MinerGate: A Novel Generic and Accurate Defense Solution Against Web Based Cryptocurrency Mining Attacks

Guorui Yu[1], Guangliang Yang[2], Tongxin Li[1], Xinhui Han[1(✉)], Shijie Guan[1], Jialong Zhang[3], and Guofei Gu[2]

[1] Peking University, Beijing 100871, China
{yuguorui,litongxin,hanxinhui,1600012835}@pku.edu.cn
[2] Texas A&M University, Texas, TX 77843, USA
ygl@tamu.edu, guofei@cse.tamu.edu
[3] ByteDance AI Lab, Beijing 100098, China
zjl.xjtu@gmail.com

Abstract. Web-based cryptocurrency mining attacks, also known as cryptojacking, become increasingly popular. A large number of diverse platforms (e.g., Windows, Linux, Android, and iOS) and devices (e.g., PC, smartphones, tablets, and even critical infrastructures) are widely impacted. Although a variety of detection approaches were recently proposed, it is challenging to apply these approaches to attack prevention directly.

Instead, in this paper, we present a novel generic and accurate defense solution, called "MinerGate", against cryptojacking attacks. To achieve the goal, MinerGate is designed as an extension of network gateways or proxies to protect all devices behind it. When attacks are identified, MinerGate can enforce security rules on victim devices, such as stopping the execution of related JavaScript code and alerting victims. Compared to prior approaches, MinerGate does not require any modification of browsers or apps to collect the runtime features. Instead, MinerGate focuses on the semantics of mining payloads (usually written in WebAssembly/asm.js), and semantic-based features.

In our evaluation, we first verify the correctness of MinerGate by testing MinerGate in a real environment. Then, we check MinerGate's performance and confirm MinerGate introduces relatively low overhead. Last, we verify the accuracy of MinerGate. For this purpose, we collect the largest WebAssembly/asm.js related code with ground truth to build our experiment dataset. By comparing prior approaches and MinerGate on the dataset, we find MinerGate achieves better accuracy and coverage (i.e., 99% accuracy and 98% recall). Our dataset will be available online, which should be helpful for more solid understanding of cryptojacking attacks.

Keywords: Cryptojacking · WebAssembly · asm.js

W. Lu et al. (Eds.): CNCERT 2020, CCIS 1299, pp. 50–70, 2020.
https://doi.org/10.1007/978-981-33-4922-3_5

1 Introduction

Recently, cryptocurrency mining attacks, also known as cryptojacking attacks, are becoming increasingly popular. Different from regular attacks, which usually aim at the access or destruction of private data or sensitive functionalities, this attack mainly focuses on stealing the computing resources (e.g., CPU) of victim Internet-connected devices for covertly mining cryptocurrencies and accumulating wealth.

Although the mining attack does not make malicious and notorious actions, it can still cause serious consequences. For example, the mining code usually occupies the most (or even the whole) of physical resources (e.g., CPU, Memory, and network), which results in all services and apps in the victim devices become inactive, unresponsive, or even crashed. Furthermore, this attack also significantly reduces the life cycle of hardware, such as the battery of laptops and smartphones.

With the significant development of web techniques (e.g., WebSocket [31], Web Worker [30], WebAssembly [9], asm.js [6]), more and more mining attacks are moved to the web platform, which means they can be simply launched by embedding JavaScript snippets. The attack scenario is shown in Fig. 1. First, in the victim websites, attackers include mining script code [15, 16], which is used to initialize the environment, and download and execute mining payloads. Please note that in general the mining payloads are written in WebAssembly/asm.js, which are intermediate languages and allow web browsers to run low-level languages (e.g., C/C++) for near-native performance.

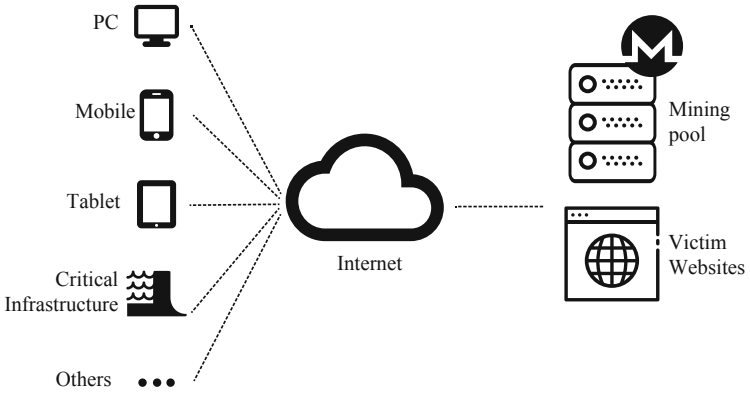

Fig. 1. Example attack scenarios

Up to now, a large number of diverse platforms (e.g., Windows, Linux, Android, and iOS) and devices (e.g., PC, smartphones, tablets, and even critical infrastructures) have been widely impacted. For example, recent studies [1, 5] showed popular applications running in smartphones or PC might silently launch the attacks by executing web mining payloads in the background. Furthermore, critical infrastructures (e.g., industrial control systems) may also be threatened by the mining attack. A recent report showed that a water utility [20] was attacked which might cause its industrial control application to be paused and even crashed.

Even worse, the attack may be hardly stopped once the related web code is executed in the background. A recent report [4] showed attackers could even continue mining with the help of service worker after closing the infected web page.

Therefore, a defense solution that can provide protection on all various devices and eliminate the threats of mining attacks is expected. Recently, a variety of detection solutions [10, 12, 13, 17, 32] have been proposed. However, these approaches do not meet the requirements. First, they are not scalable. Most of them [13, 17, 32] require the modification of web browser engines to collect runtime features, such as the usage of CPU, memory, and network activities. The above solutions not only bring considerable additional overhead to the browser, but also make it difficult to deploy the defense. Second, in case users access infected websites, the mining code should be immediately stopped. However, prior approaches [10, 12] does not meet the requirements. Third, the user experience should not be significantly influenced. However, prior tools may introduce high overhead. For example, [32] introduced almost 100% overhead.

