# POSTER: Detecting WebAssembly-based Cryptocurrency Mining

Weikang Bian
Chinese University of Hong Kong
wkbian@cse.cuhk.edu.hk

Wei Meng
Chinese University of Hong Kong
wei@cse.cuhk.edu.hk

Yi Wang
Southern University of Science and Technology
wy@ieee.org

## ABSTRACT

In-browser cryptojacking is an emerging threat to web users. The attackers can abuse the users' computation resources to perform cryptocurrency mining without obtaining their consent. Moreover, the new web feature –WebAssembly (Wasm)– enables efficient in-browser cryptocurrency mining and has been commonly used in mining applications. In this work, we use the dynamic Wasm instruction execution trace to model the behavior of different Wasm applications. We observe that the cryptocurrency mining Wasm programs exhibit very different execution traces from other Wasm programs (*e.g.*, games). Based on our findings, we propose a novel browser-based methodology to detect in-browser Wasm-based cryptojacking.

## CCS CONCEPTS

• **Security and privacy** → **Browser security**; *Malware and its mitigation.*

## KEYWORDS

WebAssembly; Cryptojacking; Cryptocurrency mining

**ACM Reference Format:**
Weikang Bian, Wei Meng, and Yi Wang. 2019. POSTER: Detecting WebAssembly-based Cryptocurrency Mining. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3319535.3363287

## 1 INTRODUCTION

In-browser cryptocurrency mining draws increasing attention from cybercriminals for the rising value of cryptocurrencies [7]. According to a report from Cyber Threat Alliance in 2018 [1], there had been a 459 percent increase in illicit cryptocurrency mining malware detection since 2017. Several studies have shown that website owners deploy cryptocurrency mining code for extra profit [3, 9, 11]. Many websites do not obtain any consent from the users before running the mining code. Such malicious practice is called cryptojacking, which is often implemented using WebAssembly (Wasm) for its great computation efficiency. Recently, researchers have started to investigate cryptojacking in the wild [2, 8] and proposed several detection strategies [4–6, 10].


*CCS '19, November 11–15, 2019, London, United Kingdom*

ACM ISBN 978-1-4503-6747-9/19/11.
https://doi.org/10.1145/3319535.3363287

In this work, we aim to detect in-browser Wasm-based cryptojacking, where a website automatically starts the cryptocurrency mining procedure after it is loaded. We assume that a remote attacker controls the website that a victim user may visit. Thus the attacker can execute arbitrary JavaScript code and Wasm code in the victim user's browser. The attacker uses the JavaScript code to manage the in-browser mining tasks and communicate with a remote server to submit the mining results and request for new tasks. The Wasm code is mainly used for cryptocurrency mining which is computationally intensive.

We further assume that the attacker may leverage various techniques to evade the existing detection methods. One of the most commonly used techniques is code obfuscation. In other words, simply computing a code signature is not reliable to identify a mining script. Methods that use the function names to build a script signature are also likely to be bypassed. Furthermore, the attacker may apply throttling mechanism on the number of mining threads or the mining speed, to circumvent detection methods that monitor CPU usage. Finally, the attacker may communicate with his/her own custom servers instead of the servers of publicly known mining service providers to bypass detection methods which rely on matching the hostnames of WebSocket or XHR communication targets in some community-maintained public blacklists (*e.g.*, NoCoin).

To reliably identify Wasm-based cryptojacking code, we have to find other behavioral/semantic features that can well represent the nature of the mining activities. In this work, we study and propose to use the subsequences of Wasm instruction execution trace as a signature to identify Wasm programs running a known cryptocurrency mining algorithm.

## 2 METHODOLOGY

In this section, we present a new approach to dynamically detecting in-browser cryptojacking behavior. We focus on the websites that use Wasm to perform cryptocurrency mining on the user's device, because Wasm offers great performance and is platform independent.

We first describe how we model the behavior of Wasm programs using the subsequences of Wasm instruction execution trace (§2.1). Then we discuss how we leverage the model to detect in-browser Wasm-based cryptojacking (§2.2).

