MINOS*: A Lightweight Real-Time Cryptojacking Detection System

Faraz Naseem, Ahmet Aris, Leonardo Babun, Ege Tekiner, and A. Selcuk Uluagac
Cyber-Physical Systems Security Lab
Department of Electrical & Computer Engineering, Florida International University, Miami, FL, USA
{fnase001, aaris, lbabu002, egetekiner, suluagac}@fiu.edu

***Abstract*—Emerging WebAssembly(Wasm)-based cryptojacking malware covertly uses the computational resources of users without their consent or knowledge. Indeed, most victims of this malware are unaware of such unauthorized use of their computing power due to techniques employed by cryptojacking malware authors such as CPU throttling and obfuscation. A number of dynamic analysis-based detection mechanisms exist that aim to circumvent such techniques. However, since these mechanisms use dynamic features, the collection of such features, as well as the actual detection of the malware, require that the cryptojacking malware run for a certain amount of time, effectively mining for that period, and therefore causing significant overhead. To solve these limitations, in this paper, we propose MINOS, a novel, extremely lightweight cryptojacking detection system that uses deep learning techniques to accurately detect the presence of unwarranted Wasm-based mining activity in real-time. MINOS uses an image-based classification technique to distinguish between benign webpages and those using Wasm to implement unauthorized mining. Specifically, the classifier implements a convolutional neural network (CNN) model trained with a comprehensive dataset of current malicious and benign Wasm binaries. MINOS achieves exceptional accuracy with a low TNR and FPR. Moreover, our extensive performance analysis of MINOS shows that the proposed detection technique can detect mining activity instantaneously from the most current in-the-wild cryptojacking malware with an accuracy of 98.97%, in an average of 25.9 milliseconds while using a maximum of 4% of the CPU and 6.5% of RAM, proving that MINOS is highly effective while lightweight, fast, and computationally inexpensive.**

I. INTRODUCTION

In recent years, a new type of fileless malware that exploits the computational resources of end-users via browsers, has become increasingly common [1]. This new strain of malware, known as *Cryptojacking* (a.k.a., drive-by-mining) malware, performs unauthorized and covert cryptocurrency mining operations in browsers without the end-users' knowledge [2]. Both the tremendous rise in the monetary value of cryptocurrencies, and the profitability of browser-based mining, have been major driving forces behind the use of cryptojacking. As such, there have been a number of major cryptojacking incidents that have affected various popular services and websites in the past. For instance, some of these cryptojacking incidents have affected popular streaming services, and web applications like YouTube [3], Openload, Streamango, Rapidvideo, OnlineVideoConverter [4], Los Angeles Times [5], and some other organizational websites (e.g., US and UK government websites) [6]. A prime example that made the news recently constitutes the episode of the Starbucks' WiFi network in Buenos Aires [7], which was injecting cryptojacking malware through all its outgoing connections due to 1.4 million compromised MikroTik routers [8]. Also, several researchers [2], [9], [10], [11], [12] confirmed that cryptojacking is prevalent in the wild based on their analyses' of websites in the Alexa and Zmap top 1 million lists.

The birth of cryptojacking can be attributed to a number of emerging technologies such as WebAssembly (Wasm), WebWorkers, and WebSockets. In general, these technologies have served to facilitate high-performing, scalable web applications running on browsers. In the context of cryptocurrency, Monero came forward with the promise of untraceable transactions, which caught the attention of malicious entities in the dark web [13]. The Coinhive mining service provided WebAssembly-based Monero-mining scripts to website owners as an alternative source of income/profit [9], [2], [13]. Thus, cryptojacking was born, a new cryptocurrency mining malware running covertly on end-user browsers that relies on the latest web technologies and easily reaches its victims via websites without requiring any software installation. The misuse of Coinhive scripts by malicious entities without the consent of end-users facilitated the shut down of Coinhive in March 2019 [14]. Although Coinhive is no longer maintained or operational, numerous studies [15], [14] indicate that cryptojacking malware is still in use in the wild. The findings of our study add further support to this statement.

There exist several detection and protection mechanisms against cryptojacking malware. In this respect, several browser extensions (e.g., Nocoin, minerBlock, etc.) exist that employ the use of blacklists to stop malicious cryptocurrency mining operations. However, blacklists are not a suitable remedy since malicious entities that use cryptojacking malware change their domains' names frequently [9], [16]. Antivirus software, on the other hand, performs signature matching by looking for the presence of specific keywords (e.g., miner, coin, Coinhive, etc.), and function names that are indicative of a cryptojacking script on the webpage. This detection technique was also reported to be easily bypassed [17]. In addition to these protection mechanisms, various researchers proposed a

* Minos is the beast in Dante's Divine Comedy that acts as a judge in underworld and decides which layer of the hell the sinner goes to.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24444
www.ndss-symposium.org

number of highly accurate detection systems for cryptojacking malware. Most of these techniques are based on the analysis of dynamic features. These include fixed thresholds [16], semantic instruction-count-based signature matching [17], CPU, memory and network traffic features [18], [19], [20], runtime, network, mining-related and browser-based features [12], block-level profiling and dynamic instrumentation [15], [21], [22], hardware performance counters [23], and instruction-count-based analysis and memory events [10]. Although such existing detection systems are promising and report high detection accuracy, there are several issues to consider. Firstly, the studies using dynamic analysis techniques have high runtime overheads. They also suffer from measurement inaccuracies due to noise resulting from external processes/applications. In addition, they require administrator privileges to monitor CPU and memory events. Since benign web applications also use the aforementioned web technologies (i.e., WebAssembly, WebWorkers, WebSockets), the existing dynamic techniques can have high false positives. Moreover, cryptojacking authors employ dynamically generated domain names, proxies, and encrypted communications, which render detection systems that rely on network-data analysis inefficient. Finally, dynamic analysis and instrumentation-based detection systems running continuously in the background can negatively affect the quality of the web surfing experience of end-users.

In this paper, we propose MINOS, a lightweight, fast and efficient defense solution against cryptojacking malware. MINOS is a novel cryptojacking detection system that employs the use of gray-scale image representations of Wasm-binaries in web browsers. Specifically, MINOS converts a suspected Wasm binary to a gray-scale image, and utilizes a Convolutional Neural Network (CNN)-based Wasm classifier to classify the image as either malicious (i.e., cryptojacking) or benign. Unlike dynamic analysis-based techniques, MINOS does not require continuous monitoring of CPU, memory, and network events or counting running instructions. Hence, its runtime overhead is significantly lower, and it does not affect the quality of the web surfing experience of end-users. In addition, evasion attempts employed by cryptojacking malware authors based on CPU throttling, dynamic domain names usage, and encrypted communications, would be unsuccessful against MINOS.

We designed and implemented the MINOS lightweight cryptojacking detection system as an end-to-end framework. We trained and evaluated MINOS using one of the most comprehensive collected datasets on cryptojacking malware samples in the wild. The framework is further evaluated by testing it against a dataset of real webpages in the wild that use Wasm. The results of this evaluation conclude that the classifier achieves extremely high accuracy, and is able to accurately and instantaneously identify the presence of cryptojacking malware while consuming minimal computational resources.

Contributions: The contributions of this paper are listed as follows:

- A novel cryptojacking detection mechanism that implements a Wasm binary classifier in a lightweight and computationally inexpensive end-to-end framework with instantaneous detection capabilities - MINOS.
- A novel Wasm binary classification technique that utilizes gray-scale image representations of the binaries to train a convolutional neural network.
- Our extensive evaluation demonstrates that MINOS is capable of detecting Wasm-based cryptojacking with 98.97% accuracy on a compiled dataset of real, in-the-wild samples, with minimal overhead. Specifically, the proposed detection mechanism successfully detected all cryptojacking malware in our datasets within 25.9 milliseconds on average, with only 6.5% and 4% maximum utilization of RAM and CPU, respectively.

Organization: The rest of the paper is organized as follows: Section II gives the background information. Section III analyzes the existing detection mechanisms, and outlines why there is a need for a new cryptojacking detection system. Section IV defines the threat model. Section V proposes an overview of our novel MINOS framework. Section VI describes how we implemented MINOS. Performance evaluation of MINOS is performed in Section VII. Section VIII provides discussions and considerations on various aspects. Related work is summarized in Section IX, and Section X finalizes the paper with the conclusion.

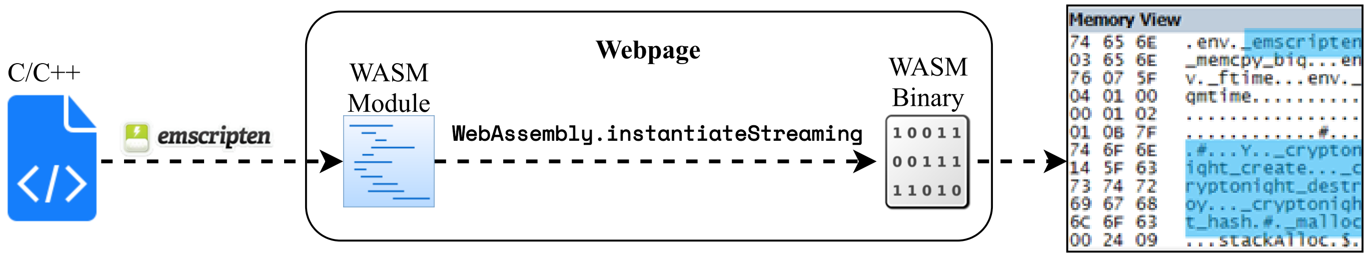
II. BACKGROUND

A. *Cryptocurrency Mining and Cryptojacking*

Cryptocurrency mining started with CPU-bound PoW schemes that mine Bitcoin or Ether currencies [9]. When one of the miners computes a block successfully, a new block for the blockchain is generated by the miner, and in return, the miner receives a certain amount of cryptocurrency as a reward. Since the revenue that could be obtained was directly proportional to the amount of processing power, Application-Specific Integrated Circuits (ASICs), Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) became the de facto platforms for CPU-bound PoW schemes instead of ordinary desktop computers [9], [2]. In order to remedy this issue, new cryptocurrencies (i.e., Monero, Bytecoin, etc.) that utilize memory-bound PoW schemes emerged. These schemes were based on hash puzzles that required voluminous interactions with the memory rather than CPU power. Hence, ordinary computers started to be suitable mining environments for such cryptocurrencies.

