WebAssembly Cryptojacking Attack

An Attack on a Defense

Kieran Arora
College of Science and Engineering
University of Minnesota, Twin Cities
Minneapolis, MN, US
arora119@umn.edu

Jonathan Dekraker
College of Science and Engineering
University of Minnesota, Twin Cities
Minneapolis, MN, US
dekra008@umn.edu

Sean Mangan
College of Science and Engineering
University of Minnesota, Twin Cities
Minneapolis, MN, US
manga093@umn.edu

Aaron Marsland
College of Science and Engineering
University of Minnesota, Twin Cities
Minneapolis, MN, US
marsl003@umn.edu

Madelyn Ogorek
College of Science and Engineering
University of Minnesota, Twin Cities
Minneapolis, MN, US
ogore014@umn.edu

ABSTRACT

By getting internet users to visit and spend time on their website, attackers can utilize the computation power of devices across the globe in order to mine for cryptocurrency. With WebAssembly, these strategies have become highly efficient at carrying out the attack. Defense techniques have recently emerged that combat these attacks by detection using an in-browser execution trace. In our work, we aim to demonstrate that these existing defenses are easily fooled by multiple obfuscation techniques and therefore they are insufficient for this new kind of attack.

CCS CONCEPTS

• **Security and Privacy → Browser Security**; *Attack on Malware mitigation techniques*

KEYWORDS

WebAssembly; Cryptojacking; Cryptocurrency mining

ACM Reference format:

Kieran Arora, Jonathan Dekraker, Aaron Marsland, Sean Mangan, and Madelyn Ogorek. 2021. Web-Assembly Crypto Mining Attack: An Attack on a Defense. *October - December 2021, The University of Minnesota, Twin Cities. Minneapolis, MN, USA, 2 pages.*

1 INTRODUCTION

With cryptocurrency becoming an increasingly important part of the global economy, cybercriminals have been looking for new ways to get their hands on resources for producing said currency. Crypto mining operations involving hundreds of computers and requiring several thousands of kilowatt-hours have been set up to solve the complex cryptographic problems that produce cryptocurrency.

A new form of crypto mining has started gathering the attention of some cyber criminals. By attracting victims to their websites, attackers utilize the processing power of the victims' computers to mine for cryptocurrency. "Cryptojacking" is recognized by Interpol as the practice of "secretly" using a "victim's computing power to generate cryptocurrency" [1] Interpol has clarified that cryptojacking can occur both when a victim visits an attacker-controlled website as well as when an unsuspecting victim clicks on an infected link (possibly sent by the attacker through email). Although it does not entail a theft of funds or personal information, cryptojacking is recognized as a cybercrime because computing resources are taken without the consent of the victim. In a study conducted at Queen's University, researchers found that in 2018, cryptojacking brought in a total of $1.8 million [2].

The implications of this new threat are varied. First and foremost, this process draws heavily on one's computational resources and could result in heavy delays for the victim. In addition, it was found by cybersecurity analysts at Kaspersky lab that cryptojacking can have physical effects on the victim's device [3]. Specifically, these researchers were investigating a process that targeted Android devices to mine Monero. The researchers discovered that two days of mining caused the phone's battery to expand enough that it warped the exterior of the phone. Thus, cryptojacking is a legitimate threat to victims with significant potential consequences, and it is reasonable to invest resources into its mitigation.

2 BACKGROUND

2.1 Cryptojacking Reverberations

Cryptojacking is alluring to cryptocurrency miners due to the absence of need for extensive personal hardware and processing power. The enormous pool of internet users allows cryptojackers to increase their chances of successfully mining cryptocurrency. Some of the most advanced cryptojacking operations have the ability to hide their functionality from the user. They can regulate their CPU usage and work in the background on an open browser tab. To circumvent the possibility of a curious user noticing a suspicious drop in battery level, some implementations can wait until a host is connected to a power source to scale-up their computational demands.

In an article entitled "The Crypto Grey Zone," Digital Trust Leader Matthew White contends that cryptojacking is far more prevalent on the internet than previously thought [4]. White clarifies that the practice is being done by both "legitimate and malicious" actors alike. With a rise in the use of ad blockers, website owners can no longer count on a steady stream of revenue from advertisements, and some have started offering their patrons a choice: Visitors can either disable their ad blocker or agree to donate their spare processing power to a cryptocurrency mining operation. With the amount of power required to successfully mine crypto on the rise, it is likely that the presence of cryptojacking on the internet will grow. This growth underlines the current need for adequate defense strategies.

2.2 Existing Mitigation Techniques

Multiple strategies have been proposed to mitigate the effects and effectiveness of cryptojacking. By building on the capabilities of modern ad blocking products, in-browser extensions have been developed to prevent such nefarious scripts from executing. In addition, security experts have also implemented deny-list approaches to prevent visits to any websites that are found to be engaging in cryptojacking, though these are easily bypassed by making slight changes in the cryptocurrency mining algorithms. More recently, detection mechanisms have been proposed that function by examining the most common subsequences of a program's execution trace and comparing to known mining scripts. The following section will provide an analysis of the current landscape of mitigation techniques and explore specific examples of the aforementioned strategies.

