind while thinking about achieving independence from manual work. The way Outguard parses and detects features in Tracing objects by `parser_module.py` is strictly dependent on the structure of the objects. The format in which CDP saves the data is constantly changing and evolving and the very specific field names Outguard uses, gets outdated. It would either require a human being analysing the the output and updating the functions each time something changes or taking more generic approach in how the parsing is done, which means usage of regular expressions and patterns that are less likely to change. To give a few examples, below on the left side there is an object from 2018 and on the right the same event captured 6 years later:

```
{
  "callFrame": {
    "functionName": "js-to-wasm",
    "url": "(wasm)",
    "scriptId": 0
  },
  "id": 36,
  "parent": 35
}
```

```
{
 "callFrame": {
   "codeType": "JS",
   "functionName": "js-to-wasm",
   "scriptId": 0
 },
 "id": 27,
 "parent": 26
},
{
 "callFrame": {
   "codeType": "wasm",
   "columnNumber": 116218,
   "functionName": "wasm-function[78]",
   "lineNumber": 0,
   "scriptId": 4,
   "url": "wasm://wasm/0007c082"
 },
 "id": 28,
 "parent": 27
},
```


As one can see Outguard detecting WASM presence by searching for `(wasm)` doesn't work anymore. Another example of how objects' structure have changed over the years is the object describing the capture of MessageLoop events, because it is a deprecated class and is being replaced by other classes, capable of doing its job. Below there is an example of how MessageLoop objects looked like in 2018 and how they look now.


```
{
  "pid": 2696,
  "tid": 1,
  "ts": 23678167974415,
  "ph": "X",
  "cat": "toplevel",
  "name": "MessageLoop::RunTask",
  "args": {
    "src_file": "../../third_party/WebKit/Source/platform/scheduler/base/thread_controller_impl.cc",
    "src_func": "ScheduleWork"
  },
  "dur": 535,
  "tdur": 533,
  "tts": 805712
},
```


Listing 3.6: MessageLoop events in 2018

```
{
  "args": {
    "src_file": "base/task/thread_pool/delayed_task_manager.cc",
    "src_func": "AddDelayedTask"
  },
  "cat": "toplevel",
  "dur": 13,
  "name": "ThreadControllerImpl::RunTask",
  "ph": "X",
  "pid": 1291581,
  "tdur": 12,
  "tid": 1291689,
  "ts": 2024922667849,
  "tts": 4568
},
```

Listing 3.7: New implementation of MessageLoop capabilities

Even though not all features' objects changed this significantly, even the slightest modification breaks the Outguard's parser. Here is another example of PostMessage event object:

```
  {
  "pid": 2781,
  "tid": 33,
  "ts": 23678178942334,
  "ph": "X",
  "cat": "toplevel",
  "name": "TaskQueueManager::ProcessTaskFromWorkQueue",
  "args": {
    "src_file": "../../third_party/WebKit/Source/core/workers/
DedicatedWorkerMessagingProxy.cpp",
    "src_func": "PostMessageToWorkerGlobalScope"
  },
  "dur": 1111019,
  "tdur": 1109767,
  "tts": 667601
},
```

Listing 3.8: PostMessage events in 2018

```
{
  "args": {
    "src_file": "third_party/blink/renderer/core/workers/
dedicated_worker_messaging_proxy.cc",
    "src_func": "PostMessageToWorkerGlobalScope"
  },
  "cat": "toplevel",
  "name": "ThreadControllerImpl::RunTask",
  "ph": "B",
  "pid": 1294974,
  "tid": 17,
  "ts": 2024942639108
```

```
12   },
```

Listing 3.9: PostMessage events in 2024

As one can see the class name change is enough to make Outguard fail, in this case by not meeting the condition in parser's `if` statement:

```
1 if raw_event['name'] == "TaskQueueManager::ProcessTaskFromWorkQueue":
```

The outlined lack of generic patterns is what prevents Outguard from being human independent. If this issue was addressed, there is a chance, the detector could be run unsupervised.

3.6 Summary

The authors came up with state-of-the-art methods, which help detect cryptojacking. The evasion of presented features is not an easy task to do and sometimes maybe be even almost impossible. Outguard takes the defensive techniques one step further and challenge the malicious adversaries to try harder. If Outguard was introduced to the wider public, as an independent crawler that recognizes mining threats and prepares a list of malicious websites, so they can be added to various, suitable black-lists, I would recommend using it. For sure there is a lot to work on and the state in which the detector is now, is not the desired one - to say it is ready to be used in production environment, but this was never an intention of proof-of-concept projects. Furthermore, the authors are aware of what they have invented, know the limitations of Outguard and at no point define it as an ultimate solution tackling all of the cryptojacking problems. Quite the contrary, in their paper the Outguard is described as a lower-bound estimate of the cryptojacking system.

Chapter 4

CMTracker

4.1 Overview

The third detector covered in this work is called CMTracker [5]. It was presented in October 2018 during ACM Conference on Computer and Communications Security in Toronto. This is exactly the same time and place as Minesweeper's publication. Researchers who worked on CMTracker come mainly from Fudan University in China. Similarly to the two previous detectors, this one is addressing the problem of cryptojacking, using fresh methods. More precisely two of them, which readers of this work should already be familiar with. First is based on recognizing hashing functions, which is a feature of Minesweeper and Outguard too and the other is associated with runtime stack profiling, which was a part of an Outguard as well. However, even though the features might sound to be related, the implementation and technical details shows that CMTracker is significantly different from the other detectors and the methods it introduces shouldn't be overlooked.

4.2 Prerequisites

4.2.1 Basic tools

Architecture of CMTracker is more complicated than the detectors described previously, as it uses two separate database servers to operate. Before the setup will be described however, here are the packages and other tools, which are necessary to be satisfied, to run the program. First of all, the detector uses `python3` and `Javascript`, as the whole project is based on these two languages. No `Python` specific packages have to be installed, however JS's `node` and its modules are essential. The modules, as expected, are specified in `package.json` and `npm install` is enough to download and enable them. However, the packages and their required versions are circa 6 years ago, so the `node` version should be downgraded as well - `nvm install`

`v8.9.0` with package manager `npm` in version `5.5.1` pulled automatically. Once it is all done, with `sudo apt-get install redis-server mysql-server` the aforementioned database servers have to be installed. MySQL server is there mostly to store the results of the operations performed by the in-built crawlers, Redis on the other hand provides the crawlers with the input data, which means the URLs that will be visited and profiled.

4.2.2 Pre-execution fixes

In `chrome_profiler/` directory, there are two files - `app.js` and `timespace.js` both of which require `./env` file to load predefined environmental variables, although the file is listed in `.gitignore`, so it is not provided. The fastest way to resolve this issue is either by commenting out these lines in each of the files or deleting them or simply by creating an empty `env.js` file.

Another unresolved issue is that, in the `config.js` file there are configurations of both database servers and chrome browser. It is specified what is the IP address of each of the servers, port the communication takes place on, database name and user credentials the program will use to login to the DB. Whenever the user is created on the DB end and saved in the configuration file, CMTracker fails to run. The reason is, that the node modules are old and are not compatible with the newest MySQL server anymore (with any server version 8.0+). The program throws

```
Error: ER_NOT_SUPPORTED_AUTH_MODE: Client does not support
    authentication protocol requested by server
```

and I couldn't find a way to make it work with a custom user. The solution that works however is to login in to the database as `root`. To let the program use the privileged account, these two commands could be run to set up the root's password and refresh the privileges.