Cryptocurrency mining in browsers first started with the Coinhive miner, which promised an alternative income opportunity to website owners in 2017 [2], [9], [13]. In Coinhive mining, website owners were placing cryptocurrency mining scripts on their websites that would trigger the mining process within the visitors' browsers. There were also legitimate websites that would receive the consent of users to mine cryptocurrency. However, this extra income opportunity caught the attention of malicious entities. These individuals began to covertly mine cryptocurrency without the explicit consent of users. This phenomena is known as *cryptojacking*, or *drive-by mining / coinjacking* [2]. In this new type of malware, attackers misuse the processing power of victims, and derive revenue from it.

A malicious cryptocurrency mining script can be injected into websites in a number of ways [1], [2]: 1) Website



```
74 65 6E    .env._emscripten
03 65 6E    _memcpy_big...en
76 07 5F    v._ftime...env._
04 01 00    gmtime..........
00 01 02    ................
01 0B 7F    ............#...
74 6F 6E    .#...Y.._crypton
14 5F 63    ight_create..._c
73 74 72    ryptonight_destr
69 67 68    oy..._cryptonigh
6C 6F 63    t_hash.#._malloc
00 24 09    ...stackAlloc.$.
```

Fig. 1: We illustrate the process of implementing malicious Wasm modules that mine cryptocurrecy in webpages. The right-most image shows the resulting Wasm module through a binary dissassembler. The highlighted portions of text (i.e. strings found in the binary) confirm that the module was compiled using Emscripten and that the binary is indeed executing cryptocurreny mining functions.

owners can embed such scripts in their websites and activate them without taking consent of the visitors. 2) Third-party services can inject such scripts without informing neither website owners nor end-users. 3) Malicious browser extensions can run cryptocurrency miners in the background silently. 4) Attackers can breach servers, browser extensions, third-party services, and inject cryptojacking malware. 5) Vulnerable network devices such as routers, access points, etc. can be exploited to mine cryptocurrency through web traffic.

A browser-based cryptocurrency miner typically consists of a JavaScript code snippet that has the identification number of the script owner. It also has the corresponding code that configures the mining process, communicates with the cryptocurrency service provider, and starts the mining operation. The identification number of the script owner discriminates the malicious entity that owns the script amid other entities from the cryptocurrency service provider's point of view. In this way, service providers can monitor and measure the total hashing power provided by script owners. The service provider communicates with miners through high-performance communication primitives, like WebSockets. In order to increase profit, cryptocurrency miners use WebWorkers to run the mining process in parallel via multiple threads. Further, to solve hash puzzles with high efficiency, they utilize miner implementations in WebAssembly instead of JavaScript [9].

B. *WebAssembly and Cryptojacking Malware*

WebAssembly (Wasm) is a low-level binary instruction format that promises to run code near native speeds in a stack-based virtual machine within the browsers [24]. It is currently supported by four major, widely-used browsers, including Google Chrome, Mozilla Firefox, Microsoft Edge, and Safari. Its use of binary encoding results in efficiency in size and load-time, and execution speeds that are comparative to native machine code [24]. Other principle features include it being easy to decode, hardware and platform-independent, and compact [25].

Rather than replacing JavaScript (JS), Wasm is meant to supplement and run in parallel with JS. The language is designed to be used as a target for compilation of numerous high-level languages such as C, C++, and Rust. Webpages, written in JS, will also instantiate Wasm modules that are then compiled in a sandbox environment. Using the same Web APIs available to JS, Wasm modules have the ability to call in and out of the JS context, and access browser functionality. The most widely used toolchain for compiling modules written in C/C++ into Wasm is the open source LLVM compiler, Emscripten. The near native speed of Wasm is achieved due to the fact that the modules have already been optimized during compilation, and memory management is done without the use of a garbage collector.

Advantageous features of WebAssembly make it suitable to implement and execute browser-based cryptocurrency mining functions that require substantial computational power such as `cryptonight_hash`. As such, a vast majority of browser-based cryptojacking malware implements Wasm to execute the cryptocurrency mining payload. This is apparent as Konoth et al. [10] reported that 100% of the 1,735 cryptocurrency mining websites they identified in their study utilized Wasm. In parallel to this study, Musch et al. [11] analyzed the prevalence of WebAssembly in wild. They inspected the Alexa top 1 million websites, and realized that 0.16% of the websites employ WebAssembly. Their analysis revealed that more than half of the websites that employ WebAssembly, are using it for malicious purposes. They highlighted that cryptojacking is the major application among malicious use-cases.

Crytojacking malware authors write code in C/C++ that performs mining functions including:

`cryptonight_create`,
`cryptonight_destroy`, and
`cryptonight_hash`.

They then compile this to a Wasm module using the Emscripten toolchain. This Wasm module is then accessed through the JavaScript function,

`WebAssembly.instantiateStreaming`.

The `fetch()` method of the Fetch JavaScript API is used as the function's first argument. This method fetches, compiles, and instantiates the module, enabling access to the raw byte code. During the compilation phase, the Wasm binary has already undergone optimization and can hook directly into the backend where machine code is generated and executed. This code performs mathematical operations that facilitate solving convoluted hash puzzles i.e., mining cryptocurrency. A visual depiction of this procedure can be seen in Figure 1.

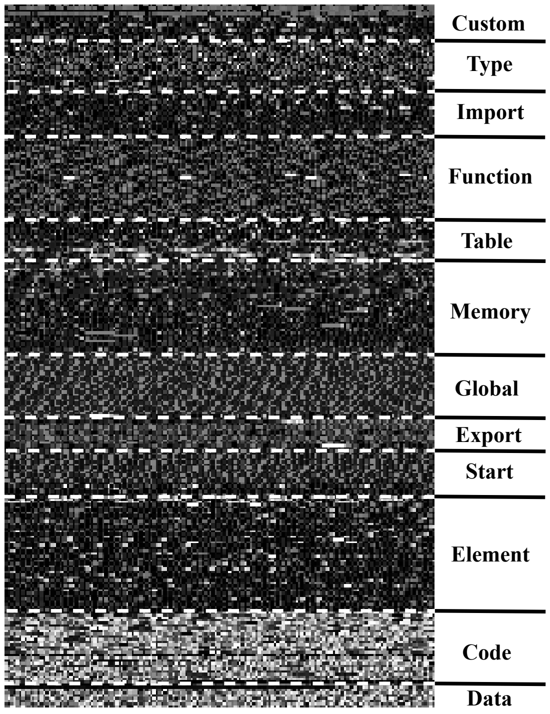


Fig. 2: An image depicting a Wasm binary with each section of the image being labelled corresponding to each of the 12 respective sections described in Section II-C.

C. Structure of WebAssembly Modules

A Wasm module comprises 12 distinct sections, each with its own section ID ranging from 0 to 11. The following list describes each module and includes each section's corresponding ID number (Figure 2).

0) **Custom section:** This section is either used for debugging purposes or by third-party extensions for custom purposes. It contains a name that identifies the section as the custom section and a custom sequence of bytes for use by third-party extensions.
1) **Type section:** The type section decodes or defines a vector of function types. This component contains every function type utilized in the module.
2) **Import section:** This section decodes a vector of a set of imports that are required in order to confirm that the module is valid during instantiation. Each import definition consists of a module name and a name for an element within that specific module.
3) **Function section:** The function section consists of a vector of type indices representative of the *type* parameter of the functions defined in the module. Each function definition consists of 3 parameters: type, locals, and body. The locals and body parameters are encoded in the code section.
4) **Table section:** This section contains a vector of tables that are defined by their table type. A table consists of a vector of values of a specific element type.
5) **Memory section:** This component decodes into a vector of linear memories described by their memory type, consisting of raw, uninterpreted bytes.
6) **Global section:** The global section contains a vector of global variables used in the module. Each global variable definition consists of a single value describing its type.
7) **Export section:** This section consists of a vector of a set of exports that the host environment can access after module instantiation.
8) **Start section:** The start component defines the index of an optional start function that is invoked during module instantiation after tables and memories are initialized. This function is used to initialize the state of the module.
9) **Element section:** The element section is comprised of a vector of element segments that initialize a subrange of a table based on a specifically defined offset.
10) **Code section:** The code section contains a vector of code entries that consist of pairs of expressions and value types. The four value types that Wasm variables can take include 32 and 64-bit integers and 32 and 64-bit floating-point data. In the module's code, these types are represented by the terms $i32$ and $i64$, and $f32$ and $f64$ respectively. Each code entry is comprised of the size of the function code in bytes, and the function code. The function code, in turn, consists of local variables and the body of the function as expressions. These correspond to the local and body parameters of the sections of the function mentioned previously.
11) **Data section:** This section is comprised of a vector of data segments that initialize a range of memory based on a specifically defined offset.

III. Need for a New Cryptojacking Detection System

The current literature has a plethora of works on the detection of malicious cryptocurrency miners. We can group the works into three categories based on the type of detection system: 1) browser-based detection (targeting cryptojacking), 2) host-based detection (targeting stand-alone miners that run as a malicious software on hosts), and 3) network-based detection (targeting any type of miners). In this section, we will briefly examine the detection systems for each category respectively. In addition, we will outline the need for a new cryptojacking detection system.

