Computers & Security 131 (2023) 103296

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

WebAssembly diversification for malware evasion

Javier Cabrera-Arteaga*, Martin Monperrus, Tim Toady, Benoit Baudry

KTH Royal Institute of Technology, Stockholm, Sweden

ARTICLE INFO

Article history:
Received 21 December 2022
Revised 27 April 2023
Accepted 13 May 2023
Available online 18 May 2023

Keywords:
WebAssembly
Cryptojacking
Software diversification
Malware evasion

ABSTRACT

WebAssembly has become a crucial part of the modern web, offering a faster alternative to JavaScript in browsers. While boosting rich applications in browser, this technology is also very efficient to develop cryptojacking malware. This has triggered the development of several methods to detect cryptojacking malware. However, these defenses have not considered the possibility of attackers using evasion techniques. This paper explores how automatic binary diversification can support the evasion of WebAssembly cryptojacking detectors. We experiment with a dataset of 33 WebAssembly cryptojacking binaries and evaluate our evasion technique against two malware detectors: VirusTotal, a general-purpose detector, and MINOS, a WebAssembly-specific detector. Our results demonstrate that our technique can automatically generate variants of WebAssembly cryptojacking that evade the detectors in 90% of cases for VirusTotal and 100% for MINOS. Our results emphasize the importance of meta-antiviruses and diverse detection techniques and provide new insights into which WebAssembly code transformations are best suited for malware evasion. We also show that the variants introduce limited performance overhead, making binary diversification an effective technique for evasion.

1. Introduction

WebAssembly is a binary format that has become an essential component of the web. It first appeared in 2017 as a fast and safe complement for JavaScript (Haas et al., 2017). The language provides low-level constructs enabling efficient execution, much closer to native code than JavaScript. Since its inception, the adoption of WebAssembly has grown exponentially, even outside the web (Cabrera Arteaga et al., 2022). Its early adoption by malicious actors is further evidence of WebAssembly's success.

The primary black-hat usage of WebAssembly is cryptojacking (Rokicki et al., 2022). Such WebAssembly code mines cryptocurrencies on users' browsers for the benefit of malicious actors and without the consent of the users (Musch et al., 2019b). The main reason for this phenomenon is that the core foundation of cryptojacking is: the faster, the better. In this context, WebAssembly, a binary instruction format designed to be portable and fast, is a feasible technology for implementing and distributing cryptojacking over the web. A Kaspersky report about the state of cryptojacking in the first three quarters of 2022 confirms the steady growth in the usage of cryptominers (Kaspersky, 2022). The report shows that Monero (2022) is the most used cryptocurrency for cryptomining in the browser. Attackers might hide WebAssembly cryptominers (Tekiner et al., 2021) in multiple locations inside web applications.

Antivirus and browsers provide support for detecting cryptojacking. For example, the Firefox browser supports the detection of cryptomining by using deny lists (Mozilla, 2019). The academic community also provides related work on detecting or preventing WebAssembly cryptojacking (Bian et al., 2020; Kelton et al., 2020; Kharraz et al., 2019; Naseem et al., 2021; Romano et al., 2020; Wang et al., 2018). Yet, it is known that black-hats can use evasion techniques to bypass detection. Only the previous work of Bhansali et al. (2022) investigates the possibility of WebAssembly cryptojacking to evade detection techniques. This is a crucial motivation for our work, one of the first to study WebAssembly malware evasion.

Our work is different from Bhansali et al. (2022)'s in the following aspects. First, we extend the evaluation of MINOS by using VirusTotal and, we empirically demonstrate that this latter is a valid cryptojacking malware meta-detector for WebAssembly to be used as baseline (see Section 5). Second, we conduct an evaluation of the correctness and efficiency of the Wasm variants, which provides insights into the trade-offs and limitations of bytecode-level transformations for malware evasion in WebAssembly. Our approach, performs bytecode transformations at the WebAssembly level, instead of source code based transformations like the

* Corresponding author.
E-mail addresses: javierca@kth.se (J. Cabrera-Arteaga), monperrus@kth.se (M. Monperrus), toady@eecs.kth.se (T. Toady), baudry@kth.se (B. Baudry).

https://doi.org/10.1016/j.cose.2023.103296

Bhansali's technique. We focus on software diversification techniques in the spirit of Cohen (1993), as this technique has the ability to generate many variants, while the impact on the binary size and performance can be controlled through the selection of specific diversification transformations (Lundquist et al., 2016).

In this paper, we design, implement and evaluate a full-fledged evasion pipeline for WebAssembly. Concretely, we use wasm-mutate as a diversifier (Bytecodealliance, 2021), which implements 135 possible bytecode transformations, grouped into three categories: peephole operator, module structure, and control flow. We demonstrate the effectiveness of our evasion technique against two cryptojacking detectors: VirusTotal, a general detection tool that comprises 60 antiviruses and, MINOS (Naseem et al., 2021), a WebAssembly-specific detector.

We evaluate our proposed evasion technique on 33 cryptojacking malware that we curated from the 8643 binaries of the wasm-bench dataset (Hilbig et al., 2021), to our knowledge, the most exhaustive collection of real-world WebAssembly binaries. We experiment with the 33 binaries marked as potentially dangerous by at least one antivirus vendor of VirusTotal. We empirically demonstrate that evasion is possible for all of these 33 real-world WebAssembly cryptojacking malware while using a WebAssembly-specific detector. Remarkably, we find 30 cryptominers for which our technique successfully generates variants that evade VirusTotal. Our set of malware includes 6 cryptojacking programs that are fully reproducible in a controlled environment. With them, we assess that our evasion method does not affect malware correctness and generates fully functional malware variants with minimal overhead.

Our work provides evidence that the malware detection community has opportunities to strengthen the automatic detection of cryptojacking WebAssembly malware. The results of this work are actionable, as we provide quantitative evidence on specific malware transformations on which detection methods can focus. To sum up, the contributions of this work are:

- A full-fledged cryptojacking malware evasion pipeline for WebAssembly, based on a state-of-the-art binary diversification. We provide the repository of the tool at https://github.com/ASSERT-KTH/wasm_evasion.
- A systematic evaluation of our cryptojacking evasion pipeline, including effectiveness, performance, and correctness.
- Actionable evidence on which transformations are better for evading WebAssembly cryptojacking detectors, calling for future work from the academic and the industrial community alike.
- A reproducible comparison of VirusTotal with MINOS, a WebAssembly-specific detector, showing the relevance of VirusTotal as a valid and practical cryptojacking meta-detector.

This paper is structured as follows. In Section 2, we introduce WebAssembly for cryptomining and the state-of-the-art on malware detection and evasion techniques and its limitations for WebAssembly cryptojacking. In Section 3, we instantiate and explain malware evasion for WebAssembly cryptojacking in a real scenario. We follow with the technical description of our malware evasion algorithms in Section 4. We formulate our research questions in Section 5, answering them in Section 6. We discuss our research in Section 7, in order to help future research projects on similar topics. We finalize with our conclusions Section 8.

2. Background & related work

In this section, we introduce WebAssembly. Besides, we illustrate its usage for cryptojacking. Then, we discuss how WebAssembly cryptojacking can be detected, and the most common techniques used to evade such detection.

2.1. WebAssembly

WebAssembly (Wasm) is a binary instruction set meant initially for the web. It was adopted as a standard language for the web by the W3C in 2017, building upon the work of Haas et al. (2017). One of Wasm's primary advantages is that it defines its own Instruction Set Architecture (ISA), which is both straightforward and platform-independent. As a result, a Wasm binary can execute on virtually any platform, including web browsers and server-side environments. Since its introduction, all major web browsers have implemented support for WebAssembly, reporting to be only 10% slower than machine code during runtime.

WebAssembly programs are compiled ahead-of-time from source languages such as C/C++, Rust, and Go, utilizing compilation pipelines like LLVM. This allows Wasm to benefit from ahead-of-time compiling optimizations, improving its performance. A Wasm binary is comprised of sections, which are consecutive sequences of bytes in the binary file. In contrast to the absolute order of sections in Windows Portable Programs, sections in Wasm binaries have a relative order between them. Thus, Wasm can be considered a more flexible binary format.

WebAssembly programs operate on a virtual stack that allows for only four data types: i32, i64, f32, and f64. These same data types are used to annotate the numeric operations in the WebAssembly code. Additionally, a WebAssembly program might include several custom sections. For example, binary producers such as compilers use it to store metadata. A WebAssembly code also declares memories and globals, which are used to store, manipulate and share data during program execution, e.g. to share data with the host engine of the WebAssembly binary.