2.2.1 Anti-Cryptojacking Browser Extension

Several in-browser extensions exist to either detect cryptocurrency mining within a browser, prevent it from occurring, or both. One of the most popular anti-cryptojacking browser extensions is MinerBlock, which can be found at xd4rker's repository on GitHub [5]. The program works by combining two common approaches available for detecting and preventing cryptojacking. The extension includes the traditional method used by several ad-blockers of checking scripts against a compiled list of blacklisted scripts and preventing them from executing if there is a match. To make the service more effective, the software also aims to detect and kill "potential mining behavior" in programs while the scripts are loading.

For even more protection, internet users can choose to install a browser extension such as ScriptSafe or NoScript, both of which specifically target and prevent executable code from running unless it is from an approved source. These types of software are proven to be effective at mitigating the risk of Spectre and Meltdown attacks that can be run through JavaScript and other executable code [6].

2.2.2 Wasm-Based Exploits and their Existing Detection Mechanisms

One of the primary reasons for the creation and preeminence of cryptojacking was the advent of WebAssembly (Wasm) technologies. Wasm is a "binary instruction format," whose central purpose is to expedite interactions between executable software programs and the hosting environment [7]. Wasm is quickly being embraced by the tech community and is currently functional in Google Chrome, Microsoft Edge, Safari, and Mozilla Firefox. In cooperation with JavaScript, Wasm can match or exceed the efficiency of native-machine code. Because of these qualities, WebAssembly is highly attractive to those looking to mine cryptocurrency through web browsers. A study done in 2019 found that a staggering 1 out of every 600 of Alexa's top 1 million websites engage in cryptojacking [8]. After writing the code that is able to perform cryptographic hashing functions (utilizing the power of the host machine), the WebAssembly modules made by these actors can then be compiled and optimized using one of several JavaScript APIs (e.g., fetch()).

There are a handful of existing defense techniques that are used to combat WebAssembly-based cryptojacking. There exist some browser extensions that specifically target Wasm (see W. Bian et al. MineThrottle) [9]. These extensions generally work by dynamically analyzing specific Wasm code signatures and comparing them to known cryptojacking modules. By monitoring a program's network traffic and system call usage, such techniques can detect mining behavior.

There also exist systems that target Wasm-based exploits that do not utilize web browsers. These exploits target the host directly, possibly due to their impressive processing capabilities. In an effort to catch these types of malware, multiple techniques have been proposed. Several existing solutions work by simply examining the fixed aspects of a particular program (e.g., API and function call information). One of the more impressive defenses uses machine learning methods to train a system to detect cryptojacking both statically and dynamically [10]. Besides the in-browser and host-based mechanisms, Neto et al. propose a solution that aims to protect all the devices connected to a particular network, by using a machine learning model to monitor the network traffic [11].

Of particular interest to this paper is a technique proposed by W. Bian et al. in 2019. Rather than computing a simple static code signature which can be easily changed by obfuscating code or modifying names of functions or variables, W. Bian et al. (2019) use the most common subsequences of the instruction execution traces generated by a suspicious script to dynamically detect in-browser Wasm cryptojacking by comparing them to the common subsequences of instructions that are found in known miners [12]. However, this approach can still be foiled by clever obfuscation techniques. The remainder of this paper is devoted to replicating the defense proposed by W. Bian et al. (2019) and demonstrating how obfuscation can be used to bypass instruction execution trace mechanisms for the detection of cryptocurrency mining.

3 METHODOLOGY

3.1 Replication of Initial Mining Script

A cryptojacking script was created that used WebAssembly to perform cryptographic hashing functions to mine Monero cryptocurrency using the RandomX algorithm. As cryptojacking has become increasingly prevalent, more "mining providers" have launched. These providers supply mining scripts that can be easily implemented in a web application, often with fewer than 5 lines of JavaScript. Webminepool.com is one of such providers, and our initial replication of the attack uses a modified version of the code supplied by Webminepool.com. However, it should be noted that regardless of the "mining provider" that a web application uses for this attack, the underlying algorithm will always be RandomX provided that the targeted coin is Monero. Therefore, without loss of generality, applying the modified script from Webminepool.com can represent any RandomX based WebAssembly mining script.

The provided mining script was originally modified in order to break blacklist-based detection methods. Most browsers will block attempts to load JavaScript from "mining providers." This modification is to simply replace any request to load JavaScript with the source code itself. This change will ensure that any requests to "mining providers" are not necessary for full functionality of the miner itself, and furthermore, will not elicit any warning messages to otherwise unsuspecting users.

With the first line of defense from cryptojacking already bypassed, more modifications were necessary to prevent detection from more robust methods. Two different implementations were constructed to represent two hypothetical attack approaches: a browser-based method and a NodeJS script method. Both approaches are plausible entry points for this attack. The NodeJS approach represents hidden crypto mining functionality from an unverified imported module. The browser approach, which we hosted at not-a-cryptojacker.com (a somewhat sarcastic title, as we made sure to obtain explicit consent from the visitor for mining cryptocurrency using their resources), represents the typical attack, wherein the attacker uses the victim's resources via use of an attacker-controlled website or web application. Despite these differing approaches, the detection method is essentially the same for both, as in either case the executed WebAssembly must be interpreted by Google's V8 engine.