Furthermore, prior approaches may face high false positives and negatives. To identify mining code, they either use a blacklist to block the access of infected websites, or leverage heuristic features to detect mining code. For the blacklist-based tools (e.g., Firefox [2]), it is difficult to keep up with the rapid iteration of mining websites, and thus may cause high false negatives. For the heuristic features, these features mainly include 1) the usage of CPU, memory, and network, 2) CPU cache events, and 3) cryptographic instructions. In our test, we find it is challenging for existing approaches to distinguish between benign CPU-intensive code and mining code.

Instead, in this paper, we propose a novel, general, and accurate protection solution, called MinerGate, that can automatically and effectively provide security enforcement on end devices against mining attacks. To achieve the goal, MinerGate is deployed as an extension of network gateways or proxies. As shown in Fig. 2, our approach is three-fold. First, MinerGate monitors network traffic to catch cryptocurrency mining payloads. For this purpose, MinerGate instruments all network traffic by injecting predefined JavaScript code "stub.js", which will be executed in local devices. The injected stub code is responsible for extracting WebAssembly/asm.js code and enforcing security rules. When stub.js uncovers web code written in WebAssembly/asm.js in a victim device, it will send the content or the reference of related code to MinerGate for further analysis.

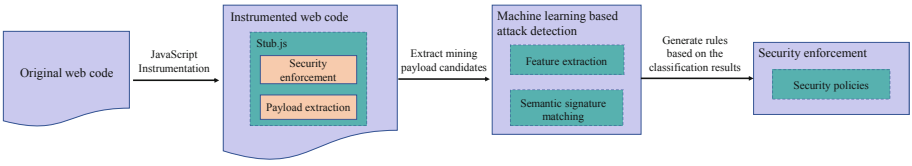

Fig. 2. MinerGate's workflow

Second, different from prior approaches, which rely on the analysis on collected runtime features, MinerGate mainly focuses on understanding the semantics (e.g., CG

and CFG) of WebAssembly/asm.js code. Through data-driven feature selection, MinerGate determines and extracts semantic-related features and forwards these features to a machine learning engine for determining the existence of mining code. Last, once mining code is found, MinerGate notifies the victim device (i.e., stub.js) to apply security rules, such as stopping the execution of web code and alerting the victim user and the network administrator.

In our evaluation, we first verify the correctness of MinerGate by testing MinerGate in a real environment. Then, we check MinerGate's performance and confirm MinerGate introduces relatively low overhead. Last, we verify the accuracy of MinerGate. For this purpose, we first address the challenge there is still not a reliable labeled dataset of cryptojacking mining payloads. To create such a dataset with ground truth, we systematically collect WebAssembly/asm.js code from the 10 million web pages, and NPM [11]. As a consequence, our dataset includes not only mining code from 4659 pages, but also 243 projects related to benign WebAssembly/asm.js. We will open up this dataset for the follow-up research. This dataset should be helpful for a better understanding of mining attacks.

Based on the dataset, we compare MinerGate and prior tools. We find MinerGate achieves better accuracy and coverage (i.e., 99% accuracy and 98% recall).

To sum up, we make the following contributions:

- We propose the novel, generic and accurate defense solution "MinerGate" against mining attacks.
- MinerGate obtains high accuracy by extracting and applying semantic-based features with help of call graph (CG) and control flow graph (CFG).
- We build the largest ground truth dataset.
- We compare MinerGate and existing related approaches, and show MinerGate is scalable, effective and accurate.

2 Background

2.1 Cryptocurrency Mining and Cryptojacking Attacks

Cryptocurrencies are digital assets designed to work as a medium of exchange that uses strong cryptography to secure financial transactions, control the creation of additional units, and verify the transfer of assets [34]. The cryptocurrency uses a distributed database, blockchain, to store the transactions in units of blocks. Each block mainly includes a unique ID, the ID of the preceding block, the timestamp, the nonce, the difficulty, and transaction records. A valid block contains a solution to a cryptographic puzzle involving the hash of the previous block, the hash of the transactions in the current block, and a cryptocurrency address which is to be credited with a reward for solving the cryptographic puzzle. The specific cryptographic puzzle is to find a block of data whose hash value is smaller than a set value which is decided by the difficulty. Most data of the block are known, and the miner should find the unknown part in a limited time. Once the pronumeral, typically is the nonce, is found, the miner will submit it to get profit. This process is called cryptocurrency mining [7].

Cryptojacking, the unauthorized use of hardware of others to mine cryptocurrency, has become the biggest cyber threat in many parts of the world. Cryptojacking was a burgeoning industry in 2018, there have been 13 million cryptojacking attempts in the case of a 40% increase in 2018 [19]. Using crypto-mining malware, criminals have mined earning up to 56 million USD in 2018. There are many reasons why cryptojacking is overgrowing. One of the most important reasons is the simplicity of deployment. Cryptojacking can be easily deployed by inserting a statement in the HTML, such as <script src="attacker.com/mining.js"></script>. This allows the attackers to deploy mining payloads to victim websites without actual control because of XSS or other vulnerabilities. The simplicity of cryptojacking leads to the threat of cryptojacking attacks as long as the cryptocurrency exists. There is no correlation between the existence of such an attack and whether or not a service is alive.

2.2 Related Web Techniques

In past years, web techniques made tremendous progress, which makes it feasible to launch mining attacks using web code. For example, the worker mechanisms provide the possibility of running web code in parallel and the background. WebAssembly/Asm.js provide chances to run mining code in machine-instruction level.

Asm.js is an embedded domain specific language that serves as a statically typed assembly-like language. It is a JavaScript subset that allows web code written in low-level languages, such as C/C++. In order to apply asm.js in runtime, the function body of asm.js code must define with a directive prologue "use asm" or "most asm". WebAssembly [26] is an abstraction over modern hardware, making it language-, hardware-, and platform-independent, with use cases beyond just the Web. WebAssembly is a binary instruction format (bytecode) for a stack-based virtual machine which is different from a text form of asm.js. WebAssembly is designed as a portable target for compilation of high-level languages like C/C++/Rust. Moreover, WebAssembly is committed to getting the speed closer to the native code, and it is faster than asm.js. Currently, WebAssembly can be only be loaded and executed by JavaScript, JavaScript calls WebAssembly in three steps: 1) loading WebAssembly bytecode, 2) compiling bytecode, and 3) instantiating and executing compiled code.