### 2.1 Modeling Wasm Program Behavior

In this section, we investigate the feasibility of using the *Wasm instruction execution trace* to model the behavior of Wasm programs, and to differentiate a cryptocurrency mining Wasm program from other benign Wasm programs. Intuitively, one can use the full trace to model a program. However, it is not practical because a program may not always generate the same trace in multiple runs. Therefore,

| Rank | CryptoLoot | | Tanks | |
|---|---|---|---|---|
| | Inst. Sequences | Percentage | Inst. Sequences | Percentage |
| 1 | add xor shl add xor | 11.20% | shl shr_s eqz shl shr_s | 4.90% |
| 2 | shl add xor shl add | 11.20% | shl shr_s add shl shr_s | 3.97% |
| 3 | xor shl add xor xor | 5.60% | shr_s add shl shr_s ne | 3.91% |
| 4 | xor shl add xor shl | 5.60% | shr_s eqz shl shr_s add | 3.83% |
| 5 | add shl add xor shl | 4.56% | eqz shl shr_s add shl | 3.83% |
| 6 | shl add shl add xor | 4.56% | add shl shr_s eqz shl | 3.83% |
| 7 | xor xor add shl add | 3.11% | add shl shr_s ne or | 3.83% |
| 8 | shl add xor xor add | 3.11% | add add add add add | 3.08% |
| 9 | add xor xor add shl | 3.11% | shl shr_s shl shr_s eq | 2.45% |
| 10 | add shl add shl add | 2.07% | add gt_u add gt_u or | 2.43% |

**Table 1: The frequency distribution of top instructions of CryptoLoot and Tanks.**

we try to determine if the subsequences of the trace can be used as a distinguishing feature to model the program behavior.

We collect and compare the Wasm instruction execution traces of several different Wasm-based web applications with a custom Chromium browser. In particular, we set a small sliding window/group of consecutive numeric instructions[1]. We denote $N_{inst}$ as the size of the sliding window. We terminate a sliding window at a control instruction to ensure that the instructions within the window are always executed consecutively.

We present the frequency distribution of the top 10 subsequences (groups) of https://crypto-loot.org/ – a popular cryptocurrency mining service provider, and https://webassembly.org/demo/Tanks/ – a web tank battling game developed in Wasm and Unity WebGL, in Table 1. Specifically, we set $N_{inst}$ to 5. It is obvious that the top instruction subsequences of the two applications are quite distinct. We also draw the accumulate frequency distribution of the top 10 instruction groups over time of the two applications in Figure 1 and Figure 2, respectively. The Y axis represents the accumulate percentage of each group of Wasm instructions out of the 10 groups. We vary the size of the instruction trace in the X axis to demonstrate how the distribution may change over time. On the one hand, the distribution of `Tanks` does change over time. On the other hand, as expected, we observe a very stable distribution for `CryptoLoot`, which had to repetitively calculate the proof of work for cryptocurrency mining. This suggests that the distribution of the top groups of $N_{inst}$ instructions is potentially a good feature to represent the runtime behavior of Wasm programs and to differentiate between different Wasm programs.

## 2.2 Detecting Cryptocurrency Mining Programs

As we observed in §2.1, the frequency distribution of the top instruction groups of CryptoLoot is very stable over time. We thus check in runtime if another Wasm program exhibits a very similar distribution. Nevertheless, as is evident in Figure 1, the accumulative distribution is not always constant over time.

To capture potential variation of the mining activities and to enable fast detection, we calculate the frequency distribution of the top $N_{group}$ groups of $N_{inst}$ instructions of a mining sample