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY
     'password';
flush privileges;
```

Assuming the database server is run locally of course, which was the case in my set-up.

Another change I have made in the `config.js`, was to set the remote debugging port of Chrome to `9222`, which I have used in the previous chapters as well and changed the path of google-chrome location, from the absolute one to relative. Moreover I have downgraded Chrome from the latest version (122 major) to 102 major, because of: `Error: Using unsafe HTTP verb GET to invoke /json/new. This action supports only PUT verb.`

4.3 Source code and features analysis

4.3.1 Architecture

The architecture of the CMTracker is as follows:

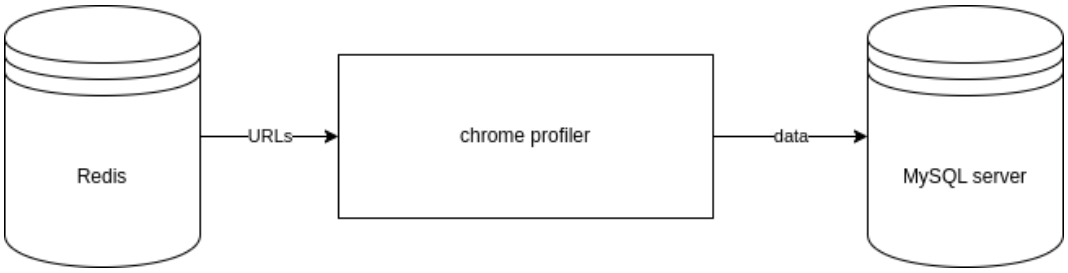
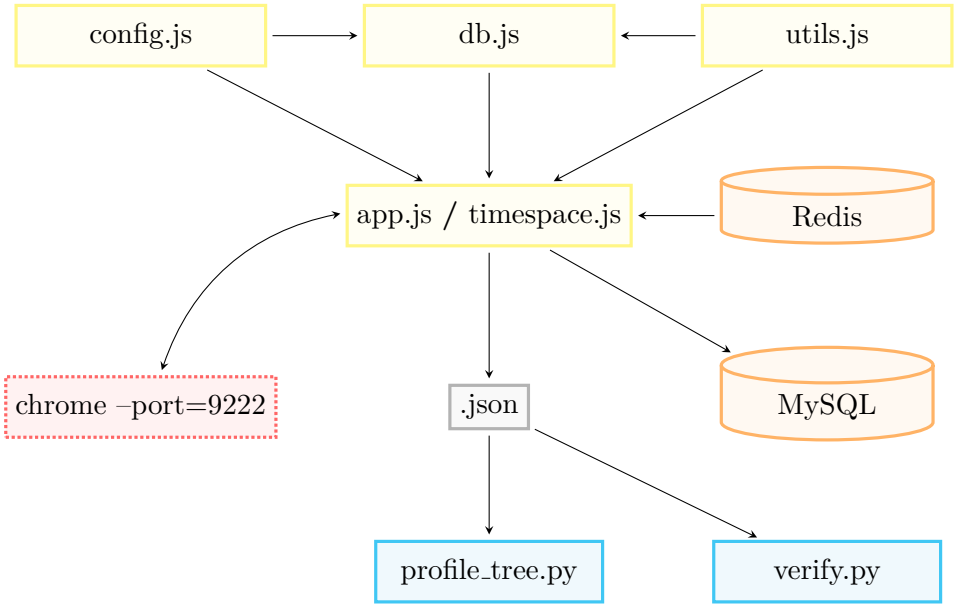


Figure 4.1: Dataflow in CMTracker

Both `app.js` and `timespace.js`, which are the chrome profilers take URLs from the Redis database, perform their operations and later save the output to MySQL instance.

All the operations covering fetching from the DBs or updates/inserts are described in a separate file - `db.js`.

4.3.2 Code structure



Structure of the code. Python files are blue, JS is yellow, external programs are red and DBs are orange.

4.3.3 Databases preview

1. **Redis** - Just as the previous detectors worked, CMTracker takes URL as an input. This time however it is not provided by the user as a command line argument, but in a form of Redis entry. The simplest way to run the detector is to manually open `redis-cli` and push the URL of the website to the Redis list, which works as a queue, where elements are added to the head. It can be done like this: `lpush list_name '{"key":value,"key2":"value2"}'`. Each time CMTracker is run, the `node` Redis client logins to the local (in my case, could be remote as well) server and fetches the elements of this list. The default configuration is to pull as many as 1000 entries from the DB. What is important however, when fetched the entry is popped from the list and would have to be added again each time the program is run. There are two Chrome profilers - `app.js` and `timespace.js`. What they are and how they differ in their runtimes will be covered later, what is worth pointing out as for now, they use different tables and lists in the DBs. On the Redis end:

   - `app.js` - uses list called `to_profile`. To add an entry which will be parsed and properly executed by this profiler, this command would have to be run on Redis server:

     ```
     lpush to_profile '{"id":1009,"url":"http://192.168.0.188"}'
     ```

   - `timespace.js` - uses list called `timespace`. It requires more arguments, compared to the other one. To add entry, this line is sufficient:

     ```
     lpush timespace '{"id":1008,"url":"http://192.168.0.188", "
         firstSeen":2137, "round":3}'
     ```

   This is not documented anywhere, hence code analysis had to be made, to figure out what the input could be.

2. **MySQL** - Once the website is fully profiled and the data is collected, there is a need to store it somewhere, to get rid of the temporariness. As it was with Redis, there is no README or any other documentation, to explain to the user, what tables with what columns are needed to successfully run the detector. This section is specifically to complete this missing part.

   Firstly, the specific database has to be created - `CREATE DATABASE tracer;`. Inside there are four tables, which existence has to be satisfied to run the profilers properly. Again, each of the profilers relies on on a different MySQL DB layout. On the left side of Figure 4.2 there is a table for the `app.js` profiler, on the right side, the remaining three snippets describe what is necessary to test the other profiler - `timespace.js`.

```
CREATE TABLE profilerUrl
(
  id INT,
  url VARCHAR(255),
  finishTimeStamp DATETIME,
  status INT,
  threads INT,
  webSocket INT
);
```

```
CREATE TABLE rerunHistory
(
  profilerUrlId INT,
  url VARCHAR(255),
  cat VARCHAR(255),
  init VARCHAR(255),
  sourceUrl VARCHAR(255)
);

CREATE TABLE timeSpaceHistory
(
  profilerUrlId INT,
  time DATETIME,
  url VARCHAR(255),
  cat VARCHAR(255),
  sourceUrl VARCHAR(255),
  init VARCHAR(4096),
  frames VARCHAR(4096),
  requestId INT
);

CREATE TABLE timeSpaceVisit
(
  id INT,
  threads INT,
  timeStamp DATETIME
);
```

Figure 4.2: Necessary tables to run CMTracker - based on source code analysis

4.3.4 Runtime and program flow

Knowing how to set-up the database servers, it is finally time to take a look at how to run the profilers and based on their output how to decide on the maliciousness of the websites.

1. At the beginning, the entries to Redis are provided. Once it is done, user runs one of the two Chrome profilers. Both of them use `config.js` to know how to connect to the databases and where to look for Chrome, `db.js` code to handle these DB operations and `utils.js` to format dates. As both tools are totally independent from each other, the order in which they will be described doesn't matter.

   1.1 Let's assume first one to be covered is `app.js`. Executing it is as simple as e.g.: `node app.js -N 3 -D ../output-dir`. The first flag tells the program

to visit 3 tabs only (for testing purposes even one would be enough), the other one determines the directory collected data will be saved to.

1.2 The first step is to parse the arguments, initialize environmental variables such as Chrome flags and create sub-directories in the output directory. Later the URLs from Redis are fetched and the crawler is started to visit the given website - function `newTab()`. There, yet again, as in the Minesweeper and Outguard, `Chrome DevTools Protocol` is used. New Chrome Remote Interface client instance is spawned, with a few well-known domains. These are `Debugger`, `Network`, `Page`, `Runtime`, `Target` and `Profiler`. Most of these domains were already covered in the previous chapters and their use in this case is very similar. The first four that were just listed are used as auxiliary domains, for instance for opening the tab, effectively loading the target website, gathering network traffic, letting the client know that the scripts have been already parsed by the browser or figuring out how many threads can the CPU handle. `Target` is crucial to "send" the other domains to the workers. By default, the monitoring is done in the main thread that the protocol is attached to, so in order to monitor the behaviour of Web Workers, `Target` has to send a message, which will invoke monitoring there. Finally, there is the last domain called `Profiler`. It is run both in the main thread and it is passed to each of the workers. Its job is to profile the call stack, by sampling it with the given interval. It allows to understand what functions were present, which were called, how many times and how long each execution took. After the time runs out (default timeout equals to 8 seconds) the monitoring stops, the domains are handled to exit gracefully, recorded data is saved to both MySQL DB and output directory, and finally the browser instance is killed and DB connection is closed.

2. Alternatively user can run `timespace.js`. Its code is almost the same as the one of the previous profiler. It employs `CDP` with the same domains, it fetches data from Redis too and saves the output in MySQL as well. The biggest and the only major one is in the output saved to the database. There is a lot more information amassed. The reason for this is that this tool was used by the authors to periodically crawl over the internet (Alexa Top100k) and check how lifetime of the cryptojacking websites looks like. Saving timestamps and very detailed information about websites' behaviour enabled them to check what is the status of each profiled website at regular intervals and observe how often malicious websites disappear, what the authors outlined in their paper (about 20% vanish in less than 9 days and the life cycle of circa 1/3 of them ends within 15 days).

3. Once either one of the Chrome profilers or both have been run (or the user has the data from any other source e.g. 200 mining `.json` samples have been provided in the `profiler_analyzer/profiler_test_case/` directory by the

authors). It's time to call `profile_tree.py` to evaluate the information collected by the CDP's `Profiler` domain. The program starts with user running `python3 profile_tree.py <path>`. It could be either a path to a single `.json` file or a path the the whole directory, in the latter case all the files inside are treated as an input. Then the `Tree` structure is initialized, because CMTracker wants to keep track of the calling tree, which basically means which function called which, just as it was with Minesweeper, however this time it is about whole website's code, not just WASM.