Asm.js and WebAssembly have similarities in many respects. For example, they are both statically typed assembly-like languages, and they have similar instruction sets, which makes it possible for them to convert between each other.

The earnings of cryptojacking attackers are strongly related to the mining speed, so the attackers implement the core logic of mining with WebAssembly and asm.js. We suggest it is more effective and robust to analyze the WebAssembly/asm.js code instead of other scaffolding code. Previous works related to WebAssembly/asm.js malware analysis only concentrate on instruction features, which makes it challenging to classify mining applications.

3 System Overview

3.1 Challenges and Our Solutions

In order to design and implement a generic defense solution against web-based mining attacks, several challenges are raised. More details about these challenges and our corresponding solutions are discussed below.

- *Diverse platforms and devices.* Nowadays, many different devices, such as PC, mobile devices and infrastructure devices, are connected to the Internet. They all are potentially affected by mining attacks. Considering these devices usually have their own operating systems, it is challenging to offer general protection.
  To address it, we design and implement MinerGate as an extension of a network proxy (e.g., network firewall or gateway). MinerGate can protect all devices behind it. In practice, once a mining attack occurs, MinerGate can enforcedly stop the attack code and alert network administrators.
  Please also note that considering HTTPS are frequently used, we assume that MinerGate can monitor all network traffic, including HTTPS-based communication. This can be achieved by installing MinerGate's certificate in all devices under the protection.
- *Obfuscated web code.* Web code, especially the code injected by adversaries, is frequently obfuscated in practice. This poses challenges to extracting adversaries' essential mining code. To address the problem, MinerGate instruments the web code and hijack crucial global JavaScript APIs, which are helpful to extract the parameters related to mining code.
  However, due to the natural flexibility of JavaScript, adversaries may still bypass the above solution. To deal with this issue, we introduce a self-calling anonymous function to protect instrumented web code, and carefully handle the creation of new JavaScript contexts.
- *Unknown mining semantics.* As introduced in Sect. 2, WebAssembly/asm.js have been widely deployed in web mining code. However, up to known, their inside semantics are still unclear, especially considering there are already many variants of the existing mining code. This may significantly reduce the detection accuracy.
  To address this problem, we do program analysis on WebAssembly/asm.js code and extract all call graph (CG) and control flow graph (CFG). Although CG and CFG are basic things for program analysis, automatically generating CG and CFG is still not an easy task, especially considering indirect-call instructions are frequently used.
- *Difficulty of mining code determination.* WebAssembly/asm.js is frequently used not only in mining but also in another area, such as gaming and data processing. It is difficult to distinguish between them accurately. In this work, we address this issue by applying machine learning. However, although existing work discovered a variety of features available for machine learning, they may cause high false positives.
  Instead, we extract features from mining semantics (e.g., CG and CFG) and obtain high accuracy. However, it is challenging to apply graph-based features in machine learning, which cause performance issues and affect scalability. To handle it, we analyze the code in units of semantic modules instead of functions or files to break the solid lines in the analysis.

- *Difficulty of stopping mining code.* Once mining attacks occur, hardware resources (e.g., CPU and memory) may be immediately occupied by adversaries. This poses challenges to stopping the corresponding malicious code in time.
  To deal with this problem, we stop the execution of the mining thread through the function hijacking beforehand and cut off the source of malicious code.
  As shown in Fig. 2, MinerGate contains three major modules: 1) JavaScript Instrumentation, which is used to instrument network traffic to inject stub.js for extracting WebAssembly/asm.js code, and enforcing security rules; 2) Machine Learning Based Detection, which can do program analysis on payloads to extract semantic-related features; 3) Security Enforcement, which defines and enforces security rules. For each module, more details are presented in the following sections.

3.2 JavaScript Instrumentation

As introduced in Sect. 3, MinerGate injects the JavaScript file "stub.js" into all web code to extract WebAssembly/asm.js code and apply security enforcement. This is achieved by hijacking and instrumenting several crucial JavaScript APIs. Please also keep in mind that the stub.js file is always placed at the beginning of web code, which can ensure all target JavaScript APIs are already instrumented before they are actually used by mining code.

In the next subsections, we explain how stub.js works. Furthermore, we also present our protection, which prevents adversaries bypass or destroy stub.js and our instrumented JavaScript APIs.

WebAssembly/asm.js Code Extraction: Our JavaScript API hijacking solution is designed based on the key observation: no matter where adversaries save the mining code, such as a URL or encrypted string, the key JavaScript APIs, such as `WebAssembly.instantiate` for WebAssembly must be called. Hence, in stub.js, we hijack all crucial JavaScript APIs to extract and collect all required parameters, which are sent back to MinerGate for further analysis. These hijacked APIs are listed in Table 1.

Table 1. Hooked APIs for WebAssembly mining payload extraction.

WebAssembly API	Description
instantiate()	Compiles and instantiates WebAssembly code
instantiateStreaming()	Compiles and instantiates a module from a streamed source
compile()	Compiles a Module from WebAssembly binary code
compileStreaming()	Compiles a Module from a streamed source
Module()	Synchronously compiles WebAssembly binary code to a Module

Let us use `WebAssembly.instantiate` as an example to describe how these APIs are hijacked and instrumented. Because JavaScript is a dynamic language, all objects can be replaced so that we can forge a `WebAssembly.instantiate` function

object and replace the original one. In this fake function, we first use a WebSocket connection to send the function parameter (the WebAssembly payload) asynchronously to the gateway and continue to execute the original code. No matter how the mining code is saved and how the code is obfuscated, the mining code will be identified and sent to MinerGate. In addition, the payload is sent asynchronously, without blocking code execution and increasing overhead.