WebAssembly is designed with isolation as a primary consideration. For instance, a WebAssembly binary cannot access the memory of other binaries or interact directly with a browser's built-in API, such as the DOM or the network. Instead, communication with these features is constrained to functions imported from the host engine, ensuring a secure and safe Wasm environment. Moreover, control flow in WebAssembly is managed through explicit labels and well-defined blocks, which means that jumps in the program can only occur inside blocks, unlike regular assembly code.

In Listing 1, we provide an example of a C program that contains a function declaration, a loop, a loop conditional, and a memory access. When the C code is compiled to WebAssembly, it produces the code shown in Listing 2. The stack operations are folded with parentheses. The module in the example contains the components described previously.

2.2. Malware in WebAssembly

The use cases of WebAssembly in browsers focuses on computation-intensive activities such as gaming or image processing. Also, malign actors have taken advantage of WebAssembly to carry out their activities, and cryptojacking is the most common usage observed so far (Musch et al., 2019a; 2019b). The reason for this is that cryptojacking involves executing vast amounts of hash

```
int a[100];
void cc1(){
    for(int i = 0; i < 100;
        ↪ i++){
        a[i] = i;
    }
}
```

Listing 1. C program containing function declaration, loops, conditionals and memory access.

```
(module
  (@custom "producer" "llvm.." )
  (func (;5;) (type 1)
    (loop  ;; label = @1
      (if  ;; label = @2
        (i32.eqz
          (i32.ge_s
            (i32.load
              (local.get 0))
            (i32.const 100)))
        (then (i32.store
            (i32.add
              (i32.shl
                (i32.load
                  (local.get 0))
                (i32.const 2))
              (i32.const 1024))
            (i32.load
              (local.get 0)))
          (i32.store
            (local.get 0)
            (i32.add
              (i32.load
                (local.get 0))
              (i32.const 1)))
          (br 1 (;@1;)))))))
  )
  (memory (;0;) 256)
  (global (;1;) i32 (i32.const 0))
  (export "memory" (memory 0)))
```

Listing 2. WebAssembly code for C code in Listing 1.

functions, which requires significant computing resources. In comparison to JavaScript, WebAssembly is significantly faster at handling these intense hashing operations in the browser (Haas et al., 2017).

Web cryptojacking is often carried out by including a malicious JavaScript+WebAssembly payload, which then execute on the victim's browser without their knowledge (Tekiner et al., 2021). For example, websites that offer illegal download or adult sites, often include cryptojacking in their webpages to generate passive income. Since cryptojacking is difficult to detect and remove, it can remain on a victim's computer for an extended period, continuing to consume resources and to generate income for the attacker. This lucrative form of malware does need vulnerabilities or stealing credentials.

2.3. Malware detection

Malware detection determines if a binary is malicious or not. This process can be based on static, dynamic, or hybrid analysis (Aslan and Samet, 2020). In this section, we highlight works in the area of malware detection. Static-based approaches analyze the source code or the binary to find malign patterns without executing them. The literature reports a range of techniques, from simple checksum checking to advanced machine learning methods, that have subsequently been adopted by commercial antiviruses (Botacin et al., 2022; Li et al., 2021). In the context of WebAssembly, MineSweeper is a detection method based on static analysis (Konoth et al., 2018) of WebAssembly. Its detection strategy depends on the knowledge of the internals of CryptoNight, one popular library for cryptomining. In the same context, MINOS is a state-of-art static detection tool that converts WebAssembly binaries to vectors for malware detection (Naseem et al., 2021).

MINOS is a practical approach for detecting malicious WebAssembly binaries. It works by converting the Wasm binary's bytestream into a 100×100 grayscale image, which is then fed into a Convolutional Neural Network (CNN). The CNN has learned patterns in the image to classify it as either benign or malicious. This approach is similar to image-based methods used in other areas (Liu and Wang, 2016), e.g., for detecting Windows malware (Kalash et al., 2018; Lachtar et al., 2023). We believe MINOS is the optimal static approach to detect WebAssembly malware due to its simplicity and practicality, such as being easily implemented as a browser extension.

Dynamic analysis for malware detection is based on the execution of the malware code to identify potentially dangerous behaviors (Egele et al., 2008). Usually, this is done by monitoring some functions, such as API calls. For example, BLADE (Lu et al., 2010) is a Windows kernel extension that aims to eliminate drive-by malware installations. It wraps the filesystem for browser downloads for which user consent has been involved. It thwarts the ability of browser-based exploits to download and execute malicious content surreptitiously. SEISMIC and MineThrotle also perform dynamic analyses (Bian et al., 2020; Wang et al., 2018) on WebAssembly binaries to profile instructions that are specific to cryptominers. For example, cryptominers overly execute `XOR` instructions. SEISMIC and MineThrotle use machine learning approaches to classify the binary as benign or malign based on collecting runtime profiles. On the same topic, MinerRay (Romano et al., 2020) detects cryptojacking in WebAssembly binaries by analyzing their control flow graph at runtime, searching for structures that are characteristic of encryption algorithms commonly used for cryptojacking. CoinSpy is another malware detector based on dynamic analysis (Kelton et al., 2020). It uses a convolutional neural network to analyse the computation, network, and memory information caused by cryptojackers running in client browsers.

Hybrid approaches use a mix of static and dynamic detection techniques. The main reason to use hybrid approaches is the impracticability of executing the whole program. Thus, only pieces of code that can be quickly executed are dynamically analyzed. For example, AppAudit embodies a novel dynamic analysis for Android applications that can simulate the execution of some parts of the program (Xia et al., 2015). For WebAssembly, Outguard (Kharraz et al., 2019) trains a Support Vector Machine model with a combination of cryptomining function names obtained statically and dynamic information such as the number of web workers used in the web application that is analyzed.

It is possible to combine several independent detectors into a meta-antivirus. Each detector embeds some heuristics that are good at detecting specific types of malware (Moser et al., 2007). Hence, their combination can effectively detect a broader range of malware, e.g., using relationship analysis. VirusTotal (GoogleLLC, 2022; VirusTotal, 2020) is a consolidated meta-antivirus. VirusTotal operates with 60 antivirus vendors to provide malware scanning. Through its API, a program can be labeled by 60 antiviruses. This aggregation is used to determine if an asset under analysis is malicious, e.g., by voting. Previous works used VirusTotal to assess detection efficiency, Peng et al. (2019) because it is a proxy to evaluate state-of-art techniques in combination with commercial antiviruses. In this work, we follow the same methodology, using VirusTotal to assess our technique's ability to evade cryptojacking detectors.

While concerns have been raised about the use of VirusTotal for some malware and file type families (Botacin et al., 2020), it can be considered for WebAssembly cryptojacking detection. We empiri-

cally highlight later in this paper that VirusTotal is slightly better than the WebAssembly-specific detector MINOS regarding the detection of cryptojacking malware.

2.4. Malware evasion

Malware evasion techniques aim at avoiding malware detection (Afianian et al., 2019) Potential attackers use a wide range of techniques to achieve evasion, such as genetic programming (Castro et al., 2019). With time, the techniques to avoid detection have grown in complexity and sophistication (Aghakhani et al., 2020). For example, Chua and Balachandran (2018) proposed a framework to automatically obfuscate Android applications' source code using method overloading, opaque predicates, try-catch, and switch statement obfuscation, creating several versions of the same malware. Also, machine learning approaches have been used to create evading malware (Dasgupta and Osman, 2021), based on a corpus of pre-existing malware (Bostani and Moonsamy, 2021). While most approaches try to break static malware detectors, more sophisticated techniques avoid dynamic detection, usually involving throttling techniques or dynamic anti-analysis checks (Lu and Debray, 2013; Payer, 2014). Wang proposes the concept of Accrued Malicious Magnitude (AMM) to identify which malware features should be manipulated to maximize the likelihood of evading detection (Wang et al., 2021).

In the context of WebAssembly, malware evasion is nearly unexplored. Only Romano et al. (2022) recently proposed wobfuscator, a code obfuscation technique that transforms JavaScript code into a new JavaScript file and a set of WebAssembly binaries. Their technique mostly focuses on JavaScript evasion and not WebAssembly evasion.

Bhansali et al. (2022) propose a technique where WebAssembly binaries are transformed while maintaining the functionality with seven different source code obfuscation techniques. They evaluate the effectiveness of the techniques against MINOS (Naseem et al., 2021). They show these transformations can generate malware variants that evade the MINOS classifier.