[1]http://webassembly.github.io/spec/core/binary/instructions.html

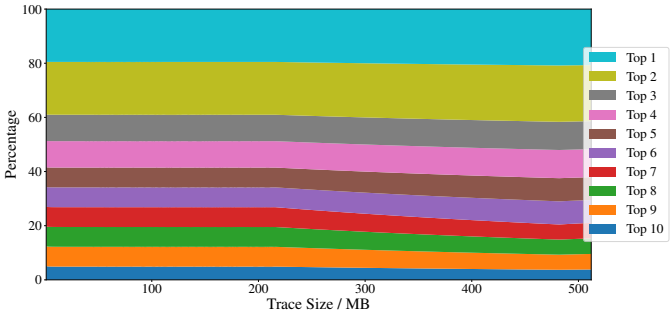


**Figure 1: The frequency distribution of top instructions of CryptoLoot.**

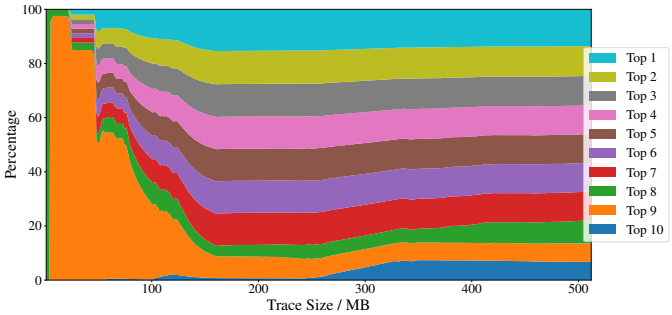


**Figure 2: The frequency distribution of top instructions of Tanks.**

program's trace in many different sequence intervals. The distribution in each interval indicates the temporal mining speed. We then compute a baseline average distribution that represents the average mining speed of the sample mining program.

At runtime, we would compute in the same interval, which we call the *detection interval*, a distribution of the same top $N_{group}$ groups of instructions of a Wasm thread. We then calculate its cosine similarity score – *score* – with the baseline distribution. If the score is close to 1, it is very likely that the thread is executing similar code in that interval. We use a parameter $T_c$ as the threshold for comparing the similarity.

It is possible that a non-mining thread executes the same code very shortly in the detection interval and then becomes inactive. Therefore, instead of using the percentage to calculate the distribution, we use the number of executed instructions divided by the detection interval to represent the mining speed. Further, we would compare the calculated mining speed – *speed* – with that of the baseline speed – $S$. We would make a *negative* decision if the mining speed is much smaller than the baseline speed. Specifically, we obtain the standard deviation $\sigma$ when calculating the baseline average speed. We use a parameter $T_\sigma$ to control the absolute detection distance from the baseline speed. A positive decision is made if the following inequality is true. We will discuss how to determine the parameters $N_{group}$ and $N_{inst}$ next.

$$score \geq T_c \wedge s \geq S - T_\sigma \sigma \tag{1}$$

*2.2.1 Studying parameters $N_{group}$ and $N_{inst}$.* We experiment with different settings of $N_{group}$ and $N_{inst}$ to study the overhead and accuracy of our method on a small-scale training set. For simplicity, we use another mining program as the positive case and the Tank game as the negative case.

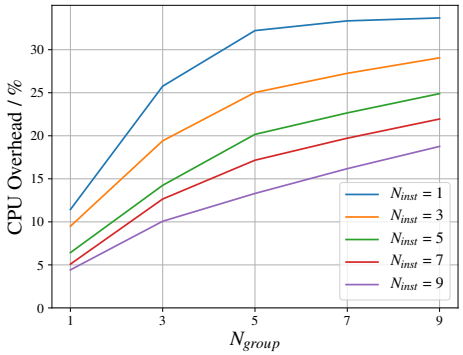


**Figure 3: The overhead for the positive case.**

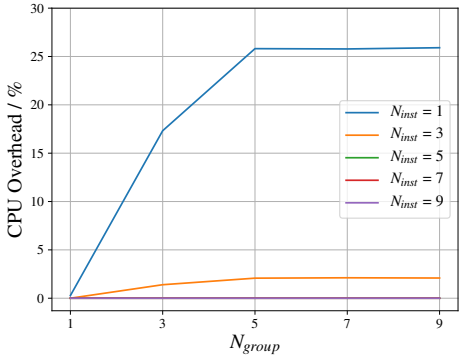


**Figure 4: The overhead for the negative case.**

To count an occurrence of the top $N_{inst}$ instructions, we need to insert four extra profiling instructions to the Wasm. We enable the Linux-perf feature to dump the per-thread executed instruction numbers to measure the overhead, which is the ratio of additional instructions executed in a fixed CPU time. As is shown in Figure 3, the overhead decreases significantly as $N_{inst}$ increases or $N_{group}$ decreases for the positive case. For the negative case (Figure 4), the overhead is very limited if we select a larger $N_{inst}$.