For asm.js, we need some extra effort to extract them. Since attacker can dynamically invoke the asm.js compiler by APIs like `eval`, `Function`, etc. We need to hijack any API that will trigger code compilation. As introduced in Sect. 2, before the asm.js code is parsed and compiled, it must be defined with the prologue directive "use asm" or "most asm" [6]. This principle offers hints to extract asm.js code from APIs. More specifically, we first do syntax analysis on the parameter of `eval` to build the AST. Next, we scan the AST to identify all functions. Then, we check each function to determine the existence of "use asm". Finally, in addition to the asm.js that appear directly in the HTTP traffic, the payloads found in the API are also sent to the gateway for analysis.

In addition to extracting WebAssembly/asm.js code, stub.js are also used to enforce security rules. More details are discussed in Sect. 3.5.

```
(function () {
  var ori_api=WebAssembly.instantiate;
  WebAssembly.instantiate = function (buf, importObj){
    if (isMalicious(buf)) {
      // Refuse to load malicious modules.
      return null;
    } else {
       return ori_api(buf, importObj);
    }
  };
})();
// Variable "ori_api" will not able to be accessed out of the scope.
```

Protections on stub.js: The stub.js solution can effectively extract the mining code and apply security enforcement. However, there are still several ways that adversaries may bypass and destroy the solution. To mitigate the problem, we provide the following protections:

- *Locating original APIs.* Considering if adversaries can find and access that variable, adversaries may still normally and freely use the hijacked APIs. To address this issue, we place stub.js inside a self-calling anonymous function. As a result, even though adversaries may find the local variables where the original APIs are saved in, such as calling `Function.toString()` to check the source code of the hijacked APIs, adversaries cannot still access them.

Furthermore, to improve the security of `toString()` and hide our defenses roughly, we can also hijack the function `toString()` to confuse the attackers.

- Starting a new web context. Mining code may use worker and iframe to run the mining payload in the background to keep the responsiveness of the main thread. Since worker and iframe create new JavaScript contexts, existing hijacked APIs becomes ineffective in the new contexts. Hence, stub.js is required to be executed again and right after the initialization of the new contexts. To achieve it, the worker and iframe object are also hijacked through Web traffic instrumentation.

More specifically, for a worker, we implement a worker agent object to protect the crucial API Worker. When a worker is created, an agent object is returned for replacement. This worker agent has the same interface as the native worker, but it will stitch the stub.js together with the original code to protect the APIs existing in the worker. We also emphasize that any subsequent calls to the worker API within this context will be protected, regardless of how it is called.

In addition to the protection mentioned above, to respond to some existing attack methods [18], such as prototype poisoning, abusing the caller-chain, etc., our work also includes defense against these attacks.

3.3 Machine Learning Based Detection

As discussed in Sect. 1, the simple heuristic features used by prior approaches may cause high false positives. This is because applications in the real world may contain instruction patterns similar to mining algorithms, such as video decoding and data encryption/decryption. This scenario makes it difficult to determine the type of programs based on the occurrences of specific instructions without context. To achieve higher accuracy, we mainly improve from two aspects. On the one hand, we add more features through data-driven feature selection; on the other hand, we divide the code into different "modules" by running the clustering algorithm on the function call graph, which helps us reduce data dimensions, improve the performance and enhance resistance to code obfuscation. The overall classification flow is described in Fig. 3.

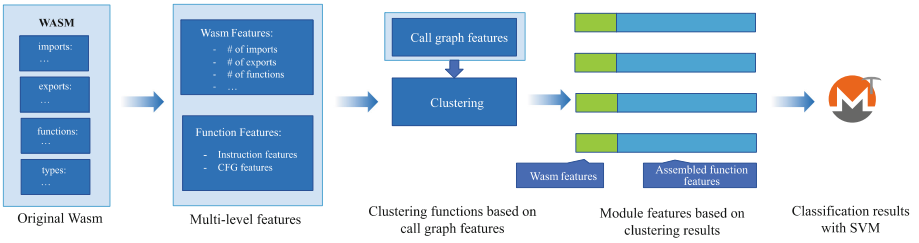

Fig. 3. The classification flow of a WebAssembly module.

CG and CFG Generation. It is worth noting that our program analysis is mainly done on WebAssembly code. There are several reasons. First, the asm.js code can be easily converted to WebAssembly bytecodes (e.g., using the "asm2wasm" tool). Second, the

WebAssembly language is well designed. Its bytecodes are simple, clean, and also easier for analysis.

Our analysis is done as follows. First, once the reference (e.g., URL or string) of WebAssembly/asm.js code is obtained, MinerGate constructs the corresponding WebAssembly binary file. Language transformation is also required if the asm.js code is faced. Next, all instructions are carefully analyzed. In a function, adjacent regular instructions (without branch and function invocation instructions) stick together as a basic block. Branch and function invocation instructions link different blocks. Considering the simplicity of WebAssembly bytecodes, this graph construction work can be easily done.

However, there is also a challenge raised in the process. When an indirect function invocation instruction is faced, it is difficult to determine the target function. Our solution is based on the observation: in runtime, when the instruction is executed, the target function's prototype F_{target} must matches the function prototype $F_{expected}$ determined by instruction itself. Therefore, MinerGate retrieves $F_{expected}$, and scan all functions with proper prototypes to determine the callee function candidates. To avoid false negatives, MinerGate links the function invocation instruction with all function candidates. Our evaluation also shows this simple solution also has relatively low false positives.

CFG Features. The critical point in mining code detection is the feature section, because of previous work relied on heuristic methods, we use data-driven feature selection to fill up the missing part of CFG in existing methods by statistics of the graph. Most graph analysis methods rely on graph statistics. Graph statistics can be used to compare graphs, classify graphs, detect anomalies in graphs, and so on. Graph's structure is mapped to a simple numerical space through graph statistic, in which many standard statistical methods can be applied.

In this paper, we introduce graph statistics as an essential part of the analysis of WebAssembly/asm.js. Examples of graph statistics are the number of nodes or the number of edges in a graph, but also more complex measures such as the diameter. Overall, graph statistic can be roughly divided into two categories, global statistics, and nodal statistics. The former describes the global properties of the graph, so only one number is needed for each graph to describe an attribute, and the latter describes the attributes of the nodes in the graph, so each attribute is represented by a vector. In order to analyze the CFG graph as a whole, we use global statistics of the graph as our CFG features, such as graph size, graph volume, graph diameter, etc. When selecting the statistical features of the graph, we mainly consider the work of [3, 14, 33].