3. WebAssembly cryptojacking malware evasion in practice

Figure 1 illustrates our attack scenario: a practical WebAssembly cryptojacking attack consists of three components: a WebAssembly binary, a JavaScript wrapper, and a backend cryptominer pool. The WebAssembly binary is responsible for executing the hash calculations, which consume significant computational resources. The JavaScript wrapper facilitates the communication between the WebAssembly binary and the cryptominer pool. Overall, a successful cryptojacking attack on a victim's browser consists in the following sequence of steps. First, the victim visits a web page infected with the cryptojacking code. The web page establishes a channel to the cryptominer pool, which then assigns a hashing job to the infected browser. The WebAssembly cryptominer calculates thousands of hashes inside the browser, in parallel using multiple browser workers (Mozilla, 2022). Once the malware server receives acceptable hashes, it is rewarded with cryptocurrencies for the mining. Then, the server assigns a new job, and the mining process starts over.

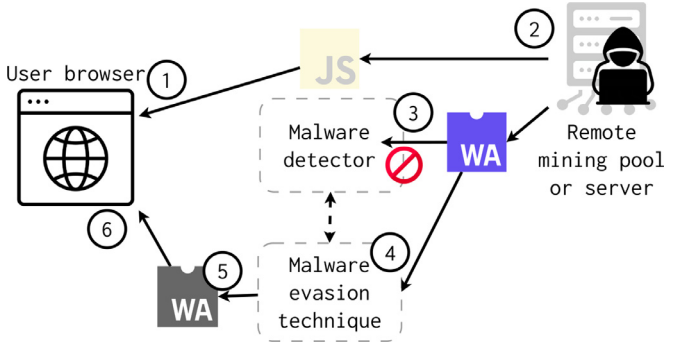

Fig. 1. WebAssembly evasion in practice. The user visits a webpage containing cryptojacking malware that uses network resources to operate. A malware detector blocks identified malicious WebAssembly binaries. The attacker, using a malware oracle, creates a WebAssembly cryptojacking malware variant that evades detection. Finally, the attacker delivers the modified binary, initiating the cryptojacking process and compromising the browser.

Some detection techniques discussed in Section 2.3 can be deployed in the browser directly to prevent cryptojacking. The primary objective of our work is to demonstrate the possibility of using code diversification to bypass cryptojacking defenses. Concretely, the following workflow can happen to successfully evade placed defenses: i) The user visits a webpage that contains a cryptojacking malware, which utilizes network resources to execute, (1) and (2) in Fig. 1. Cryptojacking malware can be injected through malicious browser extensions, malvertising, compromised websites, or deceptive links (Tekiner et al., 2021). ii) A malware detector blocks WebAssembly binaries that are identified as malicious (3). The malware detector system can be implemented locally or remotely. For instance, a proxy can intercept and send network resources to an external detector through the detector's API. iii) The attacker, based on a malware oracle, crafts a WebAssembly cryptojacking malware variant that evade the detection (4). iv) The attacker delivers the modified binary instead of the original one (5), which initiates the cryptojacking process and compromises the browser (6).

The idea is that attackers rapidly diversify their WebAssembly code to stay ahead of the defense system and maintain successful cryptojacking operations. Crucially, attackers must ensure that the diversified binaries they use for cryptojacking meet specific performance requirements, which is an aspect we will study in Section 4.

4. Diversification for malware evasion in WebAssembly

In this section, we explain a technique for potential attackers to craft a WebAssembly binary that evades detection (steps 4, 5 and 6 in Fig. 1).

In Fig. 2 we illustrate our generic architecture for the malware evasion component. The workflow starts by passing a WebAssembly malware binary to a software diversifier (1). The diversifier generates binary variants, which are passed to a malware oracle (2). The oracle returns labeling feedback for the binary variant: malware or benignware. The oracle result is the input for a fitness function that steers the construction of a new binary on top of the previously diversified one. This process is repeated until the malware oracle marks the mutated binary as benign or a timeout is reached (4). For the sake of open science and for fostering research on this important topic, our implementation is made publicly available on GitHub: https://github.com/ASSERT-KTH/wasm_evasion.

4.1. Diversifier

Conceptually, our approach is parametrized by a semantic preserving diversifier (Cohen, 1993). For our prototype implementation, we select one diversifier that supports wasm-to-wasm diversification and performs almost non-costly transformations: wasm-mutate (Bytecodealliance, 2021). This tool takes a WebAssembly module as input and returns a set of variants of that module. wasm-mutate follows the notion of program equivalence modulo input (Le et al., 2014), i.e., the variants should provide the same

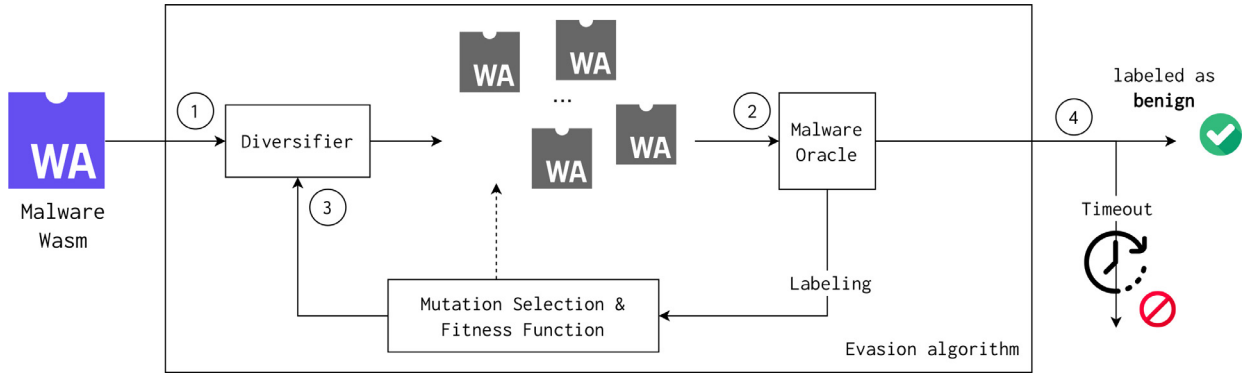

Fig. 2. Our original workflow of binary diversification for malware evasion in WebAssembly. The workflow begins with a WebAssembly malware binary sent to a software diversifier. The diversifier creates binary variants, which are analyzed by a malware oracle that labels them as malware or benignware. The oracle's results guide the development of a new binary based on the previous one. This process continues until the malware oracle labels the mutated binary as benign or a timeout occurs.

output for the same program inputs. It represents the search space for new variants as an e-graph (Willsey et al., 2020), and it exploits the property that any traversal through the e-graph represents a semantically equivalent variant of the input program. wasm-mutate can generate thousands of semantically equivalent variants from a single binary in a few minutes.

Wasm-mutate defines 135 possible transformations on an input WebAssembly binary, grouped into three categories. The peephole operator is the first category. It is responsible for rewriting instruction sequences in function bodies. It selects a random instruction within the binary functions and applies one or more of the 125 rewrite rules. Finally, it re-encodes the WebAssembly module with the new, rewritten expression to produce a binary variant. The second category of transformations in wasm-mutate implements module structure transformations. It operates at the level of the WebAssembly binary structure. It includes the following eight transformations: add a new type definition, add/modify custom sections, add a new function, add a new export, add a new import, add a new global, remove a type and remove a function. The third transformation category is on the control flow graph of a function code level. wasm-mutate performs two possible transformations: unroll a loop or swap the branches of a conditional block.

The decision of using wasm-mutate as a diversifier is based on three key factors. First, while diversification approaches for WebAssembly from LLVM sources exist (Cabrera Arteaga et al., 2022; Cabrera-Arteaga et al., 2021), compiler-based diversification may include compiler fingerprints in the built binaries, which is bad for stealthy evasion. Second, while optimization-based approaches could be used to diversify WebAssembly binaries from source code (Ren et al., 2021), optimizations are usually all applied at once, providing a smaller diversification space, hence fewer opportunities for evasion. Finally, wasm-mutate is a tool that implements many useful semantically equivalent transformations, making it well-suited as a diversifier with minimal engineering effort. Therefore, using wasm-mutate as a diversifier provides a practical approach for evasion in WebAssembly. We believe that attackers would take the same path and reach the same conclusion.