A. Malicious Cryptocurreny Mining Detection Systems

Browser-based Mining (Cryptojacking) Detection: Several researchers proposed highly accurate detection systems against cryptojacking malware. Hong et al. [16] proposed CMTracker that uses the cumulative time spent on hashing operations and stack characteristics of threads. Wang et al. [17] proposed SEISMIC, as a semantic signature-matching-based detection system that instruments the Wasm modules to count specific instructions in runtime. Rodriguez and Posegga [18] proposed RAPID, a Support Vector Machine (SVM) based classifier that uses CPU and memory events, and network traffic features. Kharraz et al. [12] proposed OUTGUARD, that builds an SVM classifier using features of runtime, network, mining, and browser events. Bian et al. [15] proposed MineThrottle, which uses a block-level profiler and dynamic instrumentation of the Wasm code that is pointed by the profiler at compile time. Kelton et al. [19] proposed CoinSpy which utilizes computation, memory, and network features. Conti et al. [23] used Hardware Performance Counters (HPC) data to detect cryptojacking. Konoth et al. [10] proposed MineSweeper that firstly analyzes the Wasm modules and counts number of specific instructions. It tries to find out how similar the analyzed module is to cryptojacking malware Wasms. In addition, it monitors cache events during runtime.

Host-based Stand-alone Mining Detection: In terms of malicious cryptocurrency miners not targeting browsers but directly hosts, Darabian et al. [26] proposed a detection system that uses opcode sequences and system calls of Windows Portable Executable (PE) files. Berecz and Czibula [27] employed both static features (i.e., entropy, and header, section, and function information) and API calls. Vladimír and Žádník [28] presented two techniques, in which the first technique uses a decision tree on the flow features, and the second technique acts like a miner client probing the miner server.

Network-based Detection: As a network-level detection mechanism, Neto et al. [29] proposed MineCap, a network-flow-based detection and blocking mechanism to protect the network of devices controlled by the SDN controller. MineCap relies on Apache Spark Streaming library and incremental ML model to detect the cryptocurrency mining flows.

B. Challenges and Need for a New Detection System

Cryptojacking detection is challenging. Benign web applications frequently use the technologies that are also used by cryptojacking malware (e.g., WebAssembly, WebWorkers, WebSockets). Similar to cryptojacking scripts, benign web applications can consume high CPU and memory resources. All of these characteristics can affect the accuracy and false positive rates of detection systems. In addition, end-users browse the web through a variety of browsers using various operating systems that require solutions to be platform-independent. Moreover, end-users expect pages to load quickly and run flawlessly, which force detection solutions to be lightweight and efficient. In addition, attackers can throttle the resource consumption of their malware, change function names and strings, use proxies, dynamically generated domain names, and encrypted communications that make their detection increasingly challenging.

However, considering the challenges and approaches proposed in prior work, cryptojacking detection can still be improved drastically. Existing detection approaches, albeit useful, have several drawbacks. Firstly, cryptojacking is moving from JavaScript to WebAssembly for various reasons (e.g., performance, hardware support) [12]. However, only a small portion of prior studies [17], [10], [12] take this change into account. Secondly, cryptojacking detection systems [17], [10], [12], [18], [15], [19] relying on dynamic analysis features can suffer from high computational overhead, reduced measurement accuracies due to noise caused by other processes, and false positives resulted from benign websites using the same technologies. For this reason, practical applications of such schemes may cause quality-of-experience issues for end-users. Moreover, attackers can easily find ways to circumvent existing detection systems. To be more specific, fixed threshold values used by CMTracker [16] can be evaded easily. The techniques of Vladimír and Žádník [28] are not effective against private mining pools or mining pools hidden behind proxies. Cryptojacking scripts can use encrypted communication, dynamic domain names, alternative communication primitives (e.g., XMLHttpRequest), and proxies to bypass detections systems that use network-based features [9]. In addition, some detection systems have drawbacks that may limit their acceptance and usage by end-users. For instance, one of the features used by OUTGUARD [12] (i.e., MessageLoop event) is browser-dependent. MineCap [29] can be utilized only by operators which employ SDN in their networks. Therefore, individual users cannot use it. Instrumentation code added by SEISMIC [17] may severely affect the performance of legitimate web applications that use Wasm, which may degrade the quality of experience of end-users.

Based on all of the mentioned reasons, a new cryptojacking detection system is needed, which is lightweight, isolated from external noise, robust against adversarial attempts, and which considers the changing landscape of cryptojacking malware. To address these issues, we propose MINOS, a new cryptojacking detection system that employs visualization of cryptojacking Wasms.

IV. THREAT MODEL

In this study, we consider an attacker model that injects cryptojacking script in a number of ways:

- The attacker can inject a cryptojacking script to their website and activate it without taking consent of visitors,
- The attacker can embed a cryptojacking script to their services that are used by various websites as a third-party service, and they may not inform the website owners,
- The attacker can compromise access points, routers, and any other intermediate devices, and configure the device to inject their cryptojacking scripts to all web traffic.

In addition to the methods used to inject cryptojacking scripts, the attacker can use dynamically generated domain names, proxies, alternative communication protocols (e.g., XMLHttpRequest) and encrypted communications to obfuscate the network-level behavior of his/her script. Moreover, the attacker can configure his/her script to throttle the CPU usage of cryptojacking. Regarding obfuscating the cryptojacking code, our attacker model considers the obfuscation of (1) strings, and (2) function names only since these are the most common applications of obfuscation currently applied to browser-based cryptocurrency miners in the wild, as reported by Wang et al. [17].

The attacker is assumed to have the major implementation of the cryptojacking malware in WebAssembly due to the performance superiority of WebAssembly over JavaScript. We find this assumption reasonable since many studies [10], [11], [12] pointed out the fact that almost all of the cryptojacking scripts that were detected in the wild were based on WebAssembly. For this reason, although it is possible, we do not

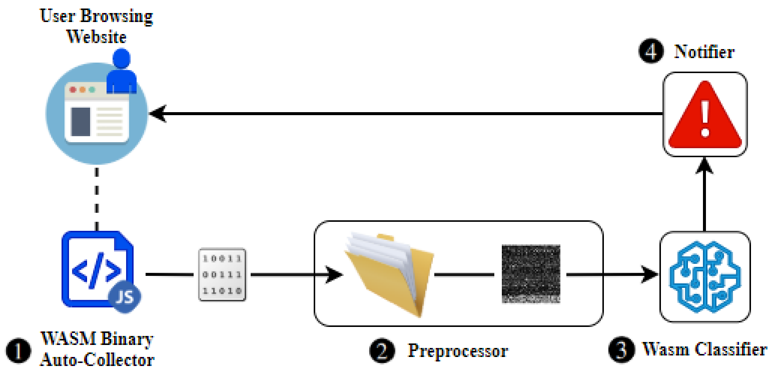


Fig. 3: Overview of the MINOS's framework detailing its four components. The Wasm binary auto-collector downloads Wasm binaries to a designated folder. The preprocessor then converts the binaries to gray-scale images and feeds them to the Wasm Classifier. A pre-trained CNN in the Wasm Classifier classifies images of each binary as either malicious or benign. Finally, the Notifier receives the classification results and alerts the user if results indicate a malicious mining activity is present.

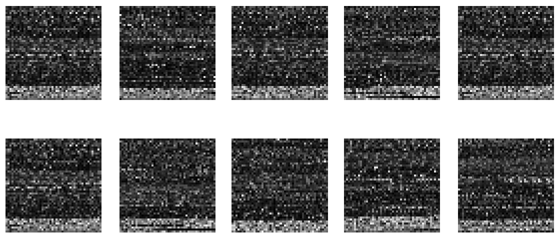


Fig. 4: Gray-scale images of Wasm binaries that belong to cryptojacking samples.

consider the attacker employing pure JavaScript-based cryptojacking implementations. In addition, we do not target legitimate cryptocurrency mining operations (e.g., UNICEF [30]) in the browsers that informs the end-users, and asks for their consent.

V. MINOS FRAMEWORK

A. Inherent Similarity of Cryptojacking Malware

Cryptojacking malware implementations have several inherent similarities as highlighted by Wang et al. [17] and Musch et al. [11]. They are constrained by optimized implementations of specific proof-of-work (PoW) schemes based on memory or CPU-bound hash puzzles. Although cryptojacking malware authors can employ various tactics to prevent their malware from being detected, they still have to implement the same PoW schemes, with the same hashing algorithms. This is in fact, one of the unique characteristics of cryptojacking that distinguishes it from other malware families. Hence, we hypothesized that their implementations should share similar characteristics, and maybe even look syntactically, and semantically similar to each other. In fact, the semantic similarity of cryptojacking malware strains was confirmed by Wang et al. [17]. In order to verify the validity of our hypothesis, we collected Wasm binaries of both benign and cryptojacking samples, and converted them to gray-scale images, as depicted in Figure 4. As shown in this figure, gray-scale images of Wasm-binaries belonging to cryptojacking samples (i.e., the gray-scale images in the first row) are visually very similar to each other. Based on this observation, and motivated by the drawbacks of existing detection mechanisms explained in Section III, we propose MINOS, a novel, lightweight cryptojacking detection system, in this study.

B. System Model

The architecture of MINOS framework, as shown in Figure 3, consists of four primary components: *Wasm module auto-collector, preprocessor, Wasm classifier, and notifier.*

The first component of MINOS is the Wasm Module Auto-Collector. As the user is browsing the Internet, this component checks if the website being visited currently produce any Wasm binaries, and if so, downloads them to a specified folder (❶). The second part is the preprocessor, which reads the specified folder where the Auto-collector downloads the Wasm binaries, and converts each binary in the folder to a gray-scale image (❷). It further preprocesses this image into a format that can be read by the next component. In the third part, the transformed binaries are input to the Wasm Classifier, a pre-trained CNN that classifies each preprocessed binary as either malicious or benign (❸). Finally, the notifier receives the classification results from the CNN and, based on those results, will either alert the user of malicious mining activity or do nothing (❹).