3.2 Replication of Execution Trace Detection Mechanism

Though NodeJS and browser-based attacks can easily bypass blacklist-based detection mechanisms against cryptojacking, attempts have been made to detect more cryptojacking scripts by analyzing the execution trace of their WebAssembly function calls. Each function call generates a unique trace of instructions that are executed over its runtime, and close examination of this execution trace can offer insight into the functionality of the WebAssembly in question. W. Bian et al. (Detecting WeAssembly-Based Cryptocurrency Mining) proposed a methodology for the detection of cryptocurrency mining scripts by looking at the top subsequences of instructions in the WebAssembly execution trace [12]. We aim to create an implementation that replicates this defense. We will subsequently contribute how to bypass this detection through obfuscation of the execution trace.

3.2.1 Generation of Execution Trace Logs

Javascript, and consequently WebAssembly, is interpreted and compiled via Google's V8 engine. In order to generate this execution trace, this WebAssembly engine must be modified with this additional functionality. To modify the WebAssembly engine, we used a taint tracking engine for interpreted WebAssembly called TaintAssembly, developed at Harvard University in 2018 [13]. The underlying mechanism is relatively simple: in the construction of the WebAssembly abstract syntax tree, assign each node (or instruction) a particular character or string and upon execution of that node, append that character or string to a logging file. Finally, upon return of a WebAssembly instruction, a delimiter can be appended to the resulting log. Furthermore, since this implementation is based on modifying Google's V8 engine, this approach can be applied in any use of said engine (NodeJS, Chrome, Edge, etc.).

With the execution trace functionality implemented in a modified V8 JavaScript engine, the trace of known cryptojackers can be generated either by running known cryptojacking scripts through a modified NodeJS or by visiting known cryptojacking websites using a V8 based browser. Due to the great compilation time to build browsers such as chromium, further cryptojacking detection via execution trace analysis will be performed using a modified NodeJS build.

3.2.2 Implementation of Proposed Detection Mechanism

Execution traces as well as instruction-based traces of a called WebAssembly function offer great insight to the overall design and use of the function. Although each instruction is recorded, execution traces are generally very obscure and difficult to

analyze precisely what the calling function is doing. In the case of WebAssembly based cryptojacking, the use or purpose of any given function call is not readily decipherable, even upon manual inspection. This is not assisted by the fact that the same function call is not guaranteed to generate the same trace. Due to conditionals and error handling, different instructions may be executed depending on the initial conditions.

Given these issues, it is necessary to analyze execution logs in a more general sense. Specifically, analysis of a given trace can be assisted by breaking down the entirety of the trace into its most likely sequences of instructions. The idea behind this approach, first proposed by W. Bian et al. (2019), is to break the execution trace into a sliding window of subsequences consisting of 5 instructions each and examine the 10 most common of these subsequences [12]. This is a particularly fruitful endeavor in the analysis of traces generated by known cryptojackers, as these top subsequences are typically uniform between runs due to the sheer number of calls to a WebAssembly based hash function.

After generating the execution log of a known cryptojacker and extracting the top series of instructions, it becomes a relatively mundane process of determining if the execution trace of a given script is a cryptojacker, and consequently malicious. There are two possible approaches, which we will call the *naïve approach* and the *targeted approach*. The naïve approach simply compares the 10 most common subsequences of 5 instructions from the script under test to the 10 most common subsequences of 5 instructions from the known cryptojacking script. If all the instruction sequences in the top series are equivalent to the reference trace derived from a known cryptojacker, then the trace in question was likely generated from a cryptojacker as well. In the case that these top instructions are not equivalent, it is likely that the program that generated the trace is not mining Monero.

However, minimizing the false positive rate is of utmost importance with the growing ubiquity of WebAssembly. Therefore, this approach can be further refined by defining a *range of acceptability*, resulting in the targeted approach. This would be necessary in the case of a benign trace generating the same top sequences as a known cryptojacker. In this case it would be prudent to examine the proportions of the top instructions themselves. We can call this proportion the *occurrence rate*. More precisely, the occurrence rate is defined as the percentage of total instructions in which that series of instructions occurs. For example, if the series ‘ADD ADD ADD LES SUB’ has an occurrence rate of 0.035, then ‘ADD ADD ADD LES SUB’ will be executed during 3.5% of the total execution of the script. This targeted approach will also examine the occurrence rate in the false positive case where the top series of instructions are the same (i.e., the script fails the naïve approach, and a benign trace is marked as malicious). It is likely that if a benign script generated the same top sequences, the proportion of these top sequences fall out of a given range of acceptability. This range of acceptability can be manually set to minimize false positives. However, despite the growing use of WebAssembly, a defined range of acceptability has not been needed for this use case. The targeted approach thus-so-far seems to be unnecessary as the naïve approach is able to detect current crypto jacking scripts with relative robustness. This is subject to change as WebAssembly based applications become more prevalent. Furthermore, this approach, with few modifications, can be even more robust. Currently the top 10 most occurring subsequences are analyzed, but this figure can be manually adjusted to compare fewer or more series, and consequently take an even more targeted approach to detection.

With all these parameters in mind, a Python script was created to parse the generated execution trace logs and implement both the naïve and targeted approaches to execution trace detection. The script finds the top subsequences of instructions from the log and implements the naïve approach by checking whether they are the same as the top subsequences from a known mining script (from Cryptoloot), If an optional “range of acceptability” command-line argument is included, the script implements the targeted approach by adding an additional step: if the top subsequences are the same, we check if their occurrence rate is within the range of acceptability of each other. However, for the scope of this paper we conducted all tests using the naïve approach, without a range of acceptability input to the script.