For the sake of illustration, in Listing 3, we present a variant of the WebAssembly code shown in Listing 2. We generate this variant using the wasm-mutate diversifier, with two transformations. The changes made to the original code are highlighted in orange and green. The first transformation, highlighted in orange, involves removing the custom section that indicates the producer of the original binary. The second transformation, highlighted in green, replaces the shift-left operation in the original binary with two consecutive multiplications of the same value.

```
(module
 - (@custom "producer" "llvm.." )
  (func (;5;) (type 1)
    (loop  ;; label = @1
      (if  ;; label = @2
        (i32.eqz
          (i32.ge_s
            (i32.load
              (local.get 0))
            (i32.const 100)))


          (then (i32.store
              (i32.add
                (i32.mul
                  (i32.load
                    (local.get 0))
                  (i32.load
                    (local.get 0))
                )
                (i32.const 1024)
              )
              (i32.load
                (local.get 0)))
            (i32.store
              (local.get 0)
              (i32.add
                (i32.load
                  (local.get 0))
                (i32.const 1)))
            (br 1 (;@1;)))))
    )
  ...
```

Listing 3. Wasm-mutate transformation applied over Listing 2.

4.2. Malware oracle

To determine whether a given sample is malicious or not, we rely on a malware oracle. The simplest form of malware oracle is a binary classifier that outputs a label of {malware, benignware}. In addition to binary classifiers, we also consider numerical oracles

that provide a value representing the likelihood that a sample is malicious.

In our experiments, we use VirusTotal (2020) as our malware oracle. VirusTotal operates with 60 antivirus vendors to provide malware scanning. Users submit binaries, and they receive labels from different vendors. The resulting 60 labels can then be used to determine if a queried asset is malicious, e.g., by voting. VirusTotal can be used as both a binary and a numerical oracle (Zhu et al., 2020). We use VirusTotal as a binary oracle by returning malware if at least one vendor classifies the binary as malware. We also use VirusTotal as a numerical oracle, using the number (between 0 and 60) of oracles labeling a binary as malware.

Research (Botacin et al., 2020), classification labels assigned to samples by vendors in VirusTotal can change over time due to new antivirus releases. In our work, we operate under the assumptions outlined in Section 3 and consider a scenario where an attacker develops and executes an evasion technique in under an hour. This timeframe is significantly shorter than the time it takes for classification labels to change in VirusTotal, which typically takes several days.

4.3. Transformation selection & fitness function

The third step of our workflow consists of two actions towards synthesizing a malware variant: select a wasm-mutate transformation to apply; and determine if the transformation is applied. This latter decision depends on: 1) the result of the malware oracle at the previous iteration and 2) an estimation of the ability of the new transformation to generate an evading binary. For this estimation, we implement two variations of our evasion algorithm.

First, we use a binary malware oracle. In this context, we always apply the transformation. This is the *baseline evasion algorithm*, which is discussed in more detail later.

The second variation of the evasion algorithm includes a fitness function that uses a numerical oracle to estimate if the transformation should be applied. The fitness function uses the information from VirusTotal and is the total number of vendors (between 0 and 60) that label a binary as malware:

$$FF(m) = \Sigma_{i=0}^{i=60} \begin{cases} 1 \text{ if } v_i(m) \text{ returns } \texttt{malware} \\ 0 \text{ i.o.c} \end{cases}$$

This fitness function is used in our second proposed algorithm and is discussed in details below.

4.3.1. Baseline evasion

The baseline evasion algorithm is described in Algorithm 1. It uses VirusTotal as a binary oracle (true if at least one VirusTotal detects malware, false otherwise). In this algorithm, step 1 is in line 3, steps 2 and 4 are in lines 4 to 6, and step 3 is in line 7. Each iteration of this algorithm, "stacks" a transformation on top of the previous ones (line 7) until the binary is marked as benign (line 5) or the maximum number of iterations is reached (line 2).

With this algorithm, a transformation is randomly selected at each iteration, and always applied. Hence, the baseline algorithm can require many iterations and oracle queries to turn the original malware into a misclassified binary. Second, some transformations might suppress the effect of previous ones. Third, the baseline algorithm considers each vendor equally good at detecting a malware, which is naive as the vendors display considerable diversity regarding detection strength. An algorithm that would target evasion on the strongest vendor first would increase the overall performance of the evasion process. Finally, we might generate a binary that entirely evades the oracle but is unpractical in terms of size or its execution performance.

Algorithm 1: Baseline evasion algorithm.

input : binary W, diversifier D, Malware Oracle MO
output: Benign binary M'
$M \leftarrow W$**while** *Not max iterations* **do**
  $M' \leftarrow D(M)$**if** $MO(M') ==$ *"benign"* **then**
    return M';
  else
    $M \leftarrow M'$;
return "Not evaded";

Algorithm 2: MCMC evasion algorithm.

input : binary W, diversifier D, fitness function FF
output: Benign binary M'
$M \leftarrow W$previous_fitness = $FF(W)$**while** *Not max iterations* **do**
  $M' \leftarrow D(M)$current_fitness = $FF(M')$**if** *current_fitness == 0* **then**
    // Zero means that none vendor marks// the binary as malware**return** M';
  else
    $p \leftarrow random()$**if** $p < \min\left(1, \exp\left(\sigma \frac{previous\_fitness}{current\_fitness}\right)\right)$ **then**
      $M \leftarrow M'$;previous_fitness = current_fitness;
return "Not evaded";

4.3.2. MCMC evasion

To overcome the limitations of the baseline algorithm, we devise the *MCMC evasion algorithm*, which we now discuss. It is a Markov Chain Monte Carlo (MCMC) sampling (Hastings, 1970) of the transformations to apply (Schkufza et al., 2012). MCMC is used to sample from the space of transformation sequences to maximize the likelihood of oracle evasion in two ways.

The algorithm for the MCMC evasion is given in Algorithm 2. The halting condition is met when a mutated binary is marked as benign, or a maximum number of iterations is reached. The algorithm implements the MCMC in lines 6 to 15. The Markov decision function in line 12 is used to determine whether it is worth applying the transformation at this step or whether it should be skipped. This decision function is based on the current transformation's fitness and the fitness value saved at the previous step (line 14). The core idea is to favor a binary variant that evades the largest number of vendors (lines 4 and 13). Therefore, the number of oracle calls should decrease as the algorithm searches for transformations that converge toward total evasion. On the other hand, if a new transformation step decreases the fitness value, it is likely to be ignored. The classical MCMC acceptance criteria in line 12 is meant to prevent the algorithm from being stuck in local minima.

In line 12, the fraction calculated in the exponentiation is controlled by the σ parameter. By setting a low σ parameter, we can turn the MCMC evasion algorithm into a greedy algorithm. In this case, the algorithm selects a new transformation only if the fitness value is higher than in the previous iteration. On the contrary, if the σ parameter is significant, the algorithm searches for local maxima. In our experiments (Section 5), we explain how we select the values of σ.

The MCMC evasion algorithm addresses the three limitations of the baseline mentioned in the previous section. First, the fitness function selects transformations, thus reducing the total number of transformations that are actually performed. Second, MCMC aims at increasing fitness, which reduces the risk of suppressing a valuable transformation performed in the previous step. Third, by

favoring solutions that maximize the number of evaded vendors, MCMC biases the search towards evading the strongest vendors.

5. Experimental methodology

In this section, we enunciate the research questions around which we assess the ability of our technique to evade malware detectors. We also describe our dataset of malware, as well as the metrics we define to answer our research questions.

5.1. Research questions

- **RQ1. To what extent can cryptojacking malware detection be bypassed by WebAssembly diversification?** With this research question, we evaluate the feasibility of binary transformations on malware and how they affect the detection of cryptojacking.
- **RQ2. To what extent can the attacker minimize the number of calls to the cryptojacking detection oracle?** In real-world scenarios, the number of calls to the oracle is limited. With this question, we analyze the ability of our technique at limiting the number of oracles calls made during the evasion process.
- **RQ3. To what extent do the evasion techniques impact cryptojacking malware functionality?** The evasion algorithms might generate variants that evade the detectors but modify their core malicious functionality. This research question evaluates the correctness of the created variants, as well as their efficiency.
- **RQ4. What are the most effective transformations for WebAssembly cryptojacking malware evasion?** This research question provides empirical evidence on which types of transformations are better for WebAssembly cryptojacking evasion.
- **RQ5. To what extent can Wasm diversification evade cryptojacking detection with MINOS?** In this research question, we evaluate the feasibility of our technique for evading a state-of-the-art detector, MINOS, which is tailored to the analysis of WebAssembly.