Instruction features. CryptoNight [27], which is a hash algorithm and heavily used in mining software, explicitly targets mining on general CPUs rather than on ASICs or GPUs. For efficient mining, the algorithm requires about 2 MB of high-speed cache per instance. Cryptography operations, such as XOR, left shift and right shift, are commonly used in CryptoNight algorithm so that we will examine their influences here. In addition to this, we also consider other instructions, not limited to the instructions described earlier, such as various control flow related instructions, memory access instructions, arithmetical operation instructions, and so on.

3.4 Data-Driven Feature Selection

We obtained 114 candidate features through the above steps. In our model, we assume that the functions in the mining samples are all related to mining, besides they are mining-related after our manual analysis, and the functions in the benign samples are not related to mining. For the estimation of dependence between features and classes, we use the χ^2 Test [8], which is commonly used in machine learning algorithms to test dependence between stochastic variables. Following this, we will get scores of features which can be used to select the top N features with the highest values. Part of the top features are shown in the Table 2.

Table 2. Top features

Features	Category
Max size of basic blocks	Graph
CFG size	Graph
CFG volume	Graph
Max out degree	Graph
CFG diameter	Graph
Number of loops	Graph
Number of branches	Graph
Number of branches	Instruction
Number of memory instructions	Instruction
Number of arithmetical instructions	Instruction
Number of cryptography instructions	Instruction
Number of instruction `get_local`	Instruction
Number of instruction `set_local`	Instruction

We can see from the Table 2 that the graph-related features are more effective than the instruction features, which may be due to the special CFG patterns of the mining code. We can also find that it confirms the previous results [13, 32], cryptography instructions do have influences on the classification results. However, those CFG-related features are more relevant to results. Besides, memory access instructions also showed in the ranking, which is consistent with the fact that the mining code is a memory-intensive application.

Overall, we demonstrate the effects of CFG features and their impact in this section. We will select the top 10 features in the ranking as the basis for subsequent analysis, so each function is represented by a vector of length 10. At this point, we get the features of each function.

Semantic Signature Matching. The instruction features or CFG features we discussed earlier can measure the functionality of a piece of code, such as a function or an entire file. The next problem is how to use these features to ensure effectiveness and robustness.

When we examine a payload by analysis of each function, it is difficult to set a proper threshold of malicious functions to discriminate malicious samples. There are many reasons for this dilemma. For example, a library for encryption, it may contain a small number of functions similar to the mining code. On the other hand, malware can also hide in many unrelated code and minimize the number of functions. Similarly, we also face a similar problem when we analyze the payload as a whole.

In this section, we use DBSCAN [28] clustering algorithm to break the solid lines in the analysis. Specifically, we divide the functions into modules according to the call graph (CG), then we generate feature vectors for each module. With clustering functions together, we combine tightly coupled functions into one module, which breaks the boundary between functions, reduces the complexity of data dimension, and enhances the ability against code obfuscation.

DBSCAN is one of the most well-known tools for clustering based on density. The algorithm grows regions with sufficiently high density into clusters and discovers clusters of arbitrary shape in spatial databases with noise. A significant advantage of DBSCAN is that it does not require the number of clusters a priori, unlike k-means, which needs to be specified manually. The number of modules in a payload is uncertain, and the DBSCAN can determine the number of clusters for us. Another advantage is that it does not rely on Euclidean distance, because it is inappropriate to convert the CG to Euclidean distance.

The algorithm requires two parameters: ϵ-neighborhood of points and the minimum number of points (*MinPts*) required to form a dense region. In order to apply the algorithm to our domain, we need to redefine the ϵ-neighborhood $N_{\in}(p)$ of a point (a function in this paper), $N_{\in}(p) = \{q \in D | if\ p\ calls\ q\}$, in which D means the database of the functions.

When $MinPts = 4$, the results of the cluster analysis on the mining payload of CoinHive are shown in Fig. 4. For the sake of brevity, only functions related to `cryptonight_hash_variant_1` are included in the figure. It can be seen that functions related to `cryptonight_hash_variant_1` are divided into two clusters. With manual analysis, it can be seen that the functions in Cluster 1 are mainly related to encryption, and functions in Cluster 2 are mainly related to memory operations. The main reason for this result is that the effect of code is closely related to the functions it calls.

Then we generate the feature vectors for each module with the methods described in Sect. 3.3, which combine with the labels will be used to train the SVM classifier. If the analysis result for a sample contains one or more malicious "modules", we label the whole sample as malicious.

3.5 Security Enforcement

Although the user is executing malicious code while detection is occurring, the main threat of cryptojacking is it occupies a lot of system resources, instead of stealing sensitive information or damaging the system like traditional malware. As long as it can prevent its operation in time, its impact is limited. Our security enforcement is mainly provided in the injected stub.js (Sect. 3.2). With detection of the mining code, MinerGate notifies stub.js through pre-established WebSocket connection. This connection can be kept alive even when CPU, memory, and network are occupied by mining code.

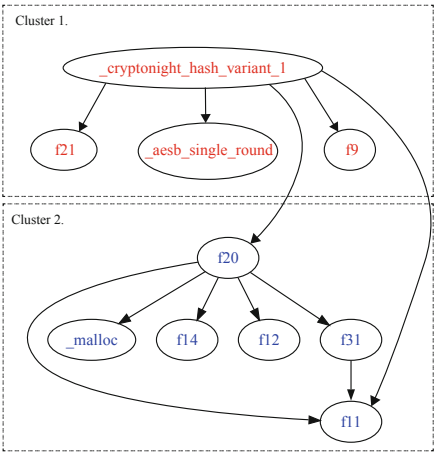

Fig. 4. The clustering result on CoinHive with *MinPts* = 4, only includes functions that are related to `cryptonight_hash_variant_1`.