Wasm Module Auto-Collector: As the user is browsing the Internet, this auto-collector is continuously, and simultaneously running in the background, and checking whether each webpage visited is utilizing Wasm. If a certain website is indeed using Wasm, and a Wasm module has been instantiated, the collector automatically downloads and extracts the associated Wasm binary to a specified folder. It should be noted that this script will only download Wasm binaries and no other web page components. In addition, if a webpage loads more than one Wasm module, the auto-collector will download all instantiated Wasm binaries simultaneously to the specified folder.

Preprocessor: Before the Wasm classifier can perform its task, the extracted Wasm binary needs to be preprocessed to convert the data into a format that the neural network is able to use as input. This involves converting each Wasm binary to a gray-scale image, and resizing it to a common size. The preprocessor converts the binary into an array of integers with each integer representing a pixel of a gray-scale image, and then normalizes and reshapes the resulting array. The final reshaped array is then passed on to the Wasm classifier.

Wasm Classifier: The Wasm classifier is a convolutional neural network (CNN) that is pre-trained on a dataset that consists of 150 malicious, and 150 benign Wasm binaries. The structure of the CNN consists of 3 sets of convolution layers followed by max-pool layers with an increasing number of filters in each successive convolution layer (16, 32, and 64). The kernel size used in each convolution layer is set to (3,3), while the pool size of each max-pool layer is set to (2,2). These layers are followed by a final dense layer, with the output being an integer representing whether the sample is benign (0) or malicious (1). The trained neural network is fed the transformed data from the preprocessor as input in order to classify the collected Wasm binary as benign or malicious.

Notifier: If the Wasm classifier classifies the binary in question as malicious, the user is informed that the webpage they are currently visiting is using their computational resources to mine cryptocurrency, and that it is recommended that they close it, and therefore terminate any mining processes running in the background. However, if the binary is classified as benign, the notifier does nothing, and the user continues to browse uninterrupted with the Wasm Module Auto-Collector continuing to check for the instantiation of Wasm modules.

VI. Implementation of Minos

This section provides details regarding the dataset used in the study, as well as the technical implementation of the Minos framework. The framework is implemented as an application written in Python 3 in only 90 lines of code, with a supplemental script written in node.js. The implementation is performed and evaluated on a system running Ubuntu 18.04 with an Intel Core i7-9700K processor and 16 GB of available RAM. The system has a total of 8 cores, with each processor running a base frequency of 3.6 GHz. The current implementation is designed to work specifically with Google Chrome, as it is the world's most used web browser. The following subsections provide an in-depth look into each component and outline details, including algorithms involved and libraries/APIs utilized.

A. Wasm Module Auto-Collector

A script written in node.js provided by the authors of [11] automatically collects, and downloads Wasm binaries to a specified folder as the user is browsing the web. It utilizes Puppeteer, a Node library that is able to communicate with and manipulate/control Google Chrome over the DevTools Protocol, through a high-level API. The code wraps JavaScript functions that instantiate a WebAssembly module such as `WebAssembly.instantiateStreaming`, and logs the module's binary file to the NodeJS backend.

B. Preprocessor

A Wasm module binary consists of a sequence of hexadecimal numbers. This vector of hexadecimal values can be modified and transformed into a gray-scale image. To facilitate

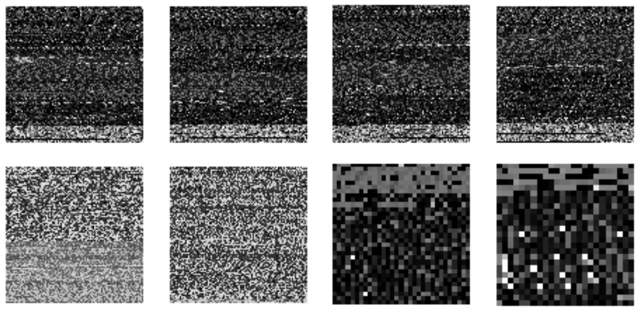


Fig. 5: The first row of gray-scale images belong to Wasm binaries of cryptocurrency mining webpages. The images in the second row represent Wasm binaries of benign webpages, primarily games that employ Wasm.

such a conversion, the Wasm binary is first converted into a vector of 8-bit unsigned integers (`uint8`) and then reshaped into a two-dimensional array. This reshaped array is then divided by 255 to represent each integer as a pixel that takes a value ranging from 0 to 255 (with 0 being black, and 255 being white). These pixels together form a gray-scale image representation of the Wasm module binary. Figure 2 depicts an example of a gray-scale image of a Wasm binary that utilizes the CoinHive cryptocurrency mining service. As shown in the figure, visually distinct regions in the gray-scale image correspond to specific sections in the Wasm binary structure. Examples of binary to image transformations of malicious and benign webpages are shown in Figure 5. As it can be observed from Figure 5, binaries that mine cryptocurrency are visually extremely similar when converted to gray-scale images. This observation also holds true for benign binaries. Another observation is that the images of malicious Wasm binaries are distinct from those belonging benign webpages.

Algorithm 1: Preprocessor

```
def preprocess():
    while $len(Wasm\_directory) == 0$ do
        $time.sleep(1)$
    if $len(Wasm\_directory) != 0$ then
        $Wasm\_images \to []$
        for $file$ in $Wasm\_directory$ do
            $f \to open(file)$
            $ln \to getSize(file)$
            $width \to math.pow(ln, 0.5)$
            $rem \to ln\%width$
            $a \to array('B')$
            $a.fromfile(f, ln - rem)$
            $f.close()$
            $os.remove(file)$
            $g \to reshape(a, (len(a)/width), width)$
            $g \to uint8(g)$
            $h \to resize(g, size, size)$
            $h \to h/255$
            $h \to h.reshape(-1, 100, 100, 1)$
            $Wasm\_images(h)$
    $classify(Wasm\_images)$
return preprocess()
```

The visualization procedure is implemented using a recursive function written in Python 3, the details of which are depicted in Algorithm 1. In Lines 2-3, a while loop is constantly checking whether the destination folder for extracted Wasm binaries is empty at one second intervals (i.e., it is checking every second). In Lines 4-7, once a binary is collected and added to the folder, it is opened and ready for further preprocessing. In Line 5, the variable `Wasm_images` is declared, which will store the converted images. Line 6 ensures that if a website loads multiple Wasm modules, each downloaded module is visited in each iteration of the for loop and is preprocessed. Lines 8-10 calculate length and width parameters, ensuring that the final resized array will have a relatively similar length and width. In Line 11, the file is converted to an array of unsigned integers. In Lines 13-14, after the array is reshaped and the file is closed, the file is deleted from the directory so that once the function has executed, it does not preprocess the same module repeatedly. Lines 15-16 reshapes the created array, and converts it into an array of 8-bit unsigned integers (`uint8`) that range in values from 0 to 255. The value of each integer in the array represents the brightness of a pixel ranging from black to white (0 to 255). In Lines 17-18, the image array is resized to a common size of 100 by 100, and the pixel values are normalized to a range of 0 to 1 by dividing the array by 255. This normalization is done as it is easier for the model to process input arrays with a smaller range of values. Line 19 reshapes the array to a four-dimensional array that the CNN can accept as input, and Line 20 appends the array to the `Wasm_images` list. Once the binary or binaries are converted to images, and added to this list, the `classify()` function is called, which takes the images as an argument. This function will be referenced to, and discussed in the following subsections. The `preprocess` function ends with a recursive call ensuring that it continues to check the directory at one second intervals for new input.

C. Convolutional Neural Network & Notifier

Algorithm 2: Classification Retriever

```
1 def classify(images):
2     $results \rightarrow []$
3     for $ima$ in $images$ do
4         $results.append(model.predict\_classes(ima))$
5     if 1 in $results$ then
6         $notify\_user()$
7 return
```

The Wasm classifier is built using the TensorFlow software library, specifically using TensorFlow's high-level API, Keras. The CNN is written in Python 3 using TensorFlow version 1.13.1. The model is trained across 50 epochs using the RMSprop optimizer, with the learning rate manually set to 0.0001.

Algorithm 2 outlines the `classify()` function called in Line 21 of Algorithm 1. This function retrieves the classification result of each binary collected and converted to images by the `preprocess` function outlined in Algorithm 1. In Line 2, the variable `results` is declared, which will store the results of the classification of each binary. In Line 3, the `model.predict_classes()` Keras function is used to classify each image in `Wasm_images`. The function returns an integer representing whether the sample is benign (0) or malicious (1). Each classification result is appended to the `results` list that the notifier will use to perform its task.

The notifier receives a classification result from the CNN as a list of integers taking values of either 0 (benign) or 1 (malicious). In Line 5, if the list contains any instances of malicious classification (i.e., any 1's), the user is informed via a pop-up dialog created by the `notify_user()` function in Line 6, that the webpage they are currently visiting is attempting to mine cryptocurrency, and that they should close it immediately to prevent continued unauthorized use of their computer's resources.

VII. Performance Evaluation

In this section, details of the dataset used in the study are outlined including sources and number of samples collected. This is followed by an evaluation of the performance of the Wasm classifier and an overhead analysis of both the classifier and the MINOS framework.

A. Methodology

To evaluate MINOS, we use a number of classification accuracy metrics across two different datasets consisting of Wasm binaries and webpages that instrument cryptojacking malware. The first is a dataset of Wasm binaries used to train MINOS. The second dataset consists of webpages in-the-wild that use Wasm and includes benign webpages and those that instrument cryptojacking malware. Details of each dataset are outlined below, followed by a description of the metrics used in our evaluation:

Training Dataset: The dataset used to train the Wasm Classifier consists of 150 malicious and 150 benign Wasm binaries that were obtained from numerous studies and resources. A large portion of the dataset consists of binaries that were collected and used in the following other studies in the literature: SEISMIC [17], MineSweeper [10] and Musch et al. [11]. The remainder of the binaries were collected manually using resources such as VirusTotal [31] and VirusShare [32], NoCoin [33] and MadeWithWasm [34]. A breakdown of the number of binaries collected from each resource, including the distribution of benign and malicious samples, can be found in Table I below. This is followed by a brief description of the other resources and how the binaries were collected from each of them.