5.2. Dataset selection

To answer our research questions, we curate a dataset of WebAssembly malware. For this, we filter the wasmbench dataset of Hilbig et al. (2021) to collect suspicious malware according to VirusTotal. The wasmbench dataset contains 8643 binaries collected from GitHub repositories and web pages in 2021. To our knowledge, this dataset is the newest and most exhaustive collection of real-world WebAssembly binaries. On August 2022, we passed the 8643 binaries of wasmbench to VirusTotal (2020), and we 33 binaries were marked as potentially dangerous by at least one antivirus vendor of VirusTotal. All malware were marked as cryptojacking programs and we use these WebAssembly binaries to answer our research questions for cryptojakcing malware evasion.

In Table 1 we describe the 33 binaries detected as malware by VirusTotal. The table contains the following properties as columns:

Table 1
The 33 real-world WebAssembly cryptojacking used in our experiments. The table contains: the 256 hash of the WebAssembly cryptojacking, its size in bytes, the number of instructions, the number of functions defined inside the binary and the number of VirusTotal vendors that detect the binary at the time of writing. The last column contains the origin of the binary.

Hash	S	#I.	#F.	#D	Origin
9d30e7f0	68,796	30,768	61	30	http archive
8ebf4e44	68,803	30,768	61	26	Web crawling
47d29959	68,796	30,768	61	31	Yara[a]
aafff587	97,551	47,033	72	6	SEISMIC[b]
dc11d82d	67,496	30,246	49	20	MinerRay[c]
0d996462	70,972	30,531	30	19	SEISMIC, MinerRay
fbdd1efa	94,270	45,905	40	18	SEISMIC
a32a6f4b	94,461	45,940	40	18	SEISMIC
d2141ff2	70,111	31,783	30	9	MinerRay, SEISMIC
046dc081	74,099	31,783	29	6	MinerRay, SEISMIC
24aae13a	62,458	28,339	37	4	SEISMIC
000415b2	62,466	28,339	37	3	SEISMIC
643116ff	73,010	31,866		6	MinerRay
006b2fb6	87,502	39,544	90	4	DeepMiner[d]
15b86a25	100,755	45,881	79	4	MinerRay
4cbdbbb1	104,666	47,916	62	3	SEISMIC
119c53eb	137,320	67,069	79	2	SEISMIC
f0b24409	77,572	34,918	59	2	MinerRay
c1be4071	77,572	34,918	59	2	MinerRay
a74a7cb8	77,572	34,918	59	2	MinerRay
a27b45ef	77,572	34,918	59	2	MinerRay
6b8c7899	77,572	34,918	59	2	MinerRay
68ca7c0e	77,572	34,918	59	2	MinerRay
65debcbe	77,572	34,918	59	2	MinerRay
5bc53343	77,572	34,918	59	2	MinerRay
59955b4c	77,572	34,918	59	2	MinerRay
942be4f7	103,520	46,208	79	4	MinerRay
fb15929f	77,054	33,562	112	4	MinerRay
7c36f462	121,931	55,839	79	4	MinerRay
89a3645c	75,003	34,134	58	2	MinerRay
dceaf65b	77,575	34,901	58	2	MinerRay
089dd312	79,883	34,989	58	2	MinerRay
e09c32c5	71,955	32,416	46	1	MinerRay

[a] Yara project https://github.com/davbo/yara-rs/tree/master/sample-miners.
[b] All Wasm binaries of the MinerRay project could be found at https://github.com/miner-ray/miner-ray.github.io/tree/master/Data/SampleWasmFiles.
[c] All Wasm binaries of the SEISMIC project could be found at https://github.com/wenhao1006/SEISMIC.
[d] Deepminer project https://github.com/deepwn/deepMiner.

the identifier of the WebAssembly binary which is the sha256 hash of its bytestream, its size in bytes, the number of instructions, the number of functions defined inside the binary and the number of VirusTotal vendors that detect the binary. The last column contains the origin of the binary according to the wasmbench dataset.[1]

The programs include between 30 and 70 functions, for a total number of instructions ranging from 30,531 to 55,839. The size of the programs ranges from 62 to 103 kilobytes. These binaries are detected as malicious by at least 1 antivirus, and at most 31. We have observed that 6 out of 33 binaries can be executed end-to-end.

To validate that the detected binaries are cryptojacking, we manually analyze each of the 33 binaries identified as malign. First, we observe that the binaries in the dataset originate from two primary sources: project SEISMIC and project MinerRay. SEISMIC (Wang et al., 2018) is a research project about instrumentation and monitoring at runtime to detect cryptojacking binaries. MinerRay is also a research project to detect crypto mining processes in web browsers (Romano et al., 2020). Both projects have collected the binaries as real cryptojacking from the web and the dataset is a union of them.

Second, we observee that all binaries share code from cryptonight (XMRIG, 2016), which is a library for cryptomining hashing. This observation is consistent with the findings of Romano et al. (2020). We find 5 binaries that are multivariant packages (Cabrera Arteaga et al., 2022) of cryptopnight. A multivariant package is a binary containing more than one hashing function. Concretely, the binaries `0d996462`, `d2141ff2`, `046dc081`, `a32a6f4b` and `fbdd1efa` contain between 2 and 3 versions of hashing functions `cryptonight_hash`.

Third, our manual analysis of the binaries reveals 6 main sources for these differences. (1) **Versions:** the binaries do not depend on the same version of cryptonight, (2) **Function reordering:** The order in which the functions are declared inside the binary changes (3) **Innocuous expressions:** Expressions have been injected into the program code, but their execution does not affect the semantic of the original program (4) **Function renaming:** The name of the functions exported to JavaScript have been changed (5) **Data layout changes:** The data needed by the cryptominers has different location in the WebAssembly linear memory. (6) **Partial cryptonight:** Some binaries exclude cryptonight functions, i.e., the `cryptonight_create`, `cryptonight_destroy`, `cryptonight_hash` functions. It is interesting to note that changes in function order, function names, or data layout leads to different programs that have the same number of instructions and functions, as is the case for 9 of our malign binaries.

5.3. Methodology

Based on our dataset of WebAssembly binaries, we follow the following procedures to answer our research questions.

RQ1: Evasion effectiveness To answer RQ1, we execute the baseline evasion algorithm discussed in Section 4.3. We first pass a suspicious binary to wasm-mutate. The diversified version is passed to VirusTotal as a binary oracle. If at least one vendor still detects the diversified binary, we pass it to wasm-mutate again to stack a new random transformation. We repeat this process until VirusTotal does not label the binary as malware or reach a limit of 1000 transformations. The process is performed 10 times with 10 different random seeds for each binary.

RQ2: Oracle minimization Malicious actors have a limited budget to perform evasion before they get caught. The number of oracle calls is a proxy for such a budget, i.e., the lesser the number of oracle calls, the lesser effort spent. To answer RQ2, we aim at minimizing the number of oracle calls while keeping the same evasion effectiveness. In particular, we assess the capability of the MCMC evasion algorithm to minimize the number of calls to the malware oracle (Metric 1). For this, we execute the MCMC evasion algorithm (Algorithm 2) one time for each malware binary of our dataset. We use VirusTotal as an oracle and stop when we reach a limit of 1000 transformations. Since MCMC has a configuration parameter σ, we repeat the process with three σ values: 0.01, 0.3, and 1.1. The σ-value weights exploration and exploitation of transformations during the evasion process. For example, the first value is low, favoring exploration at the most, meaning that the MCMC algorithm will take any new transformation, whether it increases the fitness function value or not. On the contrary, the largest value 1.1 favors the exploitation and, meaning that during evasion, MCMC will only accept transformations with higher values from the oracle, i.e., more evaded detectors. We manually select the third value 0.3 as the balance between exploration and exploitation.

RQ3: Malware functionality A diversified binary that fully evades the detection, might not be practical due to behavioral or performance issues. RQ3 complements our first two research questions with a correctness and an efficiency evaluation. For every cryptojacking that can be executed, we reproduce all cryptominer components (described in Section 2.2) and replace the WebAssembly binaries with variants that fully evade VirusTotal. For each executable cryptojacking program, we generate 10 variants with the baseline evasion algorithm as well as 10 variants with the MCMC algorithm, with σ value 0.3 to balance the MCMC exploration-exploitation. Then, we replace the original cryptojacking by each of the 20 variants in order to determine that the behavior of the original cryptojacking program is preserved in the variants (Demetrio et al., 2021).

Our end-to-end pipelines provide data on the number of generated hashes in the webpage component as an HTML element. Besides, the number of successful or incorrect jobs are logged by the miner pool. This information can be used to measure both the correctness and efficiency of the generated WebAssembly variants. To collect data on the number of hashes per second, the webpage is accessed with Puppetter, while the miner pool logs are saved to measure the number of successful and failed jobs with their respective log time. By analyzing these two types of data, the overall correctness and effectiveness of a diversified WebAssembly binary can be determined.