3.3 Anti-Detection Modifications

Our approach aims to circumvent execution trace detection mechanisms for Wasm-based cryptojacking by using code obfuscation techniques to reduce the frequency of instructions performing hashes to mine cryptocurrency. This in turn will reduce the similarity between the Wasm instruction trace of our cryptojacking program and the instruction traces of other Wasm-based cryptocurrency mining programs. Cryptojacking detection mechanisms that work by comparison of instruction traces will thus be rendered less effective at detecting our program as a cryptocurrency miner. Obfuscation of a cryptocurrency mining script can be implemented in more than one way. First, the true functionality of the script can be obfuscated by calling “useless” Wasm functions which will overload the trace with instructions without having an impact on the functionality of the mining script. This method will be referred to as *obfuscation by overloading*. Alternatively, it is possible to re-order the Wasm function calls, or even the executed instructions themselves, in such a way to generate an execution trace that is undetectable given the current mechanisms. This method will be referred to as *obfuscation by insertion* because new instructions are inserted between the existing instructions of the script.

Both approaches come with tradeoffs. Code obfuscation via the re-ordering of Wasm instructions requires an adept understanding of both WebAssembly and crypto mining algorithms, whereas code obfuscation via the addition of ‘useless’ Wasm functions will have an impact on performance as more instructions are executed for each hash. Here we will demonstrate a proof of concept for obfuscation by insertion, and a full implementation of obfuscation by overloading.

3.3.1 Obfuscation by Insertion

Obfuscation by insertion refers to the insertion of additional instructions into the instruction sequence of the mining script to differentiate the top subsequences of instructions in the obfuscated script from the top subsequences of known mining scripts. Instead of injecting benign instructions, this could also be accomplished by switching the order of individual instructions or breaking a block of instructions into multiple smaller blocks.

As mentioned earlier, a common method for in-browser cryptocurrency mining detection is to compare sequences of commonly used WebAssembly instructions of a known cryptocurrency miner to the sequences of instructions for the website that is suspected of cryptojacking. Therefore, to implement this type of obfuscation, we looked at how the detection module specifically detects cryptocurrency mining. In this case, our detection module takes a subsequence of 5 WebAssembly instructions for the comparisons, does not log any comparison instructions, and the subsequence of instructions is terminated at a control instruction to guarantee the instructions in the subsequence are executed consecutively. Thus, we inserted additional benign instructions within the common subsequences detected in the miner script.

We applied a proof-of-concept approach to this methodology. We used an already-generated execution trace from a commonly used browser-based web miner and inserted dummy Add and Sub instructions within the execution trace. This change is shown in the figures below:

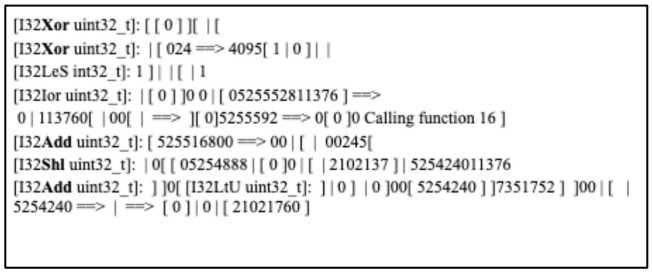

Figure 1: Excerpt from an instruction execution trace log for a known cryptocurrency miner

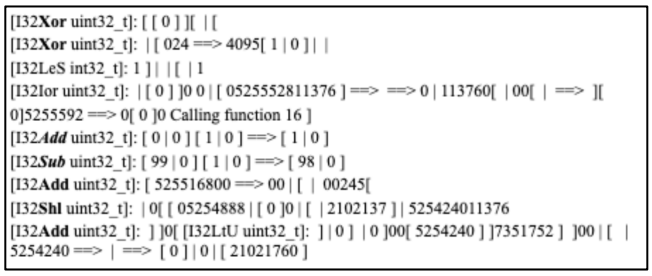

Figure 2: Instruction execution trace log for a known cryptocurrency miner, with inserted benign instructions

Figure 1 shows the unmodified instruction execution trace log, while Figure 2 shows the instruction execution trace log with Add and Sub instructions inserted between consecutive instructions.

3.3.2 Obfuscation by Overloading

Obfuscation by overloading refers to appending useless functions to a crypto mining script. These functions are called enough times to "overload" the list of top subsequences of the execution trace. Since the mining detection script only looks at the most common subsequences, all subsequences examined by the detection mechanism will exhibit benign functionality, and none of the sequences examined will be associated with cryptocurrency mining. Thus, the execution trace detection mechanism that we implemented as described earlier will be unable to identify our script as a cryptocurrency miner.

To implement this obfuscation scheme, the original mining script was modified to include dummy code that executes after each hash calculation performed by the script. The number of times this code is executed in between hashes is variable and depends on an "obfuscation factor" that is input as an argument to the script. Any code that runs a sufficient number of times and is repeatable enough to make its instruction subsequence occur very commonly is acceptable as dummy code for this purpose. The dummy code we used simply repeatedly calculated every Fibonacci number at every index until reaching the obfuscation factor, starting from the beginning of the Fibonacci sequence at each iteration:

```
   for (var i = 0; i<parseInt(process.argv[2]); i++)
(math.fib(i));
```

This script was then tested against our implemented execution trace detection script.