Stub.js can also apply pre-defined security rules. For example, stub.js can directly terminate the execution of the mining code. This is achieved by stopping the worker or removing the iframe with preset callback functions, and mining code running in the main thread of the web page will be closed immediately. Hijacked APIs (e.g., `eval`, `WebAssembly.instantiate`, etc.) in users' browser will refuse to execute code that are marked as untrusted.

The mining code needs to use WebSocket to communicate with the mining pool to obtain the necessary parameters for mining. After discovering the mining payloads, MinerGate can stop the WebSocket connection in the same context by API hooking, so that we can cut off the communications between the miner and the mining pools to forcefully terminate the mining activities.

4 Evaluation

In our evaluation, we first verify the correctness of MinerGate by testing MinerGate in a real environment. Then, we check MinerGate's performance, and confirm MinerGate introduces relatively low overhead. Last, we verify the accuracy of MinerGate.

Our test environment consists of PCs with different OS (i.e., Windows 10 version 1809, macOS 10.14.4, and Ubuntu 18.04), and smartphones (Nexus 5 with Android 6).

4.1 Dataset

There are currently no reliable labeled mining site datasets or WebAssembly/asm.js datasets. To investigate the deployment of WebAssembly in the real world, we deployed a distributed crawler cluster on Azure using Kubernetes to acquire WebAssembly files. The crawlers in the cluster are built upon Chrome and are driven by the "stub.js" described in Sect. 3.2. The cluster includes 120 crawler instances running on the top of 15 physical

nodes. We crawled the Alexa top 1 M sites and randomly selected 10 different URLs from each top site for the next level of crawling. For each website, we spend up to 30 s to load the page and close the page after 10 s. If WebAssembly is detected on the page, the page will be closed immediately (Table 3).

Table 3. Summary of our dataset and key findings

Crawling period	Apr. 25, 2019 - May. 13, 2019
# of crawled websites	10.5 M
# of *benign* web pages with WebAssembly/asm.js	5,030
# of *benign* WebAssembly/asm.js from NPM	946
# of *malicious* mining related web pages	4,659

As a result, we visited a total of 10.5 M pages and found 9,689 web pages containing WebAssembly code, which covers 2,657 registered domains (such as bbc.co.uk) and 3,012 FQDNs (such as forums.bbc.co.uk), and 1,118 top sites contain the WebAssembly code in their home page. The top 15 categories of websites that have deployed WebAssembly are shown in Table 4.

Table 4. Top 15 categories of websites which include WebAssembly.

Categories	#
Adult Content	595
News/Weather/Information	410
Blogs	199
Video & Computer Games	137
Streaming Media	105
Technology & Computing	92
Illegal Content	82
File Sharing	76
Television & Video	61
Sports	38
Weapons	36
Movies	32
Message Boards	31
Shopping	31
Arts & Entertainment	31

To build our training dataset of cryptojacking code, we first match the existing blacklist (uBlock [25], NoCoin [24] and CoinBlockerLists [35]) based on the source URL of WebAssembly/asm.js. If the payloads are from the blacklist URLs, we label the sample as malicious. Some previously unknown mining samples were recognized by reverse engineering analysis with JEB decompiler [21]. Through examination, we found 164 benign WebAssembly samples in 3296 pages (1,735 websites), 55 kinds of malicious WebAssembly, and 6 kinds of malicious asm.js samples in 4659 pages (832 websites) for cryptojacking attacks. We also found that there are 25 undetected malicious WebAssembly samples with the help of VirusTotal [29]. It is worth mentioning that many mining service providers will provide a different bootstrap JavaScript to avoid detection each time they are accessed, but the WebAssembly payloads extracted from them are generally the same. This means that we can analyze the key WebAssembly or asm.js to obtain better analysis results. The top 15 categories of websites which bring Crypto-jacking attacks are shown as Table 5.

Table 5. Top 15 categories of websites which include Cryptojacking.

Categories	#
Adult Content	148
Illegal Content	64
News/Weather/Information	61
File Sharing	42
Technology & Computing	36
Sports	29
Television & Video	27
Streaming Media	26
Video & Computer Games	24
Comic Books/Anime/Manga	22
Arts & Entertainment	16
Movies	14
Web Design/HTML	12
Music & Audio	11
Arts & Entertainment	10

To further build a ground-truth set of non-cryptojacking WebAssembly/asm.js samples, we installed all the packages that be tagged as WebAssembly/asm.js from NPM, which is the largest JavaScript software registry. After the installation is complete, we extract the WebAssembly and asm.js files from the installation folder. The projects we collected include various kinds of libraries and applications, such as video coding, data encryption, data processing, web framework, image processing, physics engine, game framework, and so on. We will publish these samples with labels for future research.

4.2 Accuracy

In this section, we examine MinerGate's classification accuracy on the ground-truth training dataset and compare it with other existing detection techniques. In order to accurately measure the performance of the classifier, we ran 10-fold ross-validation on our dataset. As shown in Fig. 5, the complete MinerGate performs with 99% precision, 98% recall and 99% f1-score. We can also see that the accuracy rate has been greatly improved after adding CFG features and cluster analysis.

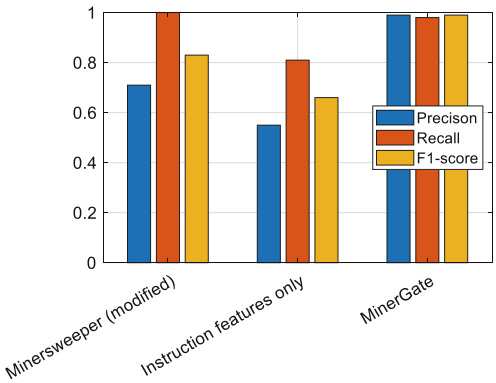

Fig. 5. Results of Cryptojacking discrimination and comparison to other approaches.

In addition to this, the results of Minesweeper [12] are not satisfactory enough. One reason is that they use Chrome's undocumented API (-dump-wasm-module) to dump WebAssembly. But this API cannot dump WebAssembly loaded by instantiateStreaming() or compileStreaming(). To this end, we have implemented a modified version of MineSweeper to take advantage of WebAssembly dumped using our system. As shown in Fig. 5, MineSweeper tends to classify samples as malicious, resulting in lower accuracy and high recall.