To check correctness, we verify that the hashes generated by the variants are valid. We determine whether the hashes reach the third component of a cryptojacking (see Section 2.2), i.e., how many successful and failed jobs are reported by the miner pool. If a miner pool correctly accepts the hashes, then the cryptojacking variant generated by our evasion algorithms is considered correct.

To check for efficiency, we measure the frequency of `hashes` produced by the variants binaries. For each WebAssembly cryptojacking and its generated variants, we execute and measure the hashes produced per second, during 1000 s. For each malware variant, we check if the hash production frequency is still in the same order of magnitude as the original.

[1] Binaries that could be found in a live webpage at the moment of this writing are marked with ● http:// .

[2] Yara project `https://github.com/davbo/yara-rs/tree/master/sample-miners`

[3] All Wasm binaries of the MinerRay project could be found at `https://github.com/miner-ray/miner-ray.github.io/tree/master/Data/SampleWasmFiles`

[4] All Wasm binaries of the SEISMIC project could be found at `https://github.com/wenhao1006/SEISMIC`

[5] Deepminer project `https://github.com/deepwn/deepMiner`

RQ4: Individual transformation effectiveness As discussed in Section 4.1, our diversifier comprises 135 possible transformation operators. In this research question, we want to study which transformations perform better in evading the VirusTotal malware detection oracle. This investigation will help future researchers and detectors engineers to improve their detection methods and tackle subversive transformations.

We use a value of $\sigma = 1.1$ to tune the MCMC evasion algorithm. With this parameter, the MCMC evasion algorithm only keeps transformations that significantly contribute to improving the fitness of the variant. In other words, a transformation is applied if at least one more detector of VirusTotal is evaded. Then, we count the number of applied transformations, aggregated by its type. By measuring this, we obtain the most used one (resp. the least used), and, therefore, understand where malware researchers should focus for counter-evasion.

RQ5: Effectiveness against MINOS This research question assesses the effectiveness of our evasion technique with respect to the state-of-the-art WebAssembly malware detector, MINOS (Naseem et al., 2021). This detector takes a Wasm binary as input and creates a 100×100 grayscale image from its pure byte stream. Using a Convolutional Neural Network, the generated image is classified as benign or malware.

We replicate MINOS. However, the model of MINOS is not publicly available, so we train our own model for this experiment. We use our own dataset to train the model with 33 malign programs and 33 benign programs. The 33 malign programs for training MINOS are the same listed in Table 1, the 33 benign programs are collected from the original MINOS reproduction steps. Our reproduction of MINOS achieves the same results as the original paper, based on one-off validation. Our reproduction of MINOS is publicly available at https://github.com/ASSERT-KTH/ralph.

We use MINOS as an oracle and follow the same method proposed in RQ1, passing each one of the binaries in Table 1 to our diversifier and then to MINOS. For each binary, we do the process 10 times with 10 different seeds, generating a total of 330 variants with no more than 1000 stacked mutations.

5.4. Metrics

In this section, we define the notions of total and partial evasion used in this work to measure the impact of the evasion algorithms proposed in Section 4.3. Besides, we also define the `number of oracle calls` and `number of stacked transformations` metrics. In addition, we define the metrics for correct hashes generated by the malware variants and the hashes generation speed.

Definition 1. Total evasion: Given a malware WebAssembly binary, an evasion algorithm generates a variant that totally evades detection if *all* detectors that originally identify the binary as malware identify the variant as benign.

Definition 2. Partial evasion: Given a malware WebAssembly binary, an evasion algorithm generates a variant that partially evades detection if *at least one* detector that originally identifies the binary as malware identifies the variant as benign.

Metric 1. Number of oracle calls: The number of calls made to the malware oracle during the evasion process.

Metric 2. Number of stacked transformations: The total number of transformations applied on the initial malware binary during the evasion process.

Notice that Metric 1 is the number of times that lines 4 and 5 in Algorithms 1 and 2 are executed, respectively. The same could be applied to Metric 2, in lines 7 and 13 of Algorithms 1 and 2, respectively.

The main purpose of a WebAssmbly cryptominer is to calculate hashes. By measuring the number of calculated hashes per time unit, we can measure how performant the cryptojacking is. Therefore, the impact of the evasion process over the performance of the created binary can be measured, calculating the number of hashes per time unit:

Metric 3. Crypto hashes per second (h/s): Given a WebAssembly cryptojacking, the crypto hashes per second metric is the number of successfully generated hashes in one second.

Metric 4. Correct crypto hashes: Given a WebAssembly cryptojacking, the number of correct crypto hashes is the number of hashes that the WebAssembly cryptojacking generates and that the miner pool accepts as valid.

6. Experimental results

In section, we answer our four research questions regarding the feasibility of WebAssembly diversification for malware evasion.

6.1. RQ1. Evasion effectiveness

We run our baseline evasion algorithm with a limit of 1000 iterations per binary. At each iteration, we query VirusTotal to check if the new binary evades the detection. This process is repeated with 10 random seeds per binary, resulting in a maximum of 10,000 queries per original binary. In total, we generate 98,714 variants for the original 33 suspicious binaries.

Table 2 shows the data to answer RQ1. The table contains as columns: the hash of the program, calculated as the sha256 hash, as its identifier, the number of initial VirusTotal detectors flagging the malware, the number of evaded antivirus vendors (cf. Definition 2) and the mean number of iterations needed to generate a variant that fully evades the detection (cf. Definition 1). The rows of the table are ordered with respect to the number of detectors for the original binary.

We observe that the baseline evasion algorithm successfully generates variants that totally evade detection for 30 out of 33 binaries. The mean value of iterations needed to generate a variant that evades all detectors ranges from 120 to 635 stacked transformations. For the 30 binaries that completely evade detection, we observe that the mean number of iterations to evade is correlated to the number of initial detectors. For example, the `a32a6f4b` binary, initially flagged by 18 detectors, requires around 635 iterations, while the `309c32c5`, with only one initial flag, needs 120 iterations. The mean number of iterations needed is always less than 1000 stacked transformations.

Figure 3 shows the evasion process with four different seeds for the binary `046dc081`. Each point in the *x*-axis represents 50 iterations, and the *y*-axis represents the number of VirusTotal detectors flagging the binary. Three out of 4 seeds manage to totally evade VirusTotal in less than 250 iterations. We have observed that there are better evasion techniques than pure random transformations. For example, the seed represented by the green line partially evades the oracle but shows no tendency to evade detection before 300 iterations. Besides, some transformations help some classifiers to detect the mutated binary. These phenomena are empirically exemplified in Fig. 3 in which the curves is not always monotonously decreasing, like the blue-colored curve. In this case, it goes from 3 VirusTotal detectors to 5 during the 50–100 iterations.

There are 3 binaries for which the baseline algorithm does not completely evade the detection. In these three cases, the algorithm misses 5 out 31, 6 out of 30 and 5 out 26 detectors. The explanation is the maximum number of iterations (1000) we use for our

Table 2
Baseline evasion algorithm for VirusTotal. The table contains as columns: the hash of the program, the number of initial VirusTotal detectors, the maximum number of evaded antivirus vendors and the mean number of iterations needed to generate a variant that fully evades detection. The rows of the table are sorted by the number of initial detectors, from left to right and top to bottom.