```
{
    "nodes": [
        ...
        {
            "id": 11,
            "callFrame": {
                "functionName": "wasm-function[56]",
                "scriptId": "4",
                "url": "wasm://wasm/0007c082",
                "lineNumber": 0,
                "columnNumber": 103834
            },
            "hitCount": 2,
            "children": [
                12,
                32
            ],
            "positionTicks": [
                {
                    "line": 1,
                    "ticks": 2
                }
            ]
        },
        ...
    ],
    "startTime": 91488436966,
    "endTime": 91496449124,
    "samples": [
        9,      // these are function IDs from the "nodes" array
        9,      // this "samples" array is bascailly a call stack
        ...
        ],
    "timeDeltas": [
        20444,
        3763,
        156,
        ...
    ]
}

```

Listing 4.1: Example of the Profiler domain output object

As one can see Listing 4.1 presents the format in which `Profiler` saves the data. Each function found in the runtime on the call stack, has its parameters. Beginning from unique identifier, name or internal browser URL, through `hitCount`, meaning how many times it has been called, ending on highlighted `children` field, which is used to build the aforementioned tree. Once the `Tree` object is built and a very clear picture of what happened in the runtime is obtained, the first detector is executed.

As it was already mentioned in the overview, CMTracker uses two approaches two detect cryptojacking:

- **hash-based recognition** - the first detector searches for hashing functions in the `Tree` object and if it finds a node storing hashing function it marks this node's `isHash` flag to `True`. The question that may come to mind is why can't this happen by going through each of the nodes in a form of a list, just as it originally is in the `.json Profiler's` output. The answer is that if node `X` performed hashing operations, the CMTracker's logic is to mark all of its children, hence, recursively all of its sub-tree (the sub-tree, which is rooted in the node `X`) as performing hashing operations. This is crucial in determining the final result, the detector produces, because the threshold to recognize cryptomining is 10%, which means if a web page uses more than 10% of its execution time on hashing, CMTracker reports it as a miner. This is why it matters which functions are tagged as hashing ones, to then count their runtime.
- **stack-based recognition** - just after the first detector execution is finished, the `profile_tree.py` moves to run the second detector. This time again, the same `Profiler's` output and the tree built earlier are used. What is different though, is the details the program pays attention to. Previously it was hashing functions, now it is the execution time of the functions repeated in the same order over and over again, constituting for more than 30% of all the runtime. In other words if a specific call chain, run periodically, represents more than 30% of the whole execution time, such a page is considered to perform cryptomining. Counting how many times each function has been run is already done by the `Profiler` (`hitCount`), so basically what is left to be done is finding whether there are functions forming a chain, which is run many times, following the same order of execution.

  Before the chain is looked for however, the detector does a few steps:

  - it calculates the total number of samples, which is nothing else, but the sum of `hitCount`-s of all functions
  - then for each function (node) it evaluates the share this function represents (percentage of all samples), because the threshold to detect cryptomining was 30%, it is important to have this information

– it omits to investigate internal Chrome nodes (functions that are recorded by the `Profiler`, but are related to browser's native code such as for instance: `(idle)` or `(program)`). This will be important later in this work.

Finally, when the program knows all this and finds a function that is run most often, it checks whether it forms (is a part of) a repeated call chain. To find it, the detector iterates over the `Profiler`'s output list and more precisely through an array called `samples` (it consists of function nodes' ids), forming a window of size 2 to 5 elements and saving each occurrence of the repetitive pattern. For the chain to be considered as valid, it must include the most encountered node from the stack and the repeated window must constitute half of all the considered samples (samples with `hitCount` over 100). In other words, if an array of length `N` of all the samples with sufficient `hitCount` is examined, any pattern of length 2,3,4 or 5, which occurs `X` times is valid, as long as its length multiplied by `X` is greater than `0.5` times `N`. (and this exact window pattern consists of the most popular function from the whole runtime).

4. In the end, there is a `verify.py` file left, which can but don't have to be run. It builds a wrapping layer over `profile_tree.py`, to run both detectors against the given input directory and calculates the efficiency of each of them, which means out of all the input samples (an assumption is all the samples are malicious), how many were correctly recognized as mining, and how many returned false negatives.

4.4 Potential evasion techniques

Knowing how the CMTracker works, it is finally the time to trick it in to believing, that hazardous mining websites are benign, at least based on its recognition, by suggesting some of the potential evasion techniques tailored for both of CMTracker's detectors.

4.4.1 Hash-based detector or black-listing detector

In the previous section hash-based detector was described. The focus there was strongly put on how the `Tree` structure is built, why is this object helpful in the latter analysis, but not much has been said about how exactly the node is processed to decide on whether the function is performing hashing. The reason is, that revealing this information earlier would automatically give away how to evade it. It is as trivial as:

```
1 if any(word in node.function_name.lower() \\
2 for word in HASH_LIBRARY_LIST) or \\
```

```
self.nodes_tree[node.parent].isHash is True:
    node.isHash = True
```

Listing 4.2: Detecting hashing in functions

As mentioned before the `isHash` value is inherited by node's children, but the the first condition in this statement is more interesting. The function name, case insensitive, has to match any of the words from the `HASH_LIBRARY_LIST` list. This array is:

```
HASH_LIBRARY_LIST = ['hash','sha256','cryptonight','_mLoop']
```

Effectively, it means that regardless of advanced profiling methods and complex `Tree` structures, the whole idea comes down to the blacklisting approach, a very simple one, without any regular expressions or extended list of forbidden words. It all hangs by a thread, depending on function names, which can effortlessly be changed to whatever the malicious actor please, resulting in detector's 0% efficacy.

4.4.2 Stack-based detector - test of time

The situation is different however with stack-based recognition of mining. This detector is undoubtedly more reliable and tougher to evade. Nevertheless, there are ways to trick it and here two of them are presented.

Chrome Profiling - native code case

The first technique relies on the analysis of the layout and contents of the `Profiler's` output. There are a few important concepts that need to be outlined first, to get a better understanding of the matter. First and foremost there are various function types monitored by the browser:

- **Javascript functions** - each o them has a unique name, the same one it got from the author of the JS code it comes from, e.g.:

```
"id": 8,
"callFrame": {
    "functionName": "CryptonightWASMWrapper.onMessage",
```

- **WASM functions** - these come from `.wasm` file and don't have names. Each of them is marked as a consecutive natural number, they are indexed by the browser, e.g.:

```
"id": 9,
"callFrame": {
    "functionName": "wasm-function[48]",
```

- **internal functions** - native Chrome functions. There is a few of them, regardless of the visited website they can be found on the call stack, e.g.:

– `(idle)` - the state of idleness occurs when the browser has not yet completed the final rendering of the page on the screen but must put the process on hold while waiting for missing data or resources needed to resume and complete it. The `hitCount` value of this function is usually significant and can make up to over `99%` of all the recorded samples. There is however one major disclaimer to be made here: **(idle) function only occurs in the main thread, worker threads do not have it.**

```
"id": 3,
"callFrame": {
    "functionName": "(idle)",
```

– `(program)` - this function represents native code. Back in the day it included idle time as well, but not anymore. Could be found in both main and workers threads.

```
"id": 10,
"callFrame": {
    "functionName": "(program)",
```

– `(root)` - the root of the tree, it is always the first node (the parent for other functions, such as `(idle)`, `(program)` or JS functions). Indexed in the array of function nodes as `id=1`. It doesn't really matter in terms of `hitCount` values counting, but it is always there, so probably should be mentioned. It occurs for every thread.