TABLE I: Training Dataset Breakdown

	Benign	**Malicious**
SEISMIC	6	4
MineSweeper	4	34
Musch et al.	105	45
VirusTotal and VirusShare	0	63
NoCoin	0	4
MadeWithWasm	35	0
Total	**150**	**150**

VirusTotal and VirusShare: VirusTotal and VirusShare are two popular websites that are used for malware research and practice purposes (e.g., scanning, sharing). Both websites provide various malware samples along with their detection results with respect to several antivirus engines. In order to obtain the newest samples that were detected by antivirus engines, both VirusTotal and VirusShare were used in a way that complemented each other. In this respect, the VirusTotal API enables researchers to automatically check the detection results for a specific sample (up to four queries per minute) but does not allow downloading of such samples. VirusShare, on the other hand, enables researchers to download large packages of malware, but does not have an API. To obtain the newest malware samples, a 73.15 GB malware package shared publicly on the 21st of April, 2020 was downloaded from VirusShare, and the hash values of the samples that have the file type of HTML were extracted. Since HTML samples can contain various malware types (e.g., redirector, downloader, ramnit, trojans, etc.), the VirusTotal Premium API was used to determine the ones labeled as malicious cryptocurrency mining scripts. Using a bash script, the hash values obtained from the VirusShare samples were checked using the VirusTotal API in every 15 seconds. If the VirusTotal API indicates that the sample was identified as cryptojacking malware, then the sample was automatically extracted from the local VirusShare package, and using the Puppeteer API [35], opened in Google Chrome in incognito mode with developer options to examine if it compiles and runs Wasm for cryptojacking.

Using the developed script, we were able to determine 34 unique websites (adult, streaming, forums, etc.) that still perform cryptojacking operations using Wasm. We would like to note that the same web pages and also different pages of the same websites were detected cryptojacking positive by antivirus engines at different time intervals. For instance, antivirus engines detected *piratebay*'s individual torrent search result pages to be employing cryptojacking. However, the website was injecting the same cryptojacking script to every other sub domain as well. Therefore, we omitted multiple occurrences of the same website, and counted websites that had every sub-domain infected as 1. We analyzed the samples, and discovered the addresses of 8 unique mining services still operational. Table II outlines the list of active mining services that use Wasm. When we checked the domain names, we found that, except for `bimeq.com.vn` and `monero.cit.net`, the remaining domains already reside in the NoCoin [33] list. In addition, we realized that the VirusShare malware package that was shared with the community in April 2020 has several cryptojacking samples that try to use the obsolete Coinhive mining service. Since Coinhive is no longer maintained, these samples do not perform any mining operations. Nevertheless, antivirus engines seem label them as cryptojacking malware since they have specific keywords (i.e., Coinhive, miner, mine, etc.). Further, we found that some samples which do not perform any mining operations, and do not have any mining-related script declarations, were falsely detected as cryptojacking samples due to having specific keywords in the actual HTML text.

NoCoin: NoCoin [33] is a browser extension available on Chrome, Firefox, and Opera, that aims to block websites that mine cryptocurrency, using a blacklist. Each website on the blacklist was visited using Google Chrome, one-by-one, to check for the presence of mining activity. This was achieved by opening Chrome DevTools, and manually checking for the instantiation of Wasm modules in the *Sources* tab. Since the blacklist is relatively old, most of the websites no longer exist or are not mining cryptocurrency. Four of the websites still functioned, and as such, the Wasm modules were downloaded. These modules are downloaded in the Web Assembly textual format rather than in binary form, and therefore must be converted to a binary. The conversion is performed using the WebAssembly Binary Toolkit (WABT), specifically, the `wat2Wasm` tool.

TABLE II: Cryptojacking Services Extracted from VirusTotal and VirusShare Samples

`https://statdynamic.com/lib/crypta.js`
`https://www.hostingcloud.racing/ATxh.js`
`https://www.hostingcloud.racing/5Dgk.js`
`https://www.hostingcloud.racing/LGIy.js`
`https://www.hostingcloud.racing/winX.js`
`https://www.hostingcloud.racing/l6nc.js`
`http://biomeq.com.vn/forum/script.min.js`
`http://monero.cit.net/monero/p.js`

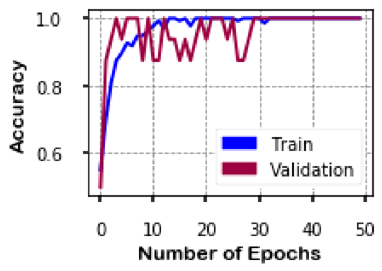
MadeWithWasm: Made with WebAssembly is a website that showcases applications, projects, and websites that use WebAssembly. Each of these were visited, and in a similar fashion to No Coin, using Chrome DevTools, the Wasm modules were downloaded in textual format, and converted to binaries using the WABT. It should be noted that not all of the use cases and websites listed on MadeWithWasm instantiated Wasm modules when checked with Chrome DevTools. A subset of the collected text modules could not be converted to binaries due to errors associated with WABT. The remaining 17 modules were successfully converted to binaries, and added to the dataset of benign samples.

In-the-wild dataset: This dataset consists of websites in the wild including both benign websites, as well as cryptomining websites, that utilize web assembly. This dataset was compiled using two sources namely, PublicWWW [36] and Tranco [37].

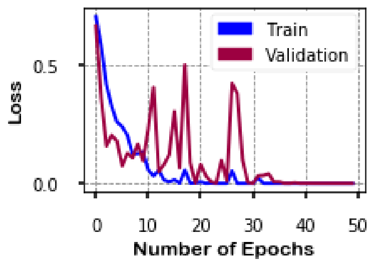
Using the PublicWWW source code search engine, we search for websites that instantiate one or more web assembly modules. This is a realistic representation of cryptomining sites in the wild as MineSweeper's [10] evaluation concluded that 100% of the cryptomining websites they found utilized Wasm while Musch et al. [11] found that in the Alexa 1M, more than 50% of websites that use Wasm, use it for cryptomining purposes. Additionally, we crawled the first 100K websites from the Tranco 1M list (Alexa 1M is no longer published) for the presence of cryptomining activity. Our search returned a combined total of 682 websites from both sources with 613 being benign and 69 malicious.

In order to label our in-the-wild dataset, we followed a similar manual labelling process to Musch et al.'s work [11] and MineSweeper [10] where certain criteria were used to label the in-the-wild samples. The criteria we used are as follows:

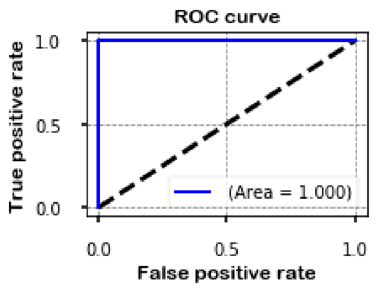
- Checking the decompiler view of the collected Wasm binary for the presence of hashing functions. This approach was also used by MineSweeper [10].



(a) Accuracy



(b) Learning Curve



(c) ROC Curve

Fig. 6: Performance metrics of MINOS framework. Accuracy, learning curve, and ROC curve of MINOS are depicted respectively.

- Checking the file size of the extracted Wasm binary of a website. We found that Wasm binaries of games/applications were substantially larger than those of malicious applications (e.g., 17MB benign and 100KB malicious). This was also reported by Musch et al. [11].
- Checking if a website loads multiple web worker threads. Running multiple hashing operations in parallel is a common behavior of cryptojacking malware.

Classification accuracy metrics: To evaluate the accuracy of MINOS, numerous accuracy metrics are calculated. Using the training dataset, we evaluate the performance of MINOS by calculating its overall accuracy, the learning curve (loss against number of epochs) and the ROC curve (true positive rate against false positive rate). The MINOS framework is further evaluated by testing it against the in-the-wild dataset using metrics such as accuracy, precision, sensitivity, specificity, and F1 score.

B. *Evaluation of* MINOS *against Training dataset*

The performance of MINOS against the training dataset was evaluated based on a number of metrics, including accuracy, optimization loss, and true positive and false positive rate. The dataset was divided into training and testing sets using an 80/20 split. In Figure 6(a), it can be seen that the model converges to 100% accuracy on both the training and test sets after 30 epochs. The optimization learning curve in Figure 6(b) shows that both the training and testing loss decrease to a point of stability after approximately 30 epochs. This indicates that the model is neither overfitting nor underfitting and therefore is able to generalize effectively. Figure 6(c) displays the receiver operating characteristic (ROC) curve for the classifier. The area under the ROC curve is 1, indicating that the model's ability to distinguish between the two classes (benign and mining) is perfect. In addition, this also signifies that the test set's classification results contained no false positives or false negatives.

C. *Evaluation of* MINOS *against in-the-wild dataset*

Using the Puppeteer API, MINOS visited each website in the dataset and determined whether or not cryptojacking malware was implemented on the website. As a result of this scraping process, the Wasm Binary Auto-Collector extracted a total of 1534 Wasm binaries to be analyzed. The reason this number more than doubles the number of websites is due to the fact that the vast majority of benign webpages in the dataset loaded more than one Wasm binary. Additionally, the size of the Wasm binaries belonging to these benign webpages was inconsistent, and had a wide range of values ranging from 1 KB - 10 MB. In contrast, the webpages that were mining cryptocurrency only loaded one Wasm binary and the size was more consistent, taking values between 87 KB - 155 KB.

TABLE III: Accuracy Metrics Against In-the-wild Dataset

Accuracy	Sensitivity	Specificity	Precision	F1 Score
0.9897	0.971	0.9918	0.9307	0.9504

TABLE IV: Overhead of Training MINOS

Wasm Classifier		
	Preprocessing	Training
RAM Usage (%)	4.3	4.1
CPU Usage (%)	2	52
Time (s)	0.497	24.8

MINOS was able to detect and correctly classify 67 malicious webpages, and 608 benign webpages, obtaining an overall accuracy of 98.97%. Table III shows other calculated accuracy metrics illustrating MINOS's performance against this dataset. From the malicious webpages, two were incorrectly classified as benign. Misclassifications in the benign webpages consisted of five false positives where MINOS identified benign webpages as malicious. The benign webpages were mostly e-commerce websites, games, and web-applications while the malicious webpages were either illegal video streaming or adult websites.