Hash	#D	Max. #evaded	Mean #trans.
47d29959	31	26 (83.8%)	N/A
9d30e7f0	30	24 (80.0%)	N/A
8ebf4e44	26	21 (80.7%)	N/A
dc11d82d	20	20 (100.0%)	355
0d996462	19	19 (100.0%)	401
a32a6f4b	18	18 (100.0%)	635
fbdd1efa	18	18 (100.0%)	310
d2141ff2	9	9 (100.0%)	461
aafff587	6	6 (100.0%)	484
046dc081	6	6 (100.0%)	404
643116ff	6	6 (100.0%)	144
15b86a25	4	4 (100.0%)	253
006b2fb6	4	4 (100.0%)	282
942be4f7	4	4 (100.0%)	200
7c36f462	4	4 (100.0%)	236
fb15929f	4	4 (100.0%)	297
24aae13a	4	4 (100.0%)	252
000415b2	3	3 (100.0%)	302
4cbdbbb1	3	3 (100.0%)	295
65debcbe	2	2 (100.0%)	131
59955b4c	2	2 (100.0%)	130
89a3645c	2	2 (100.0%)	431
a74a7cb8	2	2 (100.0%)	124
119c53eb	2	2 (100.0%)	104
089dd312	2	2 (100.0%)	153
c1be4071	2	2 (100.0%)	130
dceaf65b	2	2 (100.0%)	140
6b8c7899	2	2 (100.0%)	143
a27b45ef	2	2 (100.0%)	145
68ca7c0e	2	2 (100.0%)	137
f0b24409	2	2 (100.0%)	127
5bc53343	2	2 (100.0%)	118
e09c32c5	1	1 (100.0%)	120

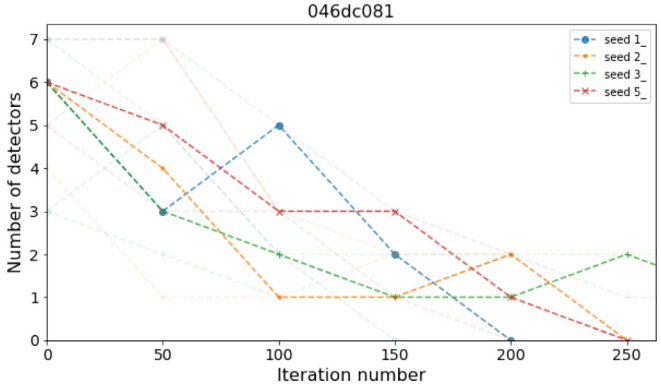

Fig. 3. The figure shows the evasion process with four seeds for binary `046dc081`. Each point in the *x*-axis represents 50 iterations, and the *y*-axis represents the number of VirusTotal detectors.

experiments. However, having more iterations seems not a realistic scenario. For example, if some transformations increment the binary size during the transformation, a considerably large binary might be impractical for bandwidth reasons.

On the other hand, there is a balance between generating variants and avoiding detection by defense mechanisms. For example, VirusTotal detects when it is being stressed with too many requests and too many queries can be detected and blocked, effectively ruining evasion. In our attack scenario, where VirusTotal is used as an external detector (see Section 3), we must be mindful of this limit. In contrast, when defense mechanisms such as MINOS are placed locally, we believe that the number of queries is not necessarily limited.

Wasm-mutate performs mutations based on the input binary. In the experiments for RQ1, the input binaries for the baseline algorithm comes from the application of a previous mutation. Yet, we have observed that some transformations can be applied in any order. This means that different sequences of transformations can produce the same binary variant. This often happens when two mutation targets inside the binary are different, such as two disjoint pieces of code. Therefore, a potential parallelization for the baseline algorithm is possible as soon as transformation sequences do not interfere with others.

Overall, our experiments prove that wasm-mutate is a powerful tool to perform malware evasion. By carefully selecting the order and type of transformations applied, it is possible to generate program variants that are both performant and effective at evading detection. This same idea is explored in the next section.

Answer to RQ1: The baseline evasion algorithm with wasm-mutate clearly decrease the detection rate by VirusTotal antivirus vendors for cryptojacking malware. We achieve total evasion of WebAssembly cryptojacking malware in 30/33 (90%) of our malware dataset.

6.2. RQ2. Oracle minimization

With RQ2, we analyze the effect of the MCMC evasion algorithm in minimizing the number of calls to the malware oracle (Metric 1). To answer RQ2, we execute the MCMC evasion algorithm discussed in Algorithm 2.

In Table 3 we can observe the impact of the MCMC evasion algorithm on our reference dataset. The first two columns of the table are the original program's hash and the number of initial VirusTotal detectors flagging the malware. The remaining columns are divided into two categories, maximum detectors evaded (Definition 2) and the number of oracle calls if total (Definition 1). Each one of the two categories contains the result for both evasion algorithms, first the baseline algorithm (BL) followed by the three σ-values analyzed in the MCMC evasion algorithm. Notice that, for the baseline algorithm, the number of oracle calls is the same value as the number of transformations needed to evade by construction. We highlight in bold text the values for which the baseline or the MCMC evasion algorithms are better than each other, the lower, the better.

We observe that the MCMC evasion algorithm successfully generates variants that totally evade the detection for 30 out of 33 binaries, it thus as good as the baseline algorithm. The improvement happens in the number of oracle calls. The oracle calls needed for the MCMC evasion algorithm are 92% of the needed on average for the baseline evasion algorithm.

For 21 of 30 binaries that evade detection entirely, we observe that the mean number of oracle calls needed is lower than those in the baseline evasion algorithm. For example, `f0b24409` needs 11 oracle calls with the MCMC evasion algorithm to fully evade VirusTotal, while for the baseline evasion algorithm, it needs 127 oracles calls. For those 21 binaries, it needs only 40% of the calls the baseline evasion algorithm needs.

The impact of the MCMC evasion algorithm is illustrated in Fig. 4. Each point in the *x*-axis represents 50 iterations, and the *y*-axis represents the number of VirusTotal detectors flagging binary 046dc081. 2 out of 3 σ-values manage to totally evade VirusTotal in less than 400 iterations. On the contrary, lower acceptance criteria $\sigma = 1.1$ (green line) partially evades the oracle, but does not fully evade within the maximum 1000 iterations limit of the experiment.

The σ value in the Algorithm 2 provides the acceptance criteria for new transformations in the MCMC evasion algorithm.

Table 3
MCMC evasion algorithm for VirusTotal. The first two columns of the table are: the identifier of the original program and the number of initial detectors. The remaining columns are divided into two categories, maximum detectors evaded if partial evasion and number of oracle calls if total evasion. Each one of the two categories contains the result of the evasion algorithms, first the baseline algorithm (BL) followed by the three σ-values analysed in the MCMC evasion algorithm. We highlight in bold text the values for which the baseline or the MCMC evasion algorithms are better from each other. Overall, the MCMC evasion algorithm needs less oracle calls than the baseline algorithm.

Hash	#D	Max. evaded				#Oracle calls			
		BL	MCMC			BL	MCMC		
			$\sigma = 0.1$	$\sigma = 0.3$	$\sigma = 1.1$		$\sigma = 0.01$	$\sigma = 0.3$	$\sigma = 1.1$
47d29959	31	**26**	19	12	10				
9d30e7f0	30	**24**	17	9	10				
8ebf4e44	26	**21**	13	5	4				
dc11d82d	20	**20**	**20**	14	15	**355**	446		
0d996462	19	**19**	**19**	14	4	**401**	697		
a32a6f4b	18	**18**	**18**	6	1	635	**625**		
fbdd1efa	18	**18**	**18**	3	3	**310**	726		
d2141ff2	9	**9**	**9**	**9**	5	**461**	781	783	
aafff587	6	**6**	2	**6**	**6**	484		**331**	413
046dc081	6	**6**	**6**	**6**	5	404	397	**159**	
643116ff	6	**6**	**6**	**6**	1	**144**	436	631	
15b86a25	4	**4**	**4**	**4**	**4**	253	208	214	**131**
006b2fb6	4	**4**	**4**	**4**	0	**282**	380	709	
942be4f7	4	**4**	**4**	**4**	**4**	**200**	**200**	**200**	219
7c36f462	4	**4**	**4**	2	0	236	**221**		
fb15929f	4	**4**	2	**4**	2	**297**		475	
24aae13a	4	**4**	**4**	**4**	0	**252**	401	446	
000415b2	3	**3**	**3**	**3**	2	302	376	**34**	
4cbdbbb1	3	**3**	**3**	**3**	**3**	295	204	**72**	685
65debcbe	2	**2**	**2**	**2**	**2**	131	**33**	**33**	**33**
59955b4c	2	**2**	**2**	**2**	**2**	130	**33**	**33**	**33**
89a3645c	2	**2**	**2**	**2**	**2**	431	319	197	**107**
a74a7cb8	2	**2**	**2**	**2**	**2**	124	**33**	**33**	**33**
119c53eb	2	**2**	**2**	**2**	**2**	104	45	480	**18**
089dd312	2	**2**	**2**	**2**	**2**	153	166	167	**123**
c1be4071	2	**2**	**2**	**2**	**2**	130	**33**	**33**	**33**
dceaf65b	2	**2**	**2**	**2**	**2**	140	166	166	**132**
6b8c7899	2	**2**	**2**	**2**	**2**	143	**33**	**33**	**33**
a27b45ef	2	**2**	**2**	**2**	**2**	145	**33**	**33**	**33**
68ca7c0e	2	**2**	**2**	**2**	**2**	137	**33**	167	595
f0b24409	2	**2**	**2**	**2**	**2**	127	**11**	33	**11**
5bc53343	2	**2**	**2**	**2**	**2**	118	**33**	**33**	**33**
e09c32c5	1	**1**	**1**	**1**	0	**120**	488	921	

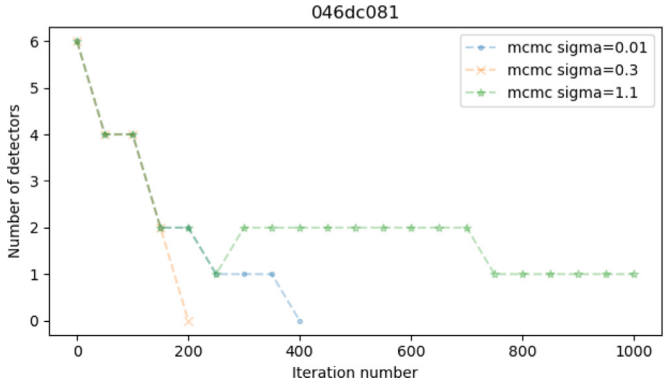

Fig. 4. The figure shows the MCMC evasion process for the binary `046dc081`. Each point in the *x*-axis represents 50 iterations, and the *y*-axis represents the number of VirusTotal detectors. The figure shows less chaotic progress for oracle evasion.