```
"id": 1,
"callFrame": {
    "functionName": "(root)",
```

– some other functions, which can be found in the Chromium source code, but are not really important from this work perspective. Here is the list of all of them:

```
const char* const CodeEntry::kProgramEntryName = "(program)";
const char* const CodeEntry::kIdleEntryName = "(idle)";
const char* const CodeEntry::kGarbageCollectorEntryName = "(
    garbage collector)";
const char* const CodeEntry::kUnresolvedFunctionName = "(
    unresolved function)";
const char* const CodeEntry::kRootEntryName = "(root)";
```

Knowing all this, let us review how this contributes to the evasion technique. The reason is that usually (almost always) mining payloads keep executing the same function. It is exported function from WASM module calculating hashes, that could be named inside WebAssembly code `cryptonight_hash()` for instance or similar. It later uses other internal functions, such as `AES` or `Keccak`, but the profiler doesn't see them. Each time I have tested CMTracker against miners, there was only one function from WASM exceeding the required `hitCount > 100` (to be even considered a potential element of the repetitive chain), other than internal functions, and to be precise only one of them - `(idle)`. As we already know however, idleness is:

1. not present in the worker threads, where mining happens and it is useless from the main thread perspective, because there is no mining there

2. it is not correlated in any way with mining, every single other website e.g: `bbc.com` had `(idle)` function `hitCount` on a indistinguishable levels.

resulting in this function being useless to take into consideration. What is interesting though, is that the authors were aware of it and created a condition, not to even count the samples of this function into the counter of all samples, if it is found on the call stack:

```
if node['callFrame']['functionName'] != '(idle)':
    totalSamples += int(node['hitCount'])
```

There is a problem however, because later in the code, when the chain is looked for, there is no sign of another condition that would exclude `(idle)` samples from being considered. This is clearly a misconception, because it should be checked twice and in both cases skipped. To prove it, here is an example straight from the directory containing the data provided by authors:

```
$ python3 ../profiler_analyzer/profile_tree.py ../profiler_analyzer/
    profiler_test_case/18.json
6,7
```

In this case I have modified the source code, to print the ids, of the nodes, that were matched in the window together and considered a repetitive chain on the call stack. If we look at the `Profiler's` output what they are:

```
{
    "id": 6,
    "callFrame": {
        "functionName": "17",
        "scriptId": "0",
        "url": "",
        "lineNumber": -1,
        "columnNumber": -1
    },
    "hitCount": 32796
},
...
    "id": 7,
    "callFrame": {
        "functionName": "(idle)",
        "scriptId": "0",
        "url": "",
        "lineNumber": -1,
        "columnNumber": -1
    },
    "hitCount": 3740
},
```

it turns out one of them is indeed `(idle)`. This worked, because function named `17` (in 2024 it would be `wasm-function[17]`) is indeed calculating hashes, but this is illogical and more importantly incorrect approach. There is a reason why `(idle)` was excluded in the first place (when counting all the samples), not to mention that it works for main threads only.

Another problem with stack-based detecting is that even if we forget about `(idle)`, there is one more internal function present. It is the `(program)`, although I mentioned earlier that it doesn't qualify to be considered as a part of the recurrent call chain. The reason is, nowadays, in 2024 `(program)`'s `hitCount` for all miners that I tested is approximately between 25-80, which is below the magic boundary of `100`, to add it to the pool of potential mining nodes. For unknown reasons, when authors tested CMTracker it was different, `(program)` number of executions floated around `15 000` (similarly to `(idle)` and WASM mining functions). The reason why it changed, might be that the `(program)` used to contain more information that is does now. Just like back in 2012 when idleness wasn't counted separately, it was a part of `(program)`. It did change in 2014 and maybe, something similar happened between 2018 and 2024 and another internal part of `(program)` function was separated. The consequence of these actions is that now `(program)` cannot be counted to the chain anymore, even though I am fairly confident it should have never been, for the same reasons as `(idle)` - internal functions of Chrome, shouldn't be something detector relies on. It should be the behaviour of the visited website and the resources it utilizes, that matters, not how browser handles native code of its own or how it performs in rendering.

Sliding Window size

Having said that, here comes the second problem. As already mentioned `(idle)` doesn't exist in threads other than main and `(program)` stopped being useful in looking for recurrences of similar patterns on the stack, because its `hitCount` dropped so significantly over the years, that it is not taken into account by CMTracker. Excluding these two functions from the valid nodes, it leaves us with one function meeting the criteria - Cryptonight hashing WASM function in miner cases. There shouldn't be a difference for CMTracker whether the behaviour of the internal functions changed, as still clearly one function takes up over 99% of the call stack and it is way over the threshold - 30%. But isn't really it a problem? It turns out it is, because even though CMTracker correctly finds hashing function and recognizes its significant share of all the samples, it cannot find a valid window containing the pattern including this function. The answer why is trivial: minimal size of the window is 2 and since there are no other functions among all samples meeting the `hitCount` condition, it is all a chain consisting of one repetitive execution. The window size of size two is hence too wide to find a place where hashing function would pair with another function to create a short 2-elements chain, it is just 1-element chain all

runtime long. Being unable to find the matching pattern, stack-based detector exits flagging malicious website as benign, which means it failed to correctly recognize the risks, due to flawed implementation.

4.4.3 Too many nodes - three approaches

The last approach to trick CMTracker is based on the unclear (for me) condition of:

```
if self.calc_periodic(maxConcentratedIdx) == False or len(self.nodes)
    > 100:
    return 0
```

in the source code of `profile_tree.py`. There are two conditions in the snippet, one is checking what is the value returned by the stack-based profiler, the other is checking how many nodes were recorded by the CDP `Profiler` domain. If there is over 100 of them, which means that over 100 functions were run, the program automatically fails, which would stop the stack-based recognition completely dead in its tracks. I have tried to create an environment in which this condition would be satisfied. There were three approaches to the idea of creating abundant functions:

1. **array of JS functions** - the first one, the simplest to code was to add some additional functions and see whether their presence affects `Profiler` and its output. Here is the snippet creating and executing an array of 60 functions (added later to the mining payload), as over 40 were already present and recorded in the miner, to make it over 100 in total.

```
const functionsArray = [];
for (let i = 1; i <= 60; i++) {
    let functionName = 'x' + i;
    functionsArray.push(new Function(`return function ${
    functionName}() { return ${i} + 2; }`)());
}
for (let i = 0; i < functionsArray.length; i++) {
    console.log(functionsArray[i]());
}
```

2. **GPT generated functions** - the other approach was to generate all these functions and add to the miner's code manually. With help of GPT, here are some various alternative ways that I have tried. *Note: Even though the snippets are shortened, there were 60 functions in total in each of them.*

   - The first method runs the functions straightaway:

```
function x1(){return 1+1;}function x2(){return 2+1;}function
    x3(){return 3+1;}function x4(){return 4+1;}function x5(){
    return 5+1;}function x6(){return 6+1;}function x7(){
    return 7+1;}function x8(){return 8+1;}function x9(){
    return 9+1;}function x10(){return 10+1;}function x11(){
    return 11+1;}function x12(){return 12+1;} .... // so on
```

```
x1();x2();x3();x4();x5();x6();x7();x8();x9();x10();x11();x12
    (); ....
```

- The second uses timeout, to let the rendering finish and the `Profiler` start:

```
setTimeout(() => x1(),3000);setTimeout(() => x2(),3000);
    setTimeout(() => x3(),3000);setTimeout(() => x4(),3000);
    setTimeout(() => x5(),3000);setTimeout(() => x6(),3000);
    setTimeout(() => x7(),3000);setTimeout(() => x8(),3000);
    setTimeout(() => x9(),3000);setTimeout(() => x10(),3000);
    setTimeout(() => x11(),3000);setTimeout(() => x12(),3000)
    ; ....
```

- Finally the last idea was to run all 60 functions at intervals:

```
setInterval(() => {x1();x2();x3();x4();x5();x6();x7();x8();x9
    ();x10();x11();x12();...}, 50);
```

3. **recompiling WASM module with extra functions** - the very last, third, approach was to leave JS alone and hide these abundant functions in the WebAssembly code.

```
#include <iostream>

extern "C" {
    int A1() {
        return 1 + 1;
    }

    int A2() {
        return 2 + 2;
    }

    int A3() {
        return 3 + 3;
    }

    int A4() {
        return 4 + 4;
    }

    int A5() {
        return 5 + 5;
    }

    int A6() {
        return 6 + 6;
    }

    ...
}
```

Listing 4.3: fact.cpp file with functions

I have created 101 functions this time, because instead of mining WASM script, I wanted to serve this `.wasm` file only. Once I had all the functions, I have compiled it with Emscripten like this:

```
emcc fact.cpp -s ENVIRONMENT=worker -s MODULARIZE=1 -s
    EXPORTED_FUNCTIONS="['_A1', '_A2', '_A3', ... '_A100']" --no-
    entry -o fact.wasm
```

And later I could create an instance of WASM in the JS script and run these exported functions (why at intervals, will be covered later):

```
var importObject = {'env': {'memoryBase': 0, 'tableBase': 0, '
    memory': new WebAssembly.Memory({ initial: 256 }), 'table':
    new WebAssembly.Table({ initial: 0, element: 'anyfunc' })}};