D. *Overhead Analysis*

Training MINOS: The overhead of training the framework is summarized in Table IV. During the preprocessing stage, when the Wasm binaries are converted to images, 4.3% of the 16 GB of available RAM is utilized with the conversion of all binaries, taking 0.497 seconds. The entire dataset is stored in virtual memory as an array of arrays taking up a mere 1.448 Kb of space. The total time taken to build, compile, and train the model in 50 epochs was 24.8 seconds. During the training process, a maximum of 4.1% of RAM was used, and no more than 52% of the CPU's processing power was in use.

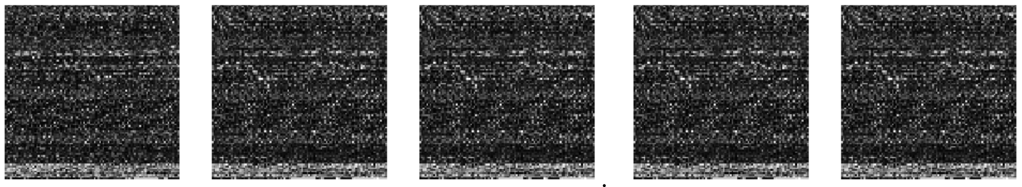

(a) Original miner (b) 25% obfuscated (c) 50% obfuscated (d) 75% obfuscated (e) 100% obfuscated

Fig. 7: Gray-scale image representations of miner binaries with respect to varying amount of obfuscations. a) Binary of un-obfuscated miner, b) 25% of function names obfuscated, c) 50% of function names obfuscated, d) 75% of function names obfuscated, e) 100% of function names obfuscated.

TABLE V: Runtime Overhead of MINOS Implementation

	MINOS Implementation			
	WASM Auto-collector	**Preprocessor**	**Conversion to Image**	**Notifier**
RAM Usage (%)	4	6.3	6.5	6.3
CPU Usage (%)	2	0	4	0
Detection time (s)	0.0259			

MINOS Implementation: The overhead of the implementation of the MINOS framework and its various modules, is outlined in Table V. Since the implementation relies on a pre-trained model, the overhead incurred during training, the model is not considered here. While the Wasm auto-collector runs and checks for the presence of Wasm binaries, 4% of the 16 GB of available RAM, and 2% of the CPU is utilized. As the preprocessor script runs recursively and continues to check for newly collected Wasm binaries, it is utilizing a constant 6.3% of RAM and 0% of the system's CPU. Once the preprocessor detects a newly collected instance or instances of Wasm binaries, the RAM usage varies between 6.3% and 6.5% while the CPU usage increases to 4%. After preprocessing, the CPU usage drops back down to 0% and RAM usage remains at 6.3%. Obtaining the prediction or predictions from the model and notifying the user caused no fluctuations in RAM or CPU usage, indicating that the processing power used during these processes was negligible. The total time is taken to execute MINOS, from the collection of the Wasm binary or binaries to notifying the user was, on average, 0.0259 seconds. Considering this, and the fact that the maximum RAM and CPU usage was 6.5% and 4% respectively, it is evident that the MINOS framework is lightweight, extremely fast and computationally inexpensive.

E. MINOS *Against Obfuscation*

As an attempt to evaluate the robustness of MINOS against adversarial evasions, we chose an open-source browser-based cryptocurrency miner, namely Webminerpool[1] as a basis mining code to obfuscate. Webminerpool provides various Cryptonight PoW implementations written in C, and creates a Wasm-based miner after compilation. Although obfuscation has been an active topic of research in the malware domain, it is an unexplored area of in cryptojacking malware. To the best of our knowledge, no open-source or proprietary obfuscation tool exists for WebAssembly. For this reason, we applied

Listing 1: An example code snippet from the source code of a browser-based cryptocurrency miner.

```

static void copy_block(uint8_t *dst, const uint8_t *src)
{
  ((uint64_t *)dst)[0] = ((uint64_t *)src)[0];
  ((uint64_t *)dst)[1] = ((uint64_t *)src)[1];
}
```

Listing 2: An example code snippet from the source code of a browser-based cryptocurrency miner that has been obfuscated.

```

static void fun55(uint8_t *dst_new, const uint8_t *src)
{
  ((uint64_t *)dst_new)[0] = ((uint64_t *)src)[0];
  ((uint64_t *)dst_new)[1] = ((uint64_t *)src)[1];
}
```

obfuscation before the compilation process of Wasm modules, in the high-level C source-code of Webminerpool. In order to perform the most common obfuscation approach in the wild, *function name obfuscation*, we firstly extracted the function names used in the source and header files of Webminerpool. Then we gradually obfuscated varying amounts of function names in the source and header files. Sample code snippets from the Cryptonight source code (`cryptonight.c`) before and after obfuscation are depicted in Listing 1 and Listing 2 respectively. In the listings, a sample obfuscation is performed by renaming the `copy_block` function to `fun55`.

In order to see the effects of different amount of obfuscations, we firstly obfuscated 25% of the function names in the source and header files of Webminerpool. Then, we increased the number of function names to 50%, 75%, and finally, to 100% in which every function name was obfuscated, excluding C memory management functions (i.e., `memset`, `memcpy`, `malloc` and `free`) [38], [39], [40]. Figure 7 shows the gray-scale image representations of the resulting miner samples. As shown in the figure, although different amounts of function names are obfuscated in the miner code, the resulting binaries of the miner have very similar gray-scale image representations. To evaluate to robustness of MINOS against the obfuscated miner samples shown in Figure 7, we fed the samples to MINOS and observed that MINOS is able to detect the obfuscated binaries regardless of the level or degree of obfuscation.

VIII. DISCUSSION AND BENEFITS

Underlying Concept: MINOS captures the essence of cryptojacking malware due to a unique feature of cryptojacking

[1]https://github.com/notgiven688/webminerpool

malware (i.e., implementation of specific PoW schemes based on memory or CPU-bound hash puzzles) that causes implementations to be syntactically, and semantically similar to each other. This was verified by both our initial analysis in Section V-A, and the study by Wang et al. [17]. In fact, gray-scale images of hundreds of unique cryptojacking malware samples that we analyzed are visually very alike, while looking significantly different from benign Wasm binaries. For these reasons, the image-based classifier of MINOS can recognize the specific patterns, and can capture the essence of cryptojacking malware effectively using gray-scale image.

Lightweight Runtime Overhead: As is evident by the results of our performance evaluation, the overhead incurred during the implementation and actual runtime of the MINOS framework is extremely minimal. While other detection systems in the literature report similar accuracy, their detection methods are based on the analysis of dynamic features. This means that in both the data collection stage, as well during the implementation of these detection methods, the webpages that use cryptojacking malware were allowed to run and effectively mine cryptocurrency until the required features are extracted or until the respective detection method is able to detect the presence of mining activity. Further, the actual collection of features in such dynamic-analysis based systems requires additional resources or supplemental applications and programs. Since MINOS does not use dynamic features, it does not require webpages that instrument cryptojacking malware to run for a certain amount of time to extract relevant features.

Instantaneous Detection Capability: MINOS is capable of detecting cryptojacking scripts instantaneously. As soon as the instantiation and compilation process of Wasm modules is completed in the browser, MINOS immediately converts the resulting Wasm binary to a gray-scale image and classifies it as either benign or malicious. Since MINOS does not rely on dynamic analysis features, the cryptojacking malware is not required to commence its mining process.

Freedom from Administrative Privileges: Cryptojacking detection systems that rely on the monitoring of CPU, memory and cache events require administrative privileges. However, such a necessity introduces additional drawbacks for end-users who do not have administrative rights. In addition, such detection systems which run with administrative privileges may result in other security and privacy issues since they can monitor almost every application running on the host system. MINOS does not force end-users to have administrative rights, and thus ordinary users need not worry about additional security issues.

Platform Independence: MINOS does not utilize browser-specific or operating system-specific features/tools. Although the majority of existing detection systems are similar to MINOS in this respect, not every detection system fulfills this condition. For instance, OUTGUARD [12] relies on browser-specific features which limits its application in other widely used browsers.

Quality of Web Surfing Experience: MINOS has extremely low runtime overhead and successfully detects the cryptojacking scripts even before they begin the mining process. However, existing detection systems either instrument and watch the Wasm modules of every web application, or continuously monitor CPU, memory and network events which may affect the quality of the web surfing experience of end-users. We believe that quality of web surfing experience is a crucial metric for the success of any cryptojacking detection system and that such systems should cause minimal interference to the quality of the web surfing experience of end users. In addition, since more and more benign applications (e.g., Autocad [41]) are moving to the web thanks to WebAssembly and other technologies (e.g., WebWorkers, WebSockets), the performance of such benign web applications needs to be ensured.

Robustness Against Common Evasion Attempts: Prior work in the literature shows that adversaries are utilizing a number of techniques to bypass the detection systems. To evade antivirus engines, they pay attention not to use well-known strings (e.g., *Coinhive, miner*). To bypass blacklists, they frequently change domain names. To eliminate any other detection systems, they throttle the CPU usage of their scripts, set up proxies to hide mining service providers, and use encryption in communication. Although these techniques can be effective against existing detection systems, MINOS stands resilient against all of these attempts since it utilizes only the gray-scale image representation of compiled Wasm binaries.