3.4 Test Methodology

With all pieces in place—the replication of a cryptocurrency mining script (both a standard version and obfuscated versions modified for stealth), the modifications to the V8 engine that allow the generation of execution trace logs for WebAssembly functions, and the implementation of a cryptocurrency mining detection script using these execution trace logs—we begin to describe the process by which we assessed our anti-detection modifications to the cryptocurrency mining script.

First, the obfuscation by insertion approach was tested. Though we did not actually modify our mining script to test this approach, we sought to prove that the concept is viable by modifying the execution trace log from the mining script. An execution trace log from a known cryptocurrency mining script was modified to contain benign instructions injected between the existing instructions. This log was input to our mining detection script, which detected the presence of mining based on whether the top instruction subsequences were the same, without a range of acceptability to prevent false positives (i.e., the naïve approach

instead of the targeted approach described earlier), and we recorded whether mining was detected.

Next, the obfuscation by overloading implementation of the mining script was tested. Due to concerns about reduced performance of the miner when obfuscated by overloading, we ran two tests to measure performance as a function of the obfuscation factor. Both tests were run on an AMD 3700x machine. First, we measured the hash rate of the miner for varying obfuscation factors ranging from 0 to 8000. Next, we measured the memory used by the program for the same range of obfuscation factors.

Next, we used the modified TaintAssembly V8 engine to produce execution trace logs from running the naïve mining script with obfuscation by overloading. At first a log was generated for an obfuscation factor of zero, and the log was input to our mining script to see whether mining was detected. We repeated this process with increasing obfuscation factors until we reached a point where the detection script could no longer reliably detect the generated execution trace as a cryptocurrency miner. Data on this critical obfuscation factor threshold was recorded.

Finally, as an extension of the project, we tested the in-browser implementation of our mining script (without obfuscation) against the most popular browser extensions intended to detect illicit cryptocurrency mining and evaluated the performance of these extensions.

4 RESULTS

The results of testing both methods of mining anti-detection—obfuscation by insertion and obfuscation by overloading—are summarized below. Effect on the performance of the mining script was considered, but the main metric by which these two methodologies were evaluated was whether they were able to pass undetected by our implemented execution trace detection mechanism.

4.1 Obfuscation by Insertion

To analyze our ideas for obfuscation by insertion, we looked at an execution trace from Cryptoloot, a commonly used browser-based web miner. After analyzing the log we extracted an excerpt from the log which contained one of the top 10 most commonly used sequences for crypto mining: [xor xor add shl add]. This excerpt can be seen in Figure 1. Note that this excerpt alone triggered the detection module and was labeled as a crypto miner, as expected.

Attempting to break this detection, we manually added two arithmetic instructions into this log snippet. We added a benign aI32Add and I32Sub instruction, as seen in Figure 2. Although we manually added the instructions, we placed them in a location where it could be possible to replicate within the code. The location where we decided to insert these two instructions was right after a function call, which we found was commonly used and called repeatedly. With these new instructions inserted into this excerpt, the new sequence of instructions is now [xor, xor, add, sub, shl] (assuming the window of instructions is still of length 5). This new sequence is no longer a commonly used sequence for cryptocurrency mining. When this modified execution trace is run through the detection program it is no longer detected as a cryptocurrency miner.

Although our study into this obfuscation technique was on one specific example of a sequence, we argue that using this technique throughout the whole cryptocurrency mining code will result in the same results. As mentioned above, these instructions were inserted at the beginning of a commonly called function. Therefore, in a wide scale experiment, adding more of these benign instructions to the beginning of commonly used functions will alter all the sequences. This in turn will result in a successful bypass of a detection module that uses comparison.

4.2 Obfuscation by Overloading

4.2.1 Performance Metrics

Obfuscation by overloading resulted in a significantly reduced performance of the mining script by some metrics due to the large number of dummy instructions needed in order to overload the top subsequences examined by the detection module. To quantify this, the hash rate of the cryptocurrency mining script—essentially the rate at which mining occurs—was measured and recorded for each obfuscation factor from 0 to 8000. Again, the obfuscation factor here simply refers to the number of times that the for loop containing the dummy Fibonacci code was executed. Results are shown in Figure 3.

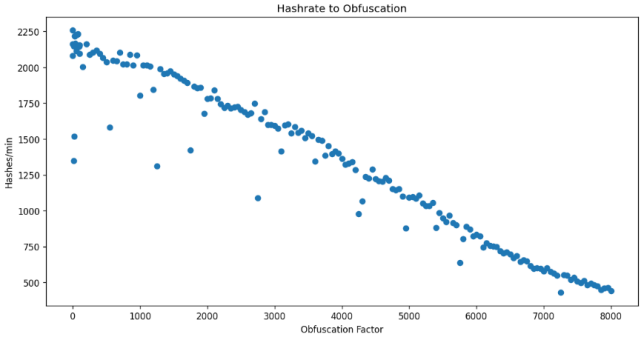

Figure 3: The hash rate of a cryptocurrency mining script (obfuscated by overloading) as a function of its obfuscation factor.

We see a consistent decreasing linear trend between these two parameters. As the obfuscation factor is increased, the number of hashes per minute decreases linearly. The number of hashes per minute is exactly correlated with the amount of cryptocurrency mined, so a higher obfuscation factor will result in a proportional decrease in income for the miner.