WebAssembly.instantiateStreaming(fetch("fact.wasm"), importObject
    ).then((obj) => {
  const x  = obj.instance.exports;
  setInterval(() => {
    console.log(x.A1());
    console.log(x.A2());
    console.log(x.A3());
    console.log(x.A4());
    console.log(x.A5());
    ...
}, "1");
});
```

All these methods were tested for a reason, the reason being, all previous ones were unsuccessful, because none of my ”abundant functions” was present in the `Profiler`'s output and I have been trying various approaches, to maybe finally find a solution. It turned out in the end however that there is no solution and to understand why such a situation takes place, the difference between `CDP Profiler` and `Tracing` has to be comprehended.

- **Tracing vs Profiler** - one could ask what is the difference between the two, as Outguard was based on `Tracing` domain and it also monitored and dumped the state of the call stack. Well, `Profiler` is a **sampling** profiler, which takes execution stack snapshots at a predefined interval, in CMTracker's case every 100ms. When it is about to do so, the JS execution is paused and functions currently on the stack are dumped. On the other hand `Tracing` is designed to record all functions, not just samples. It works by instrumenting the code by wrapping each and every function call with measuring code.

Knowing this missing piece of information, lack of a solution is understandable. It is tough to predict the exact time, the `Profiler` dumps samples from the stack, so running ”abundant functions” only once, regardless of the method is not enough to have them recorded. Halfway solution may be to to run them at intervals, with a very

short timeout, but this may take too much of a runtime, leading to inefficient mining - hashing function execution will be constantly interrupted by useless functions. To sum up this section, it may be possible to satisfy the condition of over one hundred nodes, which consequently breaks the detection mechanism of CMTracker, but in order to achieve it, the efficiency of cryptojacking operation may suffer.

4.5 Possible improvements

Once again, as it was stated in the previous chapters, the idea to outline potential improvements is focused solely on the features that are already implemented in the project. There is no point in listing missing properties, as the project is only a proof-of-concept version of what could a commercial product look like.

4.5.1 Hashing detection enhancement

The most evident misunderstanding that happened in the development of the CMTracker is the way it handles hash-based recognition of mining. Reading the paper, it seemed that advanced methods has been used to make sure that malicious adversaries will be forced to make great efforts to evade it. When it turned out that the detection is as operational as a black-listing approach of 4 words, it was surprisingly basic method. A better idea how to distinguish benign functions from the hashing ones, would be to make the recognition more generic by e.g. by using regular expressions. Furthermore, it could have been at least stated directly in the paper, how this works, not to give color to the effectiveness of the CMTracker.

4.5.2 Stack-based detection logic

In terms of the other detector, the situation was way better, but still could be improved. As presented in the previous section, stack-based recognition relies on the window of the given size to look for a repetitive pattern within its boundaries. The minimum size was coded to be 2 and because of it, the function taking more than 30% of the runtime, but not being a part of any chain is not recognized as performing hashing operations. Resizing the window to the minimum length of 1, would enable CMTracker to recognize more valid cases of cryptojacking, without having to worry about false positives, as the threshold of mining remains the same.

4.5.3 Counting threads

Both in `app.js` and `timespace.js` there is a snippet of code, which is responsible for finding out what is the concurrency CPU can handle. Typing `navigator.hardwareConcurrency` manually in the browser's console returns maximum number of

threads, that can be run simultaneously on the given hardware. As presented in section 1.4.1, `CDP` allows to perform the same operation using its domains - `Page` and `Runtime`. Looking at the source code and analysing the runtime behaviour however, one concerning issue arises. Regardless of the CPU and how many threads the website utilizes, the `CDP` attaches to `4` of them only. It might be considered to be enough to profile visited page, but later CMTracker saves the gathered data to the `MySQL` database and there one of the fields says how many threads were present. If the application is single threaded, it correctly saves the output to use one thread only - `threads=1`. But when the web application runs 10 threads, information saved says it is only five - `threads=5` (main one + 4 workers). Even more worrisome is that modifying the source code to return e.g. `10` instead of `4`, saves to the DB that the website uses `11 threads`, even when it uses less than that in reality, let's say `8`. As a result, some of the data in the `MySQL` is untrue and hard-coded return value in the `CDP` doesn't reflect the dynamic nature of mining payloads, that adjust themselves to the CPU the miner is run on.

```
await Page.addScriptToEvaluateOnNewDocument({
    source:"Object.defineProperty(navigator, 'hardwareConcurrency', {
    enumerable: true, get: function() { return 4;} } );"
});

await Runtime.evaluate({expression: "Object.defineProperty(navigator,
    'hardwareConcurrency', {enumerable: true, get: function() { return
     4;} } );"});
```

4.5.4 Database operations

The very last thing that could be improved is that running `app.js` profiler doesn't save any data to the database effectively. The reason is, the operations done from the `db.js` code, through JS `MySQL` engine, only consist of `UPDATE` SQL queries. At no point is there an `INSERT` operation that would populate the DB with the entries, leaving the defined tables with empty records eventually.

Finally the thing that could be improved is the documentation, describing how to setup the essential architecture of the program, executing CMTracker's code was by far the most complicated of the analysed detectors, because there was no template showing what databases, tables and columns have to be created to test it.

4.6 Summary

Reading all of the previous chapters, the reader may have a feeling that the opinions stated in this work towards CMTracker were the most criticizing, compared to the others detectors. This intuition would be correct, as clearly out of the three detectors CMTracker isn't the best one (in a form it is now), in terms of features

it delivered. Is it bad though? Definitely not, addressing the issues it faces now, would make it a very efficient mechanism being utilized to fight against cryptojacking. CMTracker for sure brings modern techniques, that stand out among classic black-listing approaches. I believe that if the project was further developed and published for everyone to defend themselves against malicious actors, it would succeed and prevent many cases of attempted user's resources abuse. Overall I would still recommend the detector as a base for further development and a good concept of how to approach cryptojacking related problems.

Chapter 5

SEISMIC and Retromining

5.1 Overview

At the beginning of this work it was explained what is the reason behind picking precisely this set of the projects, not the other ones. At first it seemed fair to compare all five works, as the only ones publicly available on GitHub, as mentioned in the SoK paper. Nevertheless, getting familiar with each of them made me realise that they could be divided into two categories:

- **end-user intended detectors** - ready to be used products, with name, its own source code, state-of-the-art detection techniques and property of a well-defined flow of:

  1. take URL as an input
  2. perform various operations
  3. produce an output deciding on the maliciousness of the website

  returning binary result.

- **others** - projects that dive deeply into the topic of cryptojacking, show how to recognize the threats and provide thorough study of the mining campaigns, mining websites lifetimes and so on, just as the `detectors`, however they don't fit in the same group, lacking the structure of typical standalone detector, that could be run by the user, as it requires instrumenting WASM modules and there is no code of the detector itself, or lacks novelty in detecting methods (using black-listing).

The second category is exactly where I would put SEISMIC and Retromining. Both of them introduce something new, but as they don't match the typical scheme of the detector, I decided not to analyse them thoroughly, as it happened with

Minesweeper, Outguard and CMTracker. Having said that, I wanted to briefly describe what these two projects are capable of, focusing on concepts and ideas that are unique and noteworthy for each of them.

5.2 SEISMIC - highlights

5.2.1 General description

SEISMIC [6], meaning SEcure In-lined Script Monitors for Interrupting Cryptojacks was published in 2018 in "23rd European Symposium on Research in Computer Security, ESORICS 2018" by a group of researchers from The University of Texas at Dallas. What made it not match the other projects is that it doesn't exist in a form of a detector per se. SEISMIC is a set of scripts and their instrumentation techniques, which allows to monitor crucial properties and behaviours of the observed programs. In other words in can be described not as a detector, but as a versatile method of WASM modifying, letting to collect the count of each WebAssembly operation and accessible from the JS.

The authors of the SEISMIC showed in their paper the comparison between the miners and benign WASM applications such as games. It includes outlining the semantic profiles of all of the programs. What it means is that they took hashing operations: `i32.add, i32.and, i32.shl, i32.shr and i32.xor` (the same five which were crucial from the Minesweeper's runtime perspective) and created a profile describing the ratio in which operations are for each program. It turns out that:

- **non-mining applications** - rely mostly (more than 50% of all of the counted instructions) on `add` and then on `left shift`.

- **mining payloads** - most often could be described as well-balanced, which means that each of the 5 operations constitutes a fair portion of all instructions counted e.g. around 20% each.

As one can see, this recognition method sounds fairly similar to what Minesweeper used, this time however the method of counting the instructions is different.

5.2.2 How it works?

SEISMIC modifies the WASM module to be able to count the instructions. The overview of the process looks like this:

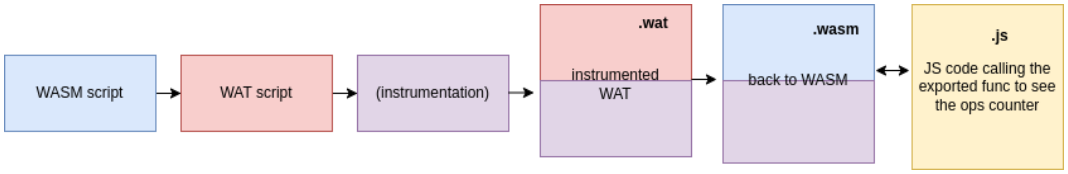


Figure 5.1: SEISMIC instrumentation flow

At first the SEISMIC takes `.wasm` file, using `wasm2wat` toolkit it transforms the script to text format, there it adds the necessary instrumentation and goes back to binary format. Later from the Javascript user can run the counters through exported functions. To fully understand this process however, let's focus on the instrumentation part, because that's what makes SEISMIC worth taking a look at.

At first let's examine the algorithm. Here is what happens in steps (assumption is made that script is already in the WAT format):

1. SEISMIC operator creates a new global variables - the counters. One is needed for each instruction. These are located at the end of the WAT file, which means after the last function and its body is defined. E.g.