4.3 Overhead

First, we test the overhead introduced by MinerGate on benign websites that do not contain WebAssembly/asm.js code. We evaluated the overhead of the system by accessing 1,000 benign web pages and measuring the load time of web pages by enabling/disabling the proxy. The overhead is about 6%, and we found that there is only the overhead of a proxy in this case, because our protected code is only triggered if the WebAssembly is loaded.

Then, we test the overhead on infected websites. We still evaluated the overhead by accessing 1,000 malicious web pages and measuring the load time of web pages by enabling/disabling the proxy. We do not prohibit the execution of the mining program during the overhead evaluation, as this behavior itself will speed up the access of the web page. The overhead is less than 9%, the extra overhead here is mainly from the transmission of WebAssembly.

The transparent proxy itself has no complicated operations. It simply inserts our protection code in the response after the browser makes the request. This is different from the instrumentation in the general sense. Therefore, the transparent proxy's overhead is less than 9% in our evaluation. Since each module of MinerGate is independent, it can be deployed in a distributed manner. To be noticed, both the code injection module and the malicious code analysis module can be independently deployed on multiple machines, so the impact of multiple devices on performance is limited.

Since we have considered performance issues when considering hooks, all code that involves external calls is asynchronous, and only minor performance impacts occur when the program calls a function that is hooked. So overall, our instrumentation will not affect the efficiency of JavaScript. But the time at which the gateway analyzes the code is still important because malicious code can consume a lot of power or block the execution of necessary transactions during the analysis. For background analysis, we plot Fig. 6 which shows the time needed to process different sizes of WebAssembly files.

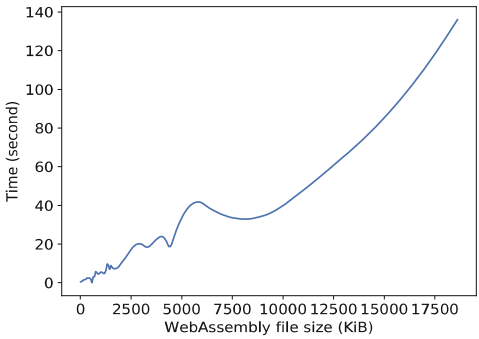

Fig. 6. The time for processing different sizes of WebAssembly in the background.

5 Related Work

Until now, there is no practical generic defense solution against Web-based cryptojacking attacks. One of the limitations of existing methods is that the semantic model of the mining payload is not efficient enough to distinguish between malicious mining applications and benign applications. More importantly, there is currently no non-intrusive defense solution, and all existing work requires modifications to the browser and even the operating system. In this paper, we first apply the CFG (control flow graph), CG (call graph) features to the malicious WebAssembly/asm.js classification, which reviews the problem from another perspective. Since the payload analysis is static, the MinerGate provides a lightweight defense and requires no browser modification by deploying the system to the gateway. The results of comparison with other existing related works are shown in the Table 6.

Table 6. Comparison with other related works.

Name	Scalable (No browser modification)	JavaScript obfuscation resistance	Security enforcement	Low overhead	Low false positives	Used features
MineSweeper [12]	×	√	×	√	×	CPU, WebAssembly Instructions, etc.
SEISMIC [32]	×	√	×	×	×	WebAssembly Instructions, etc.
BMDetector [17]	×	×	×	√	×	JavaScript heap and stack info, etc.
Outguard [12]	×	√	×	√	×	JavaScript loading, etc.
CMTracker [10]	×	√	×	√	×	JavaScript stack info, etc.
MinerGate	√	√	√	√	√	CG, CFG, WebAssembly instructions

Blacklist or Keyword-Based Methods. Some dedicated extensions [24, 25], browsers [2, 22] provide blacklists and keywords to alleviate cryptojacking by manually running honeypot [23] and collecting URLs on reports to expand the list. However, the updates of blacklists and keywords are hard to keep up with the iterative steps of malicious code, which makes the defense always behind the attack.

Instruction Features Based Methods. In the work of Konoth et al. [13], they use static analysis to count the number of cryptographic instructions (`i32.add, i32.and, i32.shl, i32.shr_u, i32.xor`) and loops to detect CryptoNight algorithm. The work of Wang et al. [32] is similar, but the number of instructions is calculated by dynamic instrumentation. However, these cryptographic instructions also exist in many benign applications, such as data encryption, image processing, video encoding, game engines and so on, which will make it difficult to classify these samples accurately.

Stack Dump-Based Methods. The critical observation of stack dump-based methods is that cryptocurrency miners run mining workloads with repeated patterns. In the work of Hong et al. [10], shows that a regular web page rarely repeats the same calling stack for more than 5.60% of the execution time. However, such performance profile requires modifications to the browser kernel, which makes it impractical. In the work of Liu [17], they extract string features from heap and stack snapshot and use RNN to detect the mining programs. This type of method built on strings or keywords is unreliable and can be easily bypassed by JavaScript code.

6 Conclusions and Future Work

With a deeper understanding of the semantics of WebAssembly/asm.js, we designed a novel generic defense solution MinerGate against Web-based cryptojacking attacks. By decentralizing computing tasks to the gateway, we implemented a common protection scheme with the lowest overhead in known scenarios, which does not require modification of the browser. Through data-driven feature selection, we not only further demonstrate the effectiveness of instruction-level features but also indicate the excellent performance of CFG features in malicious code detection.

The main limitations exist in two aspects. First of all, considering that JavaScript is a highly dynamic and continuously evolving language, it is difficult to prove that the APIs we intercept is always complete. On the other hand, since this work uses a machine learning-based method, there is the possibility of constructing adversary samples, and we may need extra work to defend against it.

Acknowledgments. This project is supported by National Natural Science Foundation of China (No. 61972224).