The other performance metric we observed was the memory used by the cryptocurrency mining script. Again, the obfuscation

factor was varied from 0 to 8000 and the memory usage of the program was recorded. These results are shown in Figure 4 below:

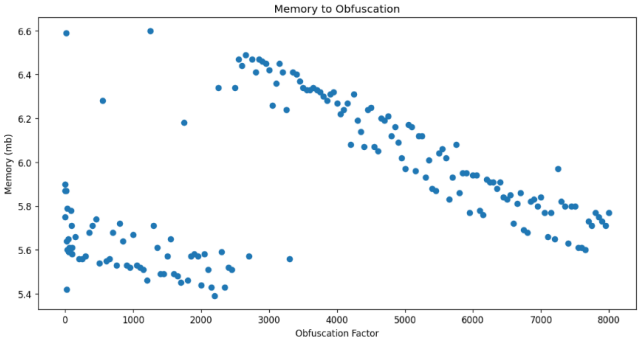

Figure 4: The memory used by a cryptocurrency mining script (obfuscated by overloading) as a function of its obfuscation factor.

As the obfuscation factor increases, more computational power is needed to hold the values of the Fibonacci sequence, and therefore more memory must theoretically be used. However, we see a trend in Figure 4 where the memory decreases steadily but has a large increase around an obfuscation factor of 2500. This could be because the interpreter recognizes how much memory is needed and then throttles performance to use less memory.

4.2.2 Detection of Mining

A mining script that had been obfuscated by overloading was run using the modified V8 engine and an execution trace log was produced. This was repeated for various obfuscation factors and the logs were input to our Python execution trace mining detection script. We found that the "critical point" where the detection script was no longer able to classify the obfuscated script as mining was consistently at an obfuscation factor of between 80 and 85. If the obfuscation factor was lower than this critical point, the detection script always was able to detect mining.

Crucially for a would-be attacker, an obfuscation factor of 85 does not have a large impact on the hash rate of cryptocurrency mining or memory used by the script. Even if the attacker were to use an obfuscation factor of 100 to be safe, they would not be making a large sacrifice in hash rate—from our previous results, the hash rate would only decrease by 250 (an overestimate), which is only about an 11% decrease from the initial, un-obfuscated hash rate. Similarly, because of the behavior of memory described earlier, an obfuscation factor that is only slightly higher may lead to a decrease in memory used. Thus, with limited drawbacks, an attacker has every incentive to use obfuscation by overloading in a mining script to bypass cryptojacking detection mechanisms.

Figure 5 shows the occurrence rate of the top instruction subsequence in the execution trace of the script for obfuscation factors up to 100. As defined earlier, the occurrence rate is the percentage of total instructions in which the top series of instructions occurs.

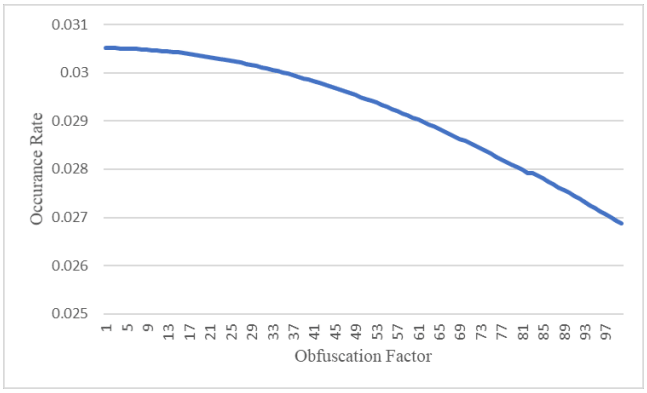

Figure 5: Occurrence Rate of Top Instruction for a mining script with various obfuscation factors.

This is important because as we found earlier, the naïve approach comparing the top subsequences fails to detect mining at an obfuscation factor of 85. What if we were to attempt to minimize false positives by implementing the targeted approach, defining an upper bound for the range of acceptability for the occurrence rate of the top subsequences? In this approach, even if the top subsequences match known crypto miners, the script will not be classified as a miner unless the occurrence rate of the top subsequence is over the upper bound of the range of acceptability. To still maintain accuracy for true positives, this range of acceptability would have to be lower than any occurrence rate of the top subsequences in an actual mining script that was detected. As we see from Figure 5, for an obfuscation factor of 85, the occurrence rate of the top subsequence was 0.0278. Therefore, a detection mechanism implementing the targeted approach should set its range of acceptability to 0.0278. However, this will only serve to prevent false positive detections, not to mitigate the fact that it is possible to defeat the detection mechanism with a relatively low amount of obfuscation while incurring limited sacrifice in performance.

4.3 Performance of Common Browser Extensions

Despite not being as advanced as the defense mechanisms we focused on in our project, most current tools to detect in-browser cryptocurrency mining simply use a deny-list to compare scripts against commonly known mining scripts. Though this was not the primary focus of this research, for completeness, we decided to demonstrate how these current browser extensions will not detect any scripts that are not on their deny-list as cryptojacking, even when the scripts have not been obfuscated.

To accomplish this, we used one of the most prominent anti-cryptojacking browser extensions, MinerBlock. MinerBlock claims that it uses a blacklist approach, which is "the traditional

approach adopted my most ad-blockers and other mining blockers," along with a mechanism to look for specific, known types of miners [5]. Our WebAssembly non-obfuscated mining script which we hosted online at not-a-cryptojacker.com was employed against MinerBlock. This script was not detected by MinerBlock even without any degree of obfuscation. This makes sense because the script was modified to not make calls to known mining providers, and instead implemented the mining itself, so it would not be blacklisted. Because the script was undetected with no obfuscation, we did not see a need to obfuscate it in further tests.