**Table 2: The similarity score of the positive case.**

| $score$ $N_{group}$ / $N_{inst}$ | 1 | 3 | 5 | 7 | 9 |
|---|---|---|---|---|---|
| 1 | 0.930 | 0.947 | 0.943 | 0.946 | 0.930 |
| 3 | 0.930 | 0.928 | 0.941 | 0.945 | 0.930 |
| 5 | 0.935 | 0.929 | 0.929 | 0.947 | 0.933 |
| 7 | 0.932 | 0.948 | 0.937 | 0.933 | 0.934 |
| 9 | 0.934 | 0.939 | 0.948 | 0.928 | 0.935 |

**Table 3: The similarity score of the negative case.**

| $score$ $N_{group}$ / $N_{inst}$ | 1 | 3 | 5 | 7 | 9 |
|---|---|---|---|---|---|
| 1 | 1.000 | 0.716 | 0.760 | 0.754 | 0.753 |
| 3 | 1.000 | 0.519 | 0.558 | 0.542 | 0.534 |
| 5 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| 7 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| 9 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |

We present the similarity scores of the positive case and negative case in Table 2 and Table 3, respectively. We found that as long as both $N_{inst}$ and $N_{group}$ are larger than 3, we could get relatively good detection performance in this small-scale experiment. Thus, we think the method can potentially perform well in detecting Wasm programs running a known cryptocurrency mining algorithm.

## 3 EVALUATION

We performed a preliminary experiment on Alexa top 100K websites using our modified browser in June 2019. We found 87 websites that included as least one Wasm script. Four websites were detected by our method as cryptojacking websites, which leveraged the same mining algorithm as CryptoLoot. We manually checked all the 87 Wasm websites and confirmed our detection was accurate. Further, we did not have any false negative websites, *i.e.*, there was no other website running the CryptoLoot mining algorithm. Our results indicate that the subsequences of Wasm instruction execution trace is potentially a reliable feature to identify similar Wasm cryptojacking programs.

## 4 DISCUSSION AND FUTURE WORK

We evaluated the preliminary method of using the subsequences of Wasm instruction execution traces to detect Wasm cryptojacking programs. We have shown that the frequency distribution of the top instruction groups can well represent the intrinsic mining behavior of a Wasm program. However, our approach may not be obfuscation-robust, although Wasm obfuscation is not widely observed yet. For instance, the attacker may inject extra instructions, reverse the order of instructions, or break a basic block into multiple smaller ones to bypass our detection. We aim to improve our methodology against possible Wasm obfuscation attacks in our future work.

## ACKNOWLEDGMENT

The work described in this paper was partly supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK 24209418).

## REFERENCES

[1] 2018. THE ILLICIT CRYPTOCURRENCY MINING THREAT. https://www.cyberthreatalliance.org/wp-content/uploads/2018/09/CTA-Illicit-CryptoMining-Whitepaper.pdf.
[2] Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. 2018. A first look at browser-based Cryptojacking. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 58–66.
[3] Guardian. 2018. https://www.theguardian.com/technology/2017/sep/27/pirate-bay-showtime-ads-websites-electricity-pay-bills-cryptocurrency-bitcoin.
[4] Geng Hong, Zhemin Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Haixin Duan. 2018. How you get shot in the back: A systematical study about cryptojacking in the real world. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.
[5] Amin Kharraz, Zane Ma, Paul Murley, Charles Lever, Joshua Mason, Andrew Miller, Nikita Borisov, Manos Antonakakis, and Michael Bailey. 2019. Outguard: Detecting In-Browser Covert Cryptocurrency Mining in the Wild. In *Proceedings of the The Web Conference (WWW)*. San Francisco, CA.
[6] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. 2018. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.
[7] Hon Lau. 2017. Browser-based cryptocurrency mining makes unexpected return from the dead. *Sympantec Threat Intelligence* (2017).
[8] Jan Rüth, Torsten Zimmermann, Konrad Wolsing, and Oliver Hohlfeld. 2018. Digging into browser-based crypto mining. In *Proceedings of the Internet Measurement Conference 2018*. ACM, 70–76.
[9] TrendMicro. 2018. https://blog.trendmicro.com/trendlabs-security-intelligence/malvertising-campaignabuses-googles-doubleclick-to-deliver-cryptocurrency-miners/.
[10] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W Hamlen, and Shuang Hao. 2018. Seismic: Secure in-lined script monitors for interrupting cryptojacks. In *European Symposium on Research in Computer Security*. Springer, 122–142.
[11] Mark Ward. 2018. http://www.bbc.com/news/technology-41518351.