```
(global (;0;) (mut i32) (i32.const 0))
...
(global (;4;) (mut i32) (i32.const 0))
```

2. Operator creates a new function to access these global variables and to make the values exportable to the JS with its use. There are three steps to follow, to create the exportable function.

   2.1 At the beginning of the WAT file (before the first function definition) the declaration of the function has to be made:

   ```
   (type (;1;) (func (result i32)))
   ```

   2.2 Later the definition, which means function body has to be added (its position depends on the compiler, sometimes names are stripped and all functions are just tagged with consecutive numbers between semicolons - `;N;` given by the compiler):

   ```
   (func $_instructionCount (type 1) (result i32) (return (get_
       global 4)))
   ```

   2.3 And finally the function has to be explicitly marked as exportable, to call it from JS:

   ```
   (export "_instructionCount" (func $_instructionCount))
   ```

3. Finally the counter has to be added the every instruction one wants to monitor. To give an example, let's assume `i32.add` is the one to be profiled. Before the instrumentation it may look like this:

```
(i32.add
  (get_local $p1)
  (i32.const 128))
```

Afterwards, it should be:

```
(i32.add (set_global 4 (i32.add (get_global 4) (i32.const 1)))
  (get_local $p1)
  (i32.const 128))
```

Which means that the inline incrementer which is the essence of SEISMIC is:

```
(set_global 4 (i32.add (get_global 4) (i32.const 1)))
```

And all it does, it is glued to the `i32.add` operation, so each time this exact line will be executed, global variable will be taken and incremented by 1.

4. Whenever user pleases they can instantiate this WASM module in the JS runtime and check the status of the counter, e.g.:

```
WebAssembly.instantiateStreaming(fetch("seismic.wasm"),
    importObject).then(obj => {
    console.log(obj.instance.exports._instructionCount())
});
```

Almost all these steps have to be performed manually as for now, hence SEISMIC couldn't be considered an outright detector. What is automated though, is the third step. SEISMIC is equipped with `inline_counter.py` that adds the counter for each instruction specified in the script. Luckily there's one counter for all functions and all occurrences of monitored operation.

One more thing worth noting is that, because of how WAT is constructed, the instruction name is in one line and its arguments in a new line or even lines, so it is very easy to add the inline incrementer by just appending the snippet at the end of the instruction name line.

Now, knowing the theory, let's take a look at how exactly the script looks like in its WebAseembly text format before and after the instrumentation (the example is based on the example created by authors by compiling the snippet):

```
int pythag(int a, int b) { return a * a + b * b; }
```

```
(module (table 0 anyfunc) (memory $0 1)
...
(export "pythag" (func $pythag))
(func $pythag (;0;) (param $0 i32) (param $1 i32) (result i32)
(i32.add (i32.mul (get_local $1) (get_local $1))
(i32.mul (get_local $0) (get_local $0)))))
```

Listing 5.1: Before the instrumentation

```
(module (table 0 anyfunc) (memory $0 1)
...
(export "pythag" (func $pythag))
(export "_getAddsLo" (func $_getAddsLo))
...
(export "_reset" (func $_reset))
(func $pythag (;0;) (param $0 i32) (param $1 i32) (result i32)
(i32.add (set_global 0 (i64.add (get_global 0) (i64.const 1)))
(i32.mul (set_global 1 (i64.add (get_global 1) (i64.const 1)))
(get_local $1) (get_local $1))
(i32.mul (set_global 1 (i64.add (get_global 1) (i64.const 1)))
(get_local $0) (get_local $0))))
(func $_getAddsLo (;1;) (result i32) (return (i32.wrap/i64 (get_global
    0))))
...
(func $_reset (; 5 ;) (set_global 0 (i64.const 0)) (set_global 1 (
   i64.const 0)))
(global (;0;) (mut i64) (i64.const 0))
(global (;1;) (mut i64) (i64.const 0)))
```

Listing 5.2: After the instrumentation

As one can see both `i32.add` and `i32.mul` are instrumented with inline counters and following the previous explanations the reader should be able to understand the code. What is visible here in the example and wasn't explained earlier, thus may need clarification, is:

- the `_reset` function, which as the name says zeros the counters. It is there if someone needed to start iterating over the instructions anew.
- the `i32.wrap/i64` that converts 64-bit numbers into 32-bit, by language specific `wrap` function, which works as modulo 2^{32} or in other words as discarding the high 32 bits and leaving low ones only. If counters exceed 32-bit values, `_getAddsHigh` function should be defined as well to return the upper part of the counter.

5.2.3 Summary

This was a brief introduction to SEISMIC and its capabilities. I believe knowing another approach towards WASM modules analysis, different from `CDP` domains and all the features of Chrome is worth mentioning, because users prefer browsers up to their preference and other ones, especially these not based on Chromium engine may have a hard time utilizing detectors dependant on features of particular browser and its remote debugging. Even tough I decided not to consider SEISMIC a detector and I still recon it was a right decision, the value added to the topic of cryptojacking and to this work is invaluable and hopefully one day techniques like the ones presented here, used to instrument WebAssembly scripts, will be utilized in the production-ready software working against cryptojacking.

5.3 Retromining - highlights

5.3.1 General description

The last work listed in the SoK paper is called Retromining [7]. It was presented during `4th Network Traffic Measurement and Analysis Conference` in June 2020 in Berlin. Just as the other projects covered earlier, this one focuses on illicit cryptomining and what makes it unique is the fact that it is written in `Go`. The reason it was declined from being acknowledged as a detector however, is because methods its scanner uses to detect mining are based mostly on black-listing approach, which is not the point of interest in this work. Nevertheless, the emphasis is put on a different matter in Retromining, which might be interesting and worth at least mentioning. It is not the new and unconventional detection methods that was the main area of research, but rather a very deep and broad search of cryptojacking sources, concentrated not only on websites' source code, but also on e.g. Tor exit nodes and mining payload injection that can happen through them.

5.3.2 Interesting points

As already mentioned, in Retromining project, the authors analysed the Tor exit nodes and their behavior in terms of cryptojacking. For those not familiar with the matter, Tor is an open-source project aiming to bring anonymity to the Internet communications. It redirects the Internet traffic through the network of voluntarily set-up relays, creating an overlay network. One of the inherent features of the Tor are exit nodes, which act as a gateways, so once the traffic hops on through some other relays (guard one, which is the first in the chain and middle one, being the second) it goes to the exit node, which forwards it to the final destination. The authors wanted to know whether these last hops tamper with the traffic, which might for instance mean injecting cryptojacking scripts. The reason this is a brilliant idea, is that due to high levels of anonymity, forming this famous `darknet`, Tor is full of malicious actors.

The way Retromining recognizes maliciousness of the exit nodes or lack of thereof is fairly straightforward:

1. Manually find and take a static website from the clearnet, which will be used as a sample.

2. Calculate the MD5 hash of its contents.

3. Using specific python library, made to interact with Tor, grab all exit nodes and fetch the sample page, routing the traffic through each of them, one by one.

4. Again calculate the hashes of the contents, for all exit nodes and compare the original MD5 with the ones just calculated.

5. If the hash is different, it means the contents have been tampered with in some way, which will be checked later, since injecting code doesn't necessarily mean mining payloads.

In the Retromining project this technique of calculating hashes to check data integrity is leveraged several times. Apart from Tor exit nodes, the same method is checked against `open proxies` and their contribution to cryptojacking. Moreover the network has been monitored with use of `Zeek` - network security monitoring tool - to find even more sources of illicit mining.

The authors implemented not only MD5 hashes - fuzzy hashes and Yara signatures are in the project as well. Furthermore the aforementioned black-listing approach focuses mostly on pattern matching based on regular expressions for both obfuscated and non-obfuscated code and global variables used to simulate namespaces in the mining code. The patterns are mainly looked for in the script names, variables names (keywords), WebSocket URIs and website URLs (public blacklists are used to make it easier).