Since MINOS is a detection tool based on the characteristics of the binary, we have to consider robustness against code obfuscation. Prior cryptojacking studies show that the only code obfuscation attempts seen in the wild are obfuscation of strings and function names. As our analysis demonstrates, MINOS is robust against these obfuscated cryptojacking samples. Currently, no obfuscation tool exists to perform other obfuscation techniques seen in previous malware research, on Wasm binaries. Obfuscation in cryptojacking is an unexplored research and practice area at the moment. In the future, it may be possible for adversaries to find ways to apply advanced obfuscation techniques in cryptojacking, and existing detection schemes including MINOS may be affected by it. However, creating an obfuscation tool for Wasm is beyond the scope of this paper. In addition to obfuscation, adversaries can apply adversarial machine learning (ML) attacks to attempt to evade the deep learning-based classifier of MINOS. Adversarial ML attacks are very common in the image domain, and a number of studies exist on the application of such attacks in the malware domain. However, adversarial ML attacks in the malware domain require that functionality of the malware has to be preserved after the perturbations. If an adversary tries to modify the gray-scale image of his cryptojacking malware using such attacks, the perturbations will result in changes in Wasm code (when converted back to a binary), which can break the functionality of the cryptojacking malware. Similar to the obfuscation case, adversarial ML attacks to cryptojacking detection is an unexplored research domain at the moment.

Generalization of the Detection Technique: Image-based detection has been proven very effective in previous research for Windows Portable Executable-based malware detection. If individual Wasm-based malware families (other than cryptojacking malware) have syntactic and semantic similarities, then it would be possible to generalize MINOS to effectively detect such Wasm-based malware as well.

IX. RELATED WORK

The research conducted by Eskandari et al. [2] was the first study in the literature that pointed out the pervasiveness of cryptojacking. They analyzed censys.io BigQuery dataset that has top 1 million sites of Zmap and 30 thousand websites from the PublicWWW dataset for the existence of Coinhive script. Prevalence of WebAssembly in the wild was also researched by Musch et al. [11], [9] by inspecting Alexa top 1 million websites. Bijmans et al. [8] analyzed one of the largest scale cryptojacking incidents, which consisted of almost 1.4M compromised MikroTik routers injecting cryptocurrency mining scripts to all outgoing connections. Carlin et al. analyzed the growth of cryptojacking in their study [1]. Varlioglu et al. [14] inspected the websites that were previously found to have cryptojacking scripts by CMTracker [16].

Several detection systems exist in the literature, as we discussed in Section III. Among the proposed detection systems, CMTracker [16] is the only scheme that uses fixed thresholds. SEISMIC [17], MineThrottle [15] and MineSweeper [10] employ code instrumentations. ML models based on a number of dynamic analysis features were utilized by RAPID [18], OUTGUARD [12], and CoinSpy [19]. Conti et al. [23] made use of Hardware Performance Counters (HPC) data to detect cryptojacking. Darabian et al. [26] utilized Windows PE opcodes and API calls. In addition to API calls, Berecz and Czibula [27] used both entropy and header, section, and function information to detect miners. Vladimír and Žádník [28] relied on the network flow features and active probing the miner servers. MineCap [29] employed network-flow-based detection via Apache Spark Streaming library and incremental ML model.

Difference from Existing Work: We can consider the differences of MINOS from existing work in two ways: performance-wise or conceptually. In terms of performance-wise comparison, detection accuracy and overhead can be examined. Considering the detection accuracy, we cannot compare MINOS against prior studies due to reproducibility issues. For instance, while only three studies in the literature openly share their source code (CMTracker [16], OUTGUARD [12] and MineSweeper [10]), it is not possible to run those systems for comparison because of either dependency issues with depreciated libraries or model-specific limitations (manually created cryptojacking fingerprints). In terms of overhead comparison, it is not fair to compare MINOS with existing detection systems since those systems are based on dynamic analysis that require cryptojacking samples to run for a certain amount of time, and analyze the memory and cache events, CPU usage, network traffic, or browser events. Depending on the implementation of cryptojacking samples their performances may change. However, MINOS does not run cryptojacking samples, and only converts them to gray-scale images for detection. Conceptually, our work differs from the existing work in several ways: (1) It does not rely on dynamic analysis features, hence it does not require the mining samples to run for a specified period of time for feature collection and detection purposes. Also, (2) the performance of MINOS is not affected by third-party applications running on the host or on the browser. In addition, (3) common evasion techniques used by adversaries (e.g., CPU throttling, dynamically generated domain names, proxies, encrypted communication, etc.) are not effective to bypass MINOS's detection. Unlike existing detection systems, (4) the proposed detection technique does not have high runtime overhead, thus promises a better quality of web surfing experience for end-users compared to other schemes. Further, (5) MINOS does make use of browser or operating system specific features/tools, which makes it platform independent. Moreover, unlike earlier work, (6) it is not necessary for MINOS to have administrative privileges to run on any specific platform. Finally, (7) MINOS represents the first work in the literature that classifies malicious and benign Wasm binaries using gray-scale image representations of the cryptojacking malware.

X. CONCLUSION

Considering the prevalence of Wasm-based cryptojacking malware in the wild, and the high overhead inherent to current dynamic-analysis based detection methods, a detection technique that is able to accurately and rapidly identify instances of such malware with low computational cost is necessary. In this paper, we proposed MINOS, a novel cryptojacking malware detection technique that utilized an image-based classification technique to distinguish between benign and malicious Wasm binaries. Specifically, MINOS implements a convolutional neural network (CNN) model trained with a comprehensive dataset of current malicious and benign Wasm binaries. MINOS achieved exceptional accuracy with a low TNR and FPR. Moreover, our extensive performance analysis showed that the proposed detection technique can detect mining activity on a dataset of webpages in-the-wild in an average of 25.9 milliseconds while using a maximum of 4% of the CPU and 6.5% of RAM, proving that MINOS is highly effective while lightweight, fast, and computationally inexpensive. As future work, we aim to develop a Chrome extension that utilizes the MINOS framework so that the overhead incurred during its operation is reduced and webpages that are mining cryptocurrency without permission are automatically closed.

ACKNOWLEDGMENTS

We would like to kindly thank the authors of [17], [10], [11], and VirusTotal who graciously shared their malware samples. We also thank the anonymous reviewers for their feedback and time. This work was partially supported by the U.S. National Science Foundation (Awards: NSF-CAREER-CNS-1453647, NSF-1663051, NSF-CNS-1718116), and Cyber Florida Capacity Building Program. The views expressed are those of the authors only.

REFERENCES

[1] D. Carlin, J. Burgess, P. O'Kane, and S. Sezer, "You could be mine(d): The rise of cryptojacking," *IEEE Security Privacy*, vol. 18, no. 2, pp. 16–22, 2020.

[2] S. Eskandari, A. Leoutsarakos, T. Mursch, and J. Clark, "A first look at browser-based cryptojacking," in *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2018, pp. 58–66.

[3] "Youtube shuts down hidden cryptojacking adverts," https://www.telegraph.co.uk/technology/2018/01/29/youtube-shuts-hidden-crypto-jacking-adverts/, accessed: 2020-06-02.

[4] "Crypto-streaming strikes back," https://adguard.com/en/blog/crypto-streaming-strikes-back.html, accessed: 2020-06-02.

[5] "Cryptojacking attack found on los angeles times website," https://threatpost.com/cryptojacking-attack-found-on-los-angeles-times-website/130041/, accessed: 2020-06-02.

[6] "Us and uk government websites hijacked to mine cryptocurrency on visitors' machines," https://www.welivesecurity.com/2018/02/12/government-websites-mine-cryptocurrency/, accessed: 2020-06-02.

[7] "Cryptojackers found on starbucks wifi network, github, pirate streaming sites," https://www.bleepingcomputer.com/news/security/cryptojackers-found-on-starbucks-wifi-network-github-pirate-streaming-sites/, accessed: 2020-06-02.

[8] H. L. J. Bijmans, T. M. Booij, and C. Doerr, "Just the tip of the iceberg: Internet-scale exploitation of routers for cryptojacking," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 449–464.

[9] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "Web-based cryptojacking in the wild," *CoRR*, vol. abs/1808.09474, 2018. [Online]. Available: http://arxiv.org/abs/1808.09474

[10] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1714–1730.

[11] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "New kid on the web: A study on the prevalence of webassembly in the wild," in *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2019.

[12] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, "Outguard: Detecting in-browser covert cryptocurrency mining in the wild," in *The World Wide Web Conference*, ser. WWW '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 840–852.

[13] B. Krebbs, "Who and what is coinhive?" https://krebsonsecurity.com/2018/03/who-and-what-is-coinhive/, March 2018, accessed: 2020-06-03.

[14] S. Varlioglu, B. Gonen, M. Ozer, and M. Bastug, "Is cryptojacking dead after coinhive shutdown?" in *2020 3rd International Conference on Information and Computer Technologies (ICICT)*, 2020.

[15] W. Bian, W. Meng, and M. Zhang, "Minethrottle: Defending against wasm in-browser cryptojacking," in *Proceedings of The Web Conference 2020*, ser. WWW '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3112–3118.

[16] G. Hong, Z. Yang, S. Yang, L. Zhang, Y. Nan, Z. Zhang, M. Yang, Y. Zhang, Z. Qian, and H. Duan, "How you get shot in the back: A systematical study about cryptojacking in the real world," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1701–1713.

[17] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "Seismic: Secure in-lined script monitors for interrupting cryptojacks," in *European Symposium on Research in Computer Security*. Springer, 2018, pp. 122–142.

[18] J. D. P. Rodriguez and J. Posegga, "Rapid: Resource and api-based detection against in-browser miners," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 313–326.

[19] C. Kelton, A. Balasubramanian, R. Raghavendra, and M. Srivatsa, "Browser-Based Deep Behavioral Detection of Web Cryptomining with CoinSpy," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[20] L. Babun, H. Aksu, L. Ryan, K. Akkaya, E. S. Bentley, and A. S. Uluagac, "Z-IoT: Passive Device-class Fingerprinting of ZigBee and Z-Wave IoT Devices," in *IEEE ICC*, 2020, pp. 1–7.