5 EVALUATION

Our results show that the current detection mechanisms of illicit in-browser cryptocurrency mining which function by examining the top subsequences of an execution trace will not detect execution traces that have been sufficiently obfuscated by insertion or obfuscated by overloading. In the interest of comprehensiveness, we must accompany these results with an honest evaluation of how they contribute to or change the current landscape of cryptojacking defenses and whether they are significant.

Our obfuscated attack is very easily successfully implemented against today's cryptojacking defenses. We chose one of the more sophisticated defenses to defeat, as most defenses are simply deny-lists. In fact, a prominent browser extension to detect cryptojacking was defeated by a non-obfuscated version of our cryptocurrency miner, simply because our miner did not make calls to known mining providers. We have further proven that it is still possible to defeat defenses that compare execution traces, such as the proposed defense that inspired our implementation, by inserting benign instructions in the middle of a mining instruction sequence or by adding dummy functionality in between hashes to overload the top instruction subsequences. These are examples of polymorphism, the various ways that a program can be changed to reduce similarity to known versions of that program.

However, there are some hypothetical methods of detection that could potentially defeat our attack implementation. For example, our detection system only examined the top 10 instruction subsequences found in the execution trace of a script. If a detection system were to examine a greater number of top subsequences, an obfuscation by overloading attack would most likely require a greater obfuscation factor to bypass the detection system. More research needs to be conducted to determine the exact relationship between the number of subsequences examined and the obfuscation threshold factor at which mining is no longer detected.

Obfuscation by insertion would still be a viable attack against such an injection mechanism because the instruction subsequences that identify an execution trace as belonging to a miner would be eliminated or muddled by extra instructions entirely, rather than kept intact but buried under many other subsequences. However, the insertions would have to be somewhat randomized, or else detection mechanisms could simply adapt by searching for the subsequences with inserted instructions. Even when randomized, the inserted instructions should not be able to break the functionality of the program. Since we tested obfuscation by insertion as a proof of concept by manually inserting the instructions, further research must be conducted on how to modify the script to produce such a randomized instruction trace that preserves functionality. Such research would establish greater statistical significance to our claim that insertion of benign instructions can bypass detection mechanisms.

Our results for obfuscation by overloading are more statistically significant. Though we only had the detection mechanism compare against one known mining script, the results can be generalized to work in comparison to other known mining scripts because no known mining script will have its top instruction subsequences be the instructions for carrying out the Fibonacci sequence. If the obfuscated script is added to a deny-list of scripts to which the detection mechanism compares, an attacker can simply stay one step ahead by implementing other useless functionality to obscure the true purpose of their script.

6 RELATED WORK

We mention here some of the most closely related work in browser-based cryptojacking and taint analysis.

A first look at browser-based cryptojacking by Eskandari et al. provides an examination of in-browser mining of cryptocurrencies [14]. Their work uses Monero to provide a background, threat model and potential mitigations for browser-based cryptojacking. Their potential mitigation approach includes: throttling clientside scripting, warning users when clientside scripting consumes excessive resources, and blocking the sources of known cryptojacking scripts. Our browser based cryptocurrency miner included its own self-throttling mechanism that was meant to limit suspicious resource use, so we could feasibly bypass this detection mechanism with our implementation. The limitations of blocking the sources of known cryptojacking scripts is a classic example of the limitations of blacklisting. The whitehats will always be reactive to the advances of the blackhats in this paradigm. Since we used a modified cryptocurrency mining script, the blacklisting detection mechanism of this paper will also not be useful in detecting our script.

The browsers strike back against parasitic miners on the web with the analysis of Tahir et al. of the top 50,000 websites from Alexa [15]. These researchers uncover that a significant percentage of websites are exploiting users' computing resources using heavily obfuscated code. Their approach uses Monero to evaluate the execution overhead of cryptojacking in the wild. The limitation of their approach is that a cryptojacking algorithm can be set to sufficiently throttle its resource usage in order to evade detection. It slows the immediate rate profit but can lead to equal or greater profits if patience and stealth are used.

Another mitigation of cryptojacking attacks, using taint analysis, seeks revenge against in-browser cryptojacking using a built-in Google Chrome extension [16]. Their approach uses a Man-in-the-Middle use case as testing for mitigation. Their proposed system uses a CPU-usage threshold algorithm. The limitation of their approach is that it relies on default 'blacklist-style' settings. An uninformed, or negligent, user could easily disable these settings from the graphical user interface. This goes against Schroeder's principle of not putting too much responsibility on the user.

The current state of academic literature regarding illicit cryptocurrency mining does not, to the best of our knowledge, include prior research from well-known venues generating potential attacks to avoid detection of mining. This paper is fairly unique in that regard and positions itself as a novel resource to be used for the improvement of future defenses.

7 FUTURE WORK

Our goal to circumvent the proper functioning of the cryptojacking defense proposed in W. Bian et al. (2019) was ultimately successful. To further test the capabilities of their defense strategy, our work could be extended to use a different taint log (than that of W. Fu et al.) to obtain the execution trace of the WebAssembly code. This could potentially make their defense more effective. More work could also be done to try to make the existing detection mechanism more robust. For example, a survey could be done of common obfuscation techniques, and the detection mechanism could be revised to ensure that it accounts for and defends against those methods of obscuring the intentions of attacker-supplied code.