References

1. Ana, A.: Report: Some crypto mining apps remain in Google play store despite recent ban (2018). https://cointelegraph.com/news/report-some-crypto-mining-apps-remain-in-google-play-store-despite-recent-ban. Accessed 21 Nov 2019
2. Andrea, M.: Firefox: implement cryptomining URL-classifier (2019). https://hg.mozilla.org/mozilla-central/rev/d503dc3fd033. Accessed 01 May 2020
3. Barrat, A., Barthelemy, M., Pastor-Satorras, R., Vespignani, A.: The architecture of complex weighted networks. Proc. Natl. Acad. Sci. **101**(11), 3747–3752 (2004)
4. Catalin, C.: New browser attack lets hackers run bad code even after users leave a web page (2019). https://www.zdnet.com/article/new-browser-attack-lets-hackers-run-bad-code-even-after-users-leave-a-web-page/. Accessed 01 May 2020
5. Daniel, P.: 8 illicit crypto-mining windows apps removed from microsoft store (2019). https://www.coindesk.com/8-illicit-crypto-mining-windows-apps-removed-from-microsoft-store. Accessed 01 May 2020
6. David, H., Luke, W., Alon, Z.: asm.js working draft (2018). http://asmjs.org/spec/latest/
7. Eyal, I., Sirer, E.G.: Majority is not enough: Bitcoin mining is vulnerable. Commun. ACM **61**(7), 95–102 (2018)
8. Pearson, K.: X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. London Edinburgh Dublin Philos. Mag. J. Sci. **50**(302), 157–175 (1900). https://doi.org/10.1080/14786440009463897
9. Group, W.C.: Webassembly specification (2018). https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. Accessed 01 May 2020
10. Hong, G., et al.: 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, CCS 2018, pp. 1701–1713. ACM, New York (2018). https://doi.org/10.1145/3243734.3243840. http://doi.acm.org/10.1145/3243734.3243840

11. npm Inc.: npm — the heart of the modern development community (2018). https://www.npmjs.com/. Accessed 01 May 2020
12. Kharraz, A., et al.: Outguard: detecting in-browser covert cryptocurrency mining in the wild (2019)
13. Konoth, R.K., et al.: 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, pp. 1714–1730. ACM (2018)
14. Kunegis, J.: KONECT – the Koblenz network collection. In: Proceedings of International Conference on World Wide Web Companion, pp. 1343–1350 (2013). http://dl.acm.org/citation.cfm?id=2488173
15. Newman, L.H.: Hack brief: hackers enlisted Tesla's public cloud to mine cryptocurrencies (2018). https://www.wired.com/story/cryptojacking-tesla-amazon-cloud/. Accessed 01 May 2020
16. Lindsey, O.: Cryptojacking attack found on los angeles times website (2018). https://threatpost.com/cryptojacking-attack-found-on-los-angeles-times-website/130041/. Accessed 01 May 2020
17. Liu, J., Zhao, Z., Cui, X., Wang, Z., Liu, Q.: A novel approach for detecting browser-based silent miner. In: Proceedings - 2018 IEEE 3rd International Conference on Data Science in Cyberspace, DSC 2018, Guangzhou, China, pp. 490–497. IEEE, June 2018. https://doi.org/10.1109/DSC.2018.00079. https://ieeexplore.ieee.org/document/8411900/
18. Magazinius, J., Phung, P.H., Sands, D.: Safe wrappers and sane policies for self protecting JavaScript. In: Aura, T., Järvinen, K., Nyberg, K. (eds.) NordSec 2010. LNCS, vol. 7127, pp. 239–255. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27937-9_17
19. Neil, B.: Kaspersky reports 13 million cryptojacking attempts this year, January 2018. https://www.cryptolinenews.com/2018/12/13-million-cryptojacking-says-kaspersky/. Accessed 01 May 2020
20. Newman, L.H.: Now cryptojacking threatens critical infrastructure too (2018). https://www.wired.com/story/cryptojacking-critical-infrastructure/. Accessed 01 May 2020
21. Nicolas, F., Joan, C., Cedric, L.: Jeb decompiler (2018). https://www.pnfsoftware.com/jeb/. Accessed 01 May 2020
22. Opera: Cryptojacking test (2018). https://cryptojackingtest.com/. Accessed 01 May 2020
23. Prakash: Drmine (2018). https://github.com/1lastBr3ath/drmine/. Accessed 01 May 2020
24. Rafael, K.: Nocoin (2018). https://github.com/keraf/NoCoin/. Accessed 01 May 2020
25. Raymond, H.: ublock (2018). https://github.com/gorhill/uBlock/. Accessed 01 May 2020
26. Rossberg, A., et al.: Bringing the web up to speed with webassembly. Commun. ACM **61**(12), 107–115 (2018). https://doi.org/10.1145/3282510
27. Seigen, Max, J., Tuomo, N., Neocortex, Antonio, M.J.: Cryptonight hash function (2013). https://cryptonote.org/cns/cns008.txt. Accessed 01 May 2020
28. Simoudis, E., Han, J., Fayyad, U.M. (eds.): Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD 1996), Portland, Oregon, USA. AAAI Press (1996). http://www.aaai.org/Library/KDD/kdd96contents.php
29. VirusTotal: Virustotal (2018). https://www.virustotal.com/. Accessed 01 May 2020
30. W3C: Web workers (2015). https://www.w3.org/TR/workers/. Accessed 01 May 2020
31. W3C: The websocket api. https://www.w3.org/TR/websockets/. Accessed 01 May 2020
32. Wang, W., Ferrell, B., Xu, X., Hamlen, K.W., Hao, S.: SEISMIC: SEcure in-lined script monitors for interrupting cryptojacks. In: Lopez, J., Zhou, J., Soriano, M. (eds.) ESORICS 2018. LNCS, vol. 11099, pp. 122–142. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98989-1_7
33. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**(6684), 440 (1998)

34. Wikipedia: Cryptocurrency (2018). https://en.wikipedia.org/wiki/Cryptocurrency. Accessed 01 May 2020
35. ZeroDot1: Coinblockerlists (2018). https://zerodot1.gitlab.io/CoinBlockerListsWeb/index.htm. Accessed 01 May 2020