[21] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive Information Tracking in Commodity IoT," in *USENIX*, 2018, pp. 1687–1704.

[22] L. Babun, Z. B. Celik, P. McDaniel, and A. S. Uluagac, "Real-time Analysis of Privacy-(un)aware IoT Applications," *PETS/PoPETS*, pp. 145–166, 2021.

[23] M. Conti, A. Gangwal, G. Lain, and S. G. Piazzetta, "Detecting covert cryptomining using hpc," *arXiv preprint arXiv:1909.00268*, 2019.

[24] "Webassembly," https://webassembly.org//, 2020.

[25] A. Rossberg, B. L. Titzer, A. Haas, D. L. Schuff, D. Gohman, L. Wagner, A. Zakai, J. F. Bastien, and M. Holman, "Bringing the web up to speed with webassembly," *Commun. ACM*, vol. 61, no. 12, p. 107–115, Nov. 2018.

[26] H. HDarabian, S. Homayounoot, A. Dehghantanha, S. Hashemi, H. Karimipour, R. M. Parizi, and K. K. Raymond Choo, "Detecting cryptomining malware: a deep learning approach for static and dynamic analysis," *J Grid Computing*, 2020.

[27] G. J. Berecz. and I. Czibula., "Hunting traits for cryptojackers," in *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications - Volume 2: SECRYPT,*, INSTICC. SciTePress, 2019, pp. 386–393.

[28] V. Veselý and M. Žádník, "How to detect cryptocurrency miners? by traffic forensics!" *Digital Investigation*, vol. 31, p. 100884, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287618303359

[29] H. N. C. Neto, M. A. Lopez, N. C. Fernandes, and D. M. F. Mattos, "Minecap: super incremental learning for detecting and blocking cryptocurrency mining on software-defined networking," *Ann. des Télécommunications*, vol. 75, no. 3-4, pp. 121–131, 2020.

[30] "The hope page," https://www.thehopepage.org/, accessed: 2020-04-13.

[31] "Virustotal," https://www.virustotal.com/, accessed: 2020-06-03.

[32] "Virusshare," https://virusshare.com/, accessed: 2020-06-03.

[33] "Nocoin adblock list," https://github.com/hoshsadiq/adblock-nocoin-list, accessed: 2020-06-03.

[34] "Made with webassembly," https://madewithwebassembly.com/, accessed: 2020-06-03.

[35] "Puppeteer," https://pptr.dev/, accessed: 2020-06-03.

[36] "Publicwww," https://publicwww.com/, 2020.

[37] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, ser. NDSS 2019, Feb. 2019.

[38] J. Lopez, L. Babun, H. Aksu, and A. S. Uluagac, "A Survey on Function and System Call Hooking Approaches," *Journal of Hardware and Systems Security*, vol. 1, no. 2, pp. 114—136, 2017.

[39] L. Babun, H. Aksu, and A. S. Uluagac, "A System-Level Behavioral Detection Framework for Compromised CPS Devices: Smart-Grid Case," *ACM TCPS*, vol. 4, no. 2, 2019.

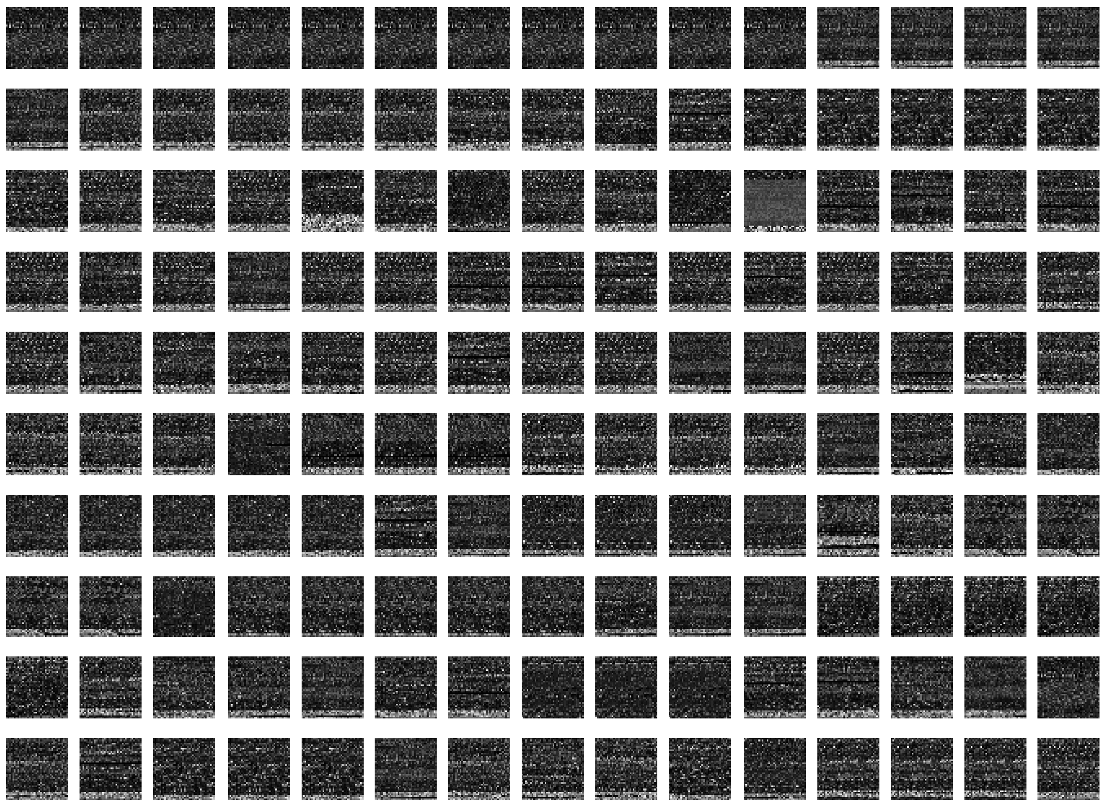
[40] L. Babun, H. Aksu and A. S. Uluagac, "Identifying Counterfeit Smart Grid Devices: A Lightweight System Level Framework," in *IEEE ICC*, Paris, France, 2017, pp. 1–6.

[41] "Tutocad webassembly: Moving a 30 year code base to the web," https://www.infoq.com/presentations/autocad-webassembly/, accessed: 2020-04-13.

Appendix

A. Gray-scale Wasm Images

Gray-scale images of all the benign and malicious Wasm binaries included in our training dataset. The vast majority of the mining samples are visually, extremely similar. This means that at the binary level, they are executing either similar or identical instructions (hashing algorithms). In addition, it can be seen that many of the benign samples are unique, and in some cases they look completely dissimilar to the malicious samples. The reason for this is that most of the benign samples are web applications or games that execute binary instructions that vastly differ from those executed by malicious samples.


(a) Malicious Samples

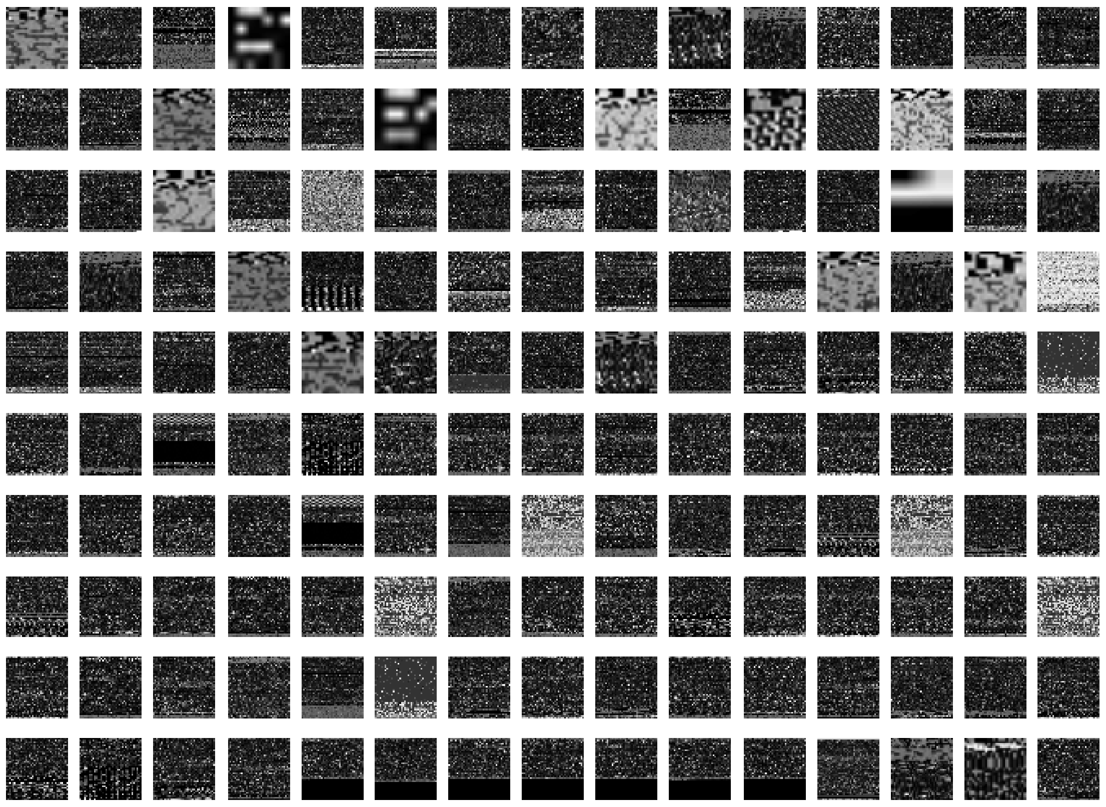

(b) Benign Samples

Fig. 1: GRAY SCALE IMAGES OF ALL 300 WASM BINARIES IN OUR TRAINING DATASET: (A) 150 MALICIOUS WASM SAMPLES AND (B) 150 BENIGN SAMPLES.