```
1 atob\(
2 deAES\(
3 decodeURI\(
4 decodeURIComponent\(
5 escape\(
6 eval\(
7 unescape\(
```

Listing 5.3: Exemplary patterns to detect obfuscation of the payloads

5.3.3 Summary

As one can see the Retromining doesn't really fit into what has been an interest of this work. Lack of automation and state-of-the-art detection mechanisms makes the project dependent on manual job and running various independent scripts. However, a very broad research of the cryptojacking topic, focused on unprecedented areas such as Tor exit nodes in combination with open-source code and data, makes it worth mentioning and complements the work well.

Chapter 6

Conclusions

Each of these projects has its own summary section, so this chapter will be just an overall conclusion of the whole work. Having said that, I believe that the most valuable takeaway of this project is that I could test how effective the modern tools detecting mining are and experimentally prove them to be pretty promising. For years there has been a discrepancy in how advanced the malware samples were in comparison to how naive the black-listing based detectors implementations were and finally the gap is closing. Even though I have found some evasion techniques that could be used to trick the detectors, the effort needed to do so is incomparably greater, than it is to fix the lacking edge-case cryptojacking recognition in these tools. There is still a lot to work on, to be able to deliver any of these detectors to the public, as a final product, but the PoCs crafted by the authors of them prove that the foundations are solid and there is hope.

Surely one could argue that leveraging CDP in these projects is impractical and if any user would like to employ any of these detectors as a defensive tool installed on their operating system, they would need to be quite technically advanced to be able to e.g. open specific local port for remote debugging purposes. However I believe there exists an operational model in which user only has a browser extension installed, which handles communication with the external server that takes the website user visited as an input and just returns back to their browser an information, it got from the detector running in the predefined environment on that server, on the maliciousness of the website.

Finally, I believe that further research in detection against illicit mining is still important, as the headlines describing cryptojacking campaigns keep appearing. In the upcoming years there might be a shift in what malicious actors attempt to target e.g. they might focus on cloud services or Kubernetes clusters more to transform them into miners, as these technologies become more and more popular these days. But regardless of where payloads will be hidden, the threat end-user might face will still prevail and there should always be a way for end-users to defend themselves by having tools like the ones assessed in this work.

Bibliography

[1] Rajba, Pawel, and Wojciech Mazurczyk. "Limitations of Web Cryptojacking Detection: A Practical Evaluation." Proceedings of the 17th International Conference on Availability, Reliability and Security, Association for Computing Machinery, 2022, pp. 1–6. ACM Digital Library, https://doi.org/10.1145/3538969.3544466.

[2] Tekiner, Ege, et al. "SoK: Cryptojacking Malware." 2021 IEEE European Symposium on Security and Privacy (EuroS&P), 2021, pp. 120–39. IEEE Xplore, https://doi.org/10.1109/EuroSP51992.2021.00019.

[3] Konoth, Radhesh Krishnan, et al. "MineSweeper: An In-Depth Look into Drive-by Cryptocurrency Mining and Its Defense." Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, 2018, pp. 1714–30. ACM Digital Library, https://doi.org/10.1145/3243734.3243858.

[4] Kharraz, Amin, et al. "Outguard: Detecting In-Browser Covert Cryptocurrency Mining in the Wild." The World Wide Web Conference, Association for Computing Machinery, 2019, pp. 840–52. ACM Digital Library, https://doi.org/10.1145/3308558.3313665.

[5] Hong, Geng, et al. "How You Get Shot in the Back: A Systematical Study about Cryptojacking in the Real World." Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, 2018, pp. 1701–13. ACM Digital Library, https://doi.org/10.1145/3243734.3243840.

[6] Wang, Wenhao, et al. "SEISMIC: SEcure In-Lined Script Monitors for Interrupting Cryptojacks." Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II, Springer-Verlag, 2018, pp. 122–42. ACM Digital Library, https://doi.org/10.1007/978-3-319-98989-1_7.

[7] Holz, Ralph, et al. A Retrospective Analysis of User Exposure to (Illicit) Cryptocurrency Mining on the Web. arXiv:2004.13239, arXiv, 25 June 2020. arXiv.org, https://doi.org/10.48550/arXiv.2004.13239.

[8] 88446 – Web Inspector: [Chromium] Profiler - Differentiate between Native Code (Program) and Idle Time. https://bugs.webkit.org/show_bug.cgi?id=88446. Accessed 3 June 2024.

[9] B, Evan. "Answer to 'What Is Sample and Feature.'" Data Science Stack Exchange, 23 July 2019, https://datascience.stackexchange.com/a/56266.

[10] "BLAKE3/README.Md at Master · BLAKE3-Team/BLAKE3." GitHub, https://github.com/BLAKE3-team/BLAKE3/blob/master/README.md. Accessed 3 June 2024.

[11] Building Projects — Emscripten 3.1.61-Git (Dev) Documentation. https://emscripten.org/docs/compiling/Building-Projects.html?highlight=standalone_wasm. Accessed 3 June 2024.

[12] Call Stack - MDN Web Docs Glossary: Definitions of Web-Related Terms — MDN. 7 May 2024, https://developer.mozilla.org/en-US/docs/Glossary/Call_stack.

[13] Calvet, Joan, et al. "Aligot: Cryptographic Function Identification in Obfuscated Binary Programs." Proceedings of the 2012 ACM Conference on Computer and Communications Security, ACM, 2012, pp. 169–82. DOI.org (Crossref), https://doi.org/10.1145/2382196.2382217.

[14] Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/tot/Profiler/. Accessed 3 June 2024.

[15] —. https://chromedevtools.github.io/devtools-protocol/. Accessed 3 June 2024.

[16] —. https://chromedevtools.github.io/devtools-protocol/tot/Target/. Accessed 3 June 2024.

[17] Chromium Docs - Checking out and Building Chromium on Linux. https://chromium.googlesource.com/chromium/src/+/main/docs/linux/build_instructions.md. Accessed 3 June 2024.

[18] Chromium Docs - Threading and Tasks in Chrome. https://chromium.googlesource.com/chromium/src/+/main/docs/threading_and_tasks.md. Accessed 3 June 2024.

[19] Chromium Source Code. https://source.chromium.org/chromium/chromium/src/+/main:v8/src/profiler/profile-generator.cc;l=136;drc=a0a8807df90f23a99f7242e9456504a932c6a4c1. Accessed 3 June 2024.

[20] "Cmtracker/README.Md at Master · Deluser8/Cmtracker." GitHub, https://github.com/deluser8/cmtracker/blob/master/README.md. Accessed 3 June 2024.

[21] "Code/README.Md at Master · Retrocryptomining/Code." GitHub, https://github.com/retrocryptomining/code/blob/master/README.md. Accessed 3 June 2024.

[22] Craver, Nick. "Answer to 'What Is "(Program)" in Chrome Debugger's Profiler?'" Stack Overflow, 3 Oct. 2010, https://stackoverflow.com/a/3848006.

[23] "Cryptocurrency." Wikipedia, 24 May 2024. Wikipedia, https://en.wikipedia.org/w/index.php?title=Cryptocurrency&oldid=1225511944.

[24] "Cryptominer." Pastebin.Com, https://pastebin.com/RgRi6Fcn. Accessed 3 June 2024.

[25] "CryptonightR/README.Md at Master · SChernykh/CryptonightR." GitHub, https://github.com/SChernykh/CryptonightR/blob/master/README.md. Accessed 3 June 2024.

[26] "deepMiner/CryptoNight.Txt at Master · Deepwn/deepMiner." GitHub, https://github.com/deepwn/deepMiner/blob/master/CryptoNight.txt. Accessed 3 June 2024.

[27] "Emscripten Doesn't Truly Produce Standalone/WASI Output When Requested · Issue #20484 · Emscripten-Core/Emscripten." GitHub, https://github.com/emscripten-core/emscripten/issues/20484. Accessed 3 June 2024.

[28] "Emscripten/Src/Settings.Js at Main · Emscripten-Core/Emscripten." GitHub, https://github.com/emscripten-core/emscripten/blob/main/src/settings.js. Accessed 3 June 2024.