Our attack is only configured to break a very specific type of cryptojacking defense. In order to make certain that there exist adequate resources to defend against these cryptocurrency mining attacks in the future, more research needs to be directed into this area. As an alternative to code obfuscation, there also exist techniques to detect cryptocurrency mining malware through the use of machine learning. As ML technology progresses in the coming years, it is likely that we will see more cryptojacking defenses that use this methodology.

One of the biggest issues with cryptojacking is that most victims don't even know that it's happening. More could also be done to provide the average internet user with information about known cryptojacking operations as well as preventative steps to take to avoid becoming victim to one of these schemes. In addition, it would be incredibly effective if large tech companies (e.g., Google, Apple, Microsoft etc.) announced this problem to their customers. To make existing defense technologies more accessible to the average consumer, they could be packaged into browser extensions and advertised by the tech companies in their app stores.

CONCLUSION

To contribute to the growing effort to defend against the new practice of hijacking the processing power of unknowing internet users, we propose our Wasm-based attack. By implementing two methods of code obfuscation on the program responsible for using a victim's processing power to mine for cryptocurrency, we successfully fooled our replicated implementation of an execution-trace cryptojacking detector proposed by W. Bian et al. (2019). Our attack is another resource that should be used by security experts when designing future anti-cryptojacking programs.

ACKNOWLEDGMENTS

The work described in this paper was supported in part by resources and faculty at the University of Minnesota, Twin Cities.

REFERENCES

[1] Interpol. "Cryptojacking." Cybercrime, 2021. https://www.interpol.int/en/Crimes/Cybercrime/Cryptojacking

[2] D. Carlin, J. Burgess, P. O'Kane and S. Sezer, "You Could Be Mine(d): The Rise of Cryptojacking," in IEEE Security & Privacy, vol. 18, no. 2, pp. 16-22, March-April 2020, doi: 10.1109/MSEC.2019.2920585.

[3] D. Goodin, "Currency-mining Android malware is so aggressive it can physically harm phones," Ars Technica, Dec. 19, 2017. [Online]. Available: https://arstechnica.com/information-technology/2017/12/ currency-mining-android-malware-is-so-aggressive-it -can-physically-harm-phones/

[4] White, Matthew. "The Crypto Grey Zone." PwC Middle East, 15 Nov. 2018, https://www.pwc.com/m1/en/media-centre/articles/the-crypto-grey-zone.html.

[5] Ismail, et al. "MinerBlock." v1.2.17, GitHub, 6 Feb. 2021. https://github.com/xd4rker/MinerBlock

[6] "NoScript FAQ." InformAction - Open Source Software, 2021. https://noscript.net/faq

[7] "Overview." WebAssembly, 1.0, 2017. https://webassembly.org/.

[8] Musch M., Wressnegger C., Johns M., Rieck K. (2019) New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In: Perdisci R., Maurice C., Giacinto G., Almgren M. (eds) Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2019. Lecture Notes in Computer Science, vol 11543. Springer, Cham. https://doi.org/10.1007/978-3-030-22038-9_2

[9] Weikang Bian, Wei Meng, and Mingxue Zhang. 2020. MineThrottle: Defending against Wasm In-Browser Cryptojacking. In Proceedings of The Web Conference 2020 (WWW '20), April 20–24, 2020, Taipei, Taiwan. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3366423.3380085

[10] Darabian, H., Homayounoot, S., Dehghantanha, A. et al. Detecting Cryptomining Malware: A Deep Learning Approach for Static and Dynamic Analysis. J Grid Computing 18, 293–303 (2020). https://doi.org/10.1007/s10723-020-09510-6

[11] Neto, H.N.C., Lopez, M.A., Fernandes, N.C. et al. MineCap: super incremental learning for detecting and blocking cryptocurrency mining on software-defined networking. Ann. Telecommun. 75, 121–131 (2020). https://doi.org/10.1007/s12243-019-00744-4

[12] Weikang Bian, Wei Meng, and Yi Wang. 2019. Poster: Detecting WebAssembly-based Cryptocurrency Mining. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19). Association for Computing Machinery, New York, NY, USA, 2685–2687. DOI:https://doi.org/10.1145/3319535.3363287

[13] William Fu, Raymond Lin, and Daniel Inge. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly, 2018. https://github.com/wfus/WebAssembly-Taint/blob/master/TaintAssembly.pdf

[14] S. Eskandari, A. Leoutsarakos, T. Mursch and J. Clark, "A First Look at Browser-Based Cryptojacking," 2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), 2018, pp. 58-66, doi: 10.1109/EuroSPW.2018.00014.

[15] R. Tahir, S. Durrani, F. Ahmed, H. Saeed, F. Zaffar and S. Ilyas, "The Browsers Strike Back: Countering Cryptojacking and Parasitic Miners on the Web," IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, 2019, pp. 703-711, doi: 10.1109/INFOCOM.2019.8737360.

[16] A. D. Yulianto, P. Sukarno, A. A. Warrdana and M. A. Makky, "Mitigation of Cryptojacking Attacks Using Taint Analysis," 2019 4th International Conference on Information Technology, Information Systems and Electrical Engineering (ICITISEE), 2019, pp. 234-238, doi: 10.1109/ICITISEE48480.2019.9003742.