[29] erosman. "Answer to 'What Is the Purpose of a Background Page?'" Stack Overflow, 27 Feb. 2017, https://stackoverflow.com/a/42478459.

[30] Gallant, C. Gerard. "WebAssembly — Web Workers." Medium, 14 Jan. 2018, https://medium.com/@c.gerard.gallant/webassembly-web-workers-f2ba637c3e4a.

[31] "GitHub - WebAssembly/WASI at 4712d490fd7662f689af6faa5d718e042f014931." GitHub, https://github.com/WebAssembly/WASI. Accessed 3 June 2024.

[32] Gröbert, Felix, et al. "Automated Identification of Cryptographic Primitives in Binary Programs." Recent Advances in Intrusion Detection, edited by Robin Sommer et al., vol. 6961, Springer Berlin Heidelberg, 2011, pp. 41–60. DOI.org (Crossref), https://doi.org/10.1007/978-3-642-23644-0_3.

[33] Hash Function Grøstl – SHA-3 Candidate. https://www.groestl.info/index.html. Accessed 3 June 2024.

[34] Important Abstractions and Data Structures. https://www.chromium.org/developers/coding-style/important-abstractions-and-data-structures/. Accessed 3 June 2024.

[35] Jack. "HTML5 Websocket within Webworker." Stack Overflow, 1 Aug. 2013, https://stackoverflow.com/q/17998011.

[36] jmrk. "Answer to 'Why Do Neither V8 nor Spidermonkey Seem to Unroll Static Loops?'" Stack Overflow, 6 Feb. 2021, https://stackoverflow.com/a/66082249.

[37] "Js-Worker-Interface/README.Md at Master · Burdiuz/Js-Worker-Interface." GitHub, https://github.com/burdiuz/js-worker-interface/blob/master/README.md. Accessed 3 June 2024.

[38] Kumar, Ritesh. "How to Use WebAssembly Modules in a Web Worker." PSPDFKit, https://pspdfkit.com/blog/2020/webassembly-in-a-web-worker/. Accessed 3 June 2024.

[39] "Libcryptonight/Cryptonight.c at Master · Har1s1m/Libcryptonight." GitHub, https://github.com/har1s1m/libcryptonight/blob/master/cryptonight.c. Accessed 3 June 2024.

[40] lightalchemist. "Answer to 'ConvergenceWarning: Liblinear Failed to Converge, Increase the Number of Iterations.'" Stack Overflow, 16 Oct. 2018, https://stackoverflow.com/a/52828900.

[41] List of Chromium Command Line Switches őPeter Beverloo. https://peter.sh/experiments/chromium-command-line-switches/. Accessed 3 June 2024.

[42] Loading and Running WebAssembly Code - WebAssembly — MDN. 20 Nov. 2023, https://developer.mozilla.org/en-US/docs/WebAssembly/Loading_and_running.

[43] Lucks, S., and J. Callas. The Skein Hash Function Family. 2009. Semantic Scholar, https://www.semanticscholar.org/paper/The-Skein-Hash-Function-Family-Lucks-Callas/64e648c42faa31c2bcbe2c97b452faea842fe9a6.

[44] Milner, James. "Using WebAssembly with Web Workers." SitePen, 22 July 2019, https://www.sitepen.com/blog/using-webassembly-with-web-workers.

[45] "Minesweeper/README.Md at Master · Vusec/Minesweeper." GitHub, https://github.com/vusec/minesweeper/blob/master/README.md. Accessed 3 June 2024.

[46] "Monero/README.Md at Master · Monero-Project/Monero." GitHub, https://github.com/monero-project/monero/blob/master/README.md. Accessed 3 June 2024.

[47] "Monero/Src/Crypto/Slow-Hash.c at Ac02af92867590ca80b2779a7bbeafa99ff94dcb · Monero-Project/Monero." GitHub, https://github.com/monero-project/monero/blob/ac02af92867590ca80b2779a7bbeafa99ff94dcb/src/crypto/slow-hash.c#L893. Accessed 3 June 2024.

[48] "Outguard/README.Md at Master · Teamnsrg/Outguard." GitHub, https://github.com/teamnsrg/outguard/blob/master/README.md. Accessed 3 June 2024.

[49] Outside the Web: Standalone WebAssembly Binaries Using Emscripten · V8. https://v8.dev/blog/emscripten-standalone-wasm. Accessed 3 June 2024.

[50] Paper. https://docs-archive.freebsd.org/44doc/psd/18.gprof/paper.html. Accessed 3 June 2024.

[51] pkozlowski.opensource. "Answer to 'Javascript Count Function Calls in Chrome Profiler.'" Stack Overflow, 29 June 2017, https://stackoverflow.com/a/44832574.

[52] "RandomForestClassifier." Scikit-Learn, https://scikit-learn/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. Accessed 3 June 2024.

[53] "RandomX/README.Md at Master · Tevador/RandomX." GitHub, https://github.com/tevador/RandomX/blob/master/README.md. Accessed 3 June 2024.

[54] Saberhagen, Nicolas van. CryptoNote v 2.0. 2013. Semantic Scholar, https://www.semanticscholar.org/paper/CryptoNote-v-2.0-Saberhagen/5bafdd891c1459ddfd22d71412d5365de723fb23.

[55] "SEISMIC/README.Md at Master · Wenhao1006/SEISMIC." GitHub, https://github.com/wenhao1006/SEISMIC/blob/master/README.md. Accessed 3 June 2024.

[56] "Server-Sent Events: A WebSockets Alternative Ready for Another Look." Ably Realtime, https://ably.com/topic/server-sent-events. Accessed 3 June 2024.

[57] Service Worker API - Web APIs — MDN. 18 May 2024, https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

[58] Soroker, Tali. "How JS Works Behind The Scenes — The Engine." Coralogix, 23 Feb. 2021, https://coralogix.com/blog/how-js-works-behind-the-scenes-the-engine/.

[59] —. "Web Assembly Deep Dive - How It Works, And Is It The Future?" Coralogix, 13 Apr. 2021, https://coralogix.com/blog/web-assembly-deep-dive-how-it-works-and-is-it-the-future/.

[60] Text Format — WebAssembly 2.0 (Draft 2024-04-28). https://webassembly.github.io/spec/core/text/index.html. Accessed 3 June 2024.

[61] The Event Loop - JavaScript — MDN. 26 Feb. 2024, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop.

[62] The WebSocket API (WebSockets) - Web APIs — MDN. 15 Mar. 2024, https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

[63] Understanding WebAssembly Text Format - WebAssembly — MDN. 25 Apr. 2024, https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format.

[64] Using the WebAssembly JavaScript API - WebAssembly — MDN. 20 Nov. 2023, https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API.

[65] Using Web Workers - Web APIs — MDN. 9 Nov. 2023, https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.

[66] Varlioglu, Said, et al. "Is Cryptojacking Dead After Coinhive Shutdown?" 2020 3rd International Conference on Information and Computer Technologies (ICICT), IEEE, 2020, pp. 385–89. DOI.org (Crossref), https://doi.org/10.1109/ICICT50521.2020.00068.

[67] Web Workers API - Web APIs — MDN. 14 Dec. 2023, https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API.

[68] WebAssembly — MDN. 16 Apr. 2024, https://developer.mozilla.org/en-US/docs/WebAssembly.

[69] WebAssembly Opcode Table. https://pengowray.github.io/wasm-ops/. Accessed 3 June 2024.

[70] WebAssembly Specification — WebAssembly 2.0 (Draft 2024-04-28). https://webassembly.github.io/spec/core/. Accessed 3 June 2024.

[71] "WebAssembly Standalone." GitHub, https://github.com/emscripten-core/emscripten/wiki/WebAssembly-Standalone. Accessed 3 June 2024.

[72] WebAssembly.instantiateStreaming() - WebAssembly — MDN. 20 Nov. 2023, https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/instantiateStreaming_static.

[73] WebAssembly.Module.Exports() - WebAssembly — MDN. 20 Nov. 2023, https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/Module/exports_static.

[74] WebSocket - Web APIs — MDN. 6 Mar. 2024, https://developer.mozilla.org/en-US/docs/Web/API/WebSocket.

[75] "WebSocket API and Protocol Explained: How They Work, Are Used and More." Ably Realtime, https://ably.com/topic/websockets. Accessed 3 June 2024.

[76] "Xmr-Wasm/README.Md at Master · Jtgrassie/Xmr-Wasm." GitHub, https://github.com/jtgrassie/xmr-wasm/blob/master/README.md. Accessed 3 June 2024.

[77] Yiu, Tony. "Understanding Random Forest." Medium, 29 Sept. 2021, https://towardsdatascience.com/understanding-random-forest-58381e0602d2.

[78] https://source.chromium.org/chromium/chromium/src/+/main:third_party/devtools-frontend/src/test/e2e/resources/performance/wasm/profiling.html. Accessed 3 June 2024.

[79] https://source.chromium.org/chromium/chromium/src/+/main:ppapi/cpp/message_loop.h. Accessed 3 June 2024.

[80] https://source.chromium.org/chromium/chromium/src/+/main:v8/tools/wasm/update-wasm-fuzzers.sh?q=dump-wasm-module. Accessed 3 June 2024.

[81] GX.Games Run in Opera Browser. https://gx.games/. Accessed 5 June 2024.