We run the MCMC evasion algorithm with three values to better understand the extent to which the transformation space is explored to find a binary that evades successfully. We have observed that for a large value of σ, meaning low acceptance, 16 binaries cannot be mutated to evade VirusTotal entirely. The main reason is that, in this case, the MCMC evasion algorithm discards transformations that increase the number of oracle calls. Therefore, the algorithm gets stuck in local minima and never finds a binary that entirely evades. On the contrary, a small value of σ accepts more new transformations even if more detectors flag the binary.

According to our experiments, $\sigma = 0.3$ offer a good acceptance trade-off. However, the best value of σ actually depends on the binary to mutate. Therefore, we cannot conclude the best value of σ for the whole dataset. This is a consequence of the particularities of each one of the original binaries and its detectors. For example, we have observed that for the bottom part of Table 3 the highest value of σ works better overall. The main reason is the low number of original detectors. In those cases, the exploration space for transformations is smaller. Thus, for the MCMC evasion algorithm, the chances of a local minimum for the fitness function to be global are larger. Therefore, the MCMC evasion algorithm with low acceptance criteria can find the binary that fully evades in fewer iterations. On the contrary, if the number of initial detectors is more significant, the exploration space is too big to explore in 1000 max iterations. The reason is that the MCMC evasion algorithm applies one transformation per time. While we provide fine-grained analysis in our work, more than one transformation per iteration could be applied in the MCMC evasion algorithm to solve this.

In all 3 cases for which neither the baseline nor the MCMC evasion algorithms could find a binary that fully evades, the maximum evaded detectors are less for the MCMC evasion algorithm. The main reason is that the MCMC evasion algorithm might prevent transformations for which the number of detectors increases. As previously discussed, it is stuck in local minima, which means that it does not explore transformation paths for which a higher

Table 4
Malware correctness and efficiency. We execute each cryptojacking that can be reproduced, with the original malware, and with 10 baseline variants and 10 MCMC variants. The first left section indicates the identifier of the original binary and its original frequency of hash generation. The second section of the table shows correctness percentage and hashes per second for ten variants generated with our baseline algorithm. The third section of the table shows correctness percentage and hashes per second for ten variants generated with the MCMC algorithm. After each frequency of hashing measurement, we indicate the relative difference with the original frequency, in parentheses. A difference larger or equal to 1.0 indicates a variant that is faster or as efficient as the original; a difference lower than 1.0 is a variant slower than the original.

Hash	Original h/s	Baseline algorithm								MCMC algorithm							
0d996462	116.0	100%	25 (0.22)	100%	24 (0.21)	100%	26 (0.22)	100%	116 (**1.00**)	100%	70 (0.60)	100%	67 (0.58)				
		100%	116 (**1.00**)	100%	110 (0.95)	100%	30 (0.26)	100%	110 (0.95)	100%	76 (0.66)	100%	60 (0.52)				
		100%	55 (0.47)	100%	27 (0.23)	100%	23 (0.20)	100%	86 (0.74)	100%	60 (0.52)	100%	72 (0.62)				
		100%	27 (0.23)					100%	76 (0.66)								
a32a6f4b	48.0	100%	25 (0.52)	100%	24 (0.50)	100%	24 (0.50)	100%	26 (0.54)	100%	45 (0.94)	100%	41 (0.85)				
		100%	26 (0.54)	100%	25 (0.52)	100%	26 (0.54)	100%	46 (0.96)	100%	41 (0.85)	100%	45 (0.94)				
		100%	26 (0.54)	100%	24 (0.50)	100%	25 (0.52)	100%	44 (0.92)	100%	42 (0.88)	100%	45 (0.94)				
		100%	23 (0.48)					100%	45 (0.94)								
fbdd1efa	37.0	100%	25 (0.68)	100%	25 (0.68)	100%	25 (0.68)	100%	28 (0.76)	100%	47 (**1.27**)	100%	48 (**1.30**)				
		100%	25 (0.68)	100%	26 (0.70)	100%	26 (0.70)	100%	47 (**1.27**)	100%	47 (**1.27**)	100%	53 (**1.43**)				
		100%	25 (0.68)	100%	25 (0.68)	100%	25 (0.68)	100%	48 (**1.30**)	100%	48 (**1.30**)	100%	49 (**1.32**)				
		100%	25 (0.68)					100%	47 (**1.27**)								
d2141ff2	113.0	100%	54 (0.48)	100%	55 (0.49)	100%	55 (0.49)	100%	107 (0.95)	100%	107 (0.95)	100%	107 (0.95)				
		100%	57 (0.50)	100%	56 (0.50)	100%	56 (0.50)	100%	109 (0.96)	100%	106 (0.94)	100%	100 (0.88)				
		100%	57 (0.50)	100%	53 (0.47)	100%	53 (0.47)	100%	101 (0.89)	100%	100 (0.88)	100%	107 (0.95)				
		100%	55 (0.49)					100%	107 (0.95)								
046dc081	118.0	100%	58 (0.49)	100%	60 (0.51)	100%	59 (0.50)	100%	118 (**1.00**)	100%	120 (**1.02**)	100%	119 (**1.01**)				
		100%	60 (0.51)	100%	55 (0.47)	100%	62 (0.53)	100%	120 (**1.02**)	100%	116 (0.98)	100%	120 (**1.02**)				
		100%	55 (0.47)	100%	50 (0.42)	100%	57 (0.48)	100%	119 (**1.01**)	100%	120 (**1.02**)	100%	119 (**1.01**)				
		100%	55 (0.47)					100%	120 (**1.02**)								
006b2fb6	8.0	100%	7 (0.88)	100%	6 (0.75)	100%	4 (0.50)	100%	6 (0.75)	100%	6 (0.75)	100%	6 (0.75)				
		100%	9 (**1.12**)	100%	6 (0.75)	100%	4 (0.50)	100%	6 (0.75)	100%	6 (0.75)	100%	6 (0.75)				
		100%	4 (0.50)	100%	6 (0.75)	100%	4 (0.50)	100%	8 (**1.00**)	100%	9 (**1.12**)	100%	6 (0.75)				
		100%	6 (0.75)					100%	6 (0.75)								

number of detectors could lead to better long-term results and, eventually, full evasion.

Answer to RQ2: The MCMC evasion algorithm needs fewer oracle calls than the baseline algorithm. In 21 cases out of 33, it needs only 40% of oracle calls compared to the baseline, providing more stealthiness to the malicious organization directing the evasion. The acceptance criterion σ of the MCMC evasion algorithm needs to be carefully crafted depending on the original binary.

6.3. RQ3. Malware functionality

To answer RQ3, we select the six binaries we can build and execute end-to-end, because we have access to the three components previously mentioned in Section 2.2. For those six binaries, we are able to replace the original WebAssembly binary with variants generated by our evasion algorithms.

We execute the original binary and the variants generated by the baseline and MCMC evasion algorithms. The essence of a cryptojacking is to generate hashes at high-speed. Consequently, we assess the correctness of the variants concerning two properties: validity of the hashes and frequency of hash generation. With Metric 4, we determine the validity of the generated hashes by checking if the backend miner pool accepts them. The frequency is measured as the amount of `hashes` produced by the variant binaries in one second (Metric 3).

Table 4 summarizes the key data for RQ3. Each row of the table corresponds to one binary that can be executed end-to-end, by reproducing the three components mentioned in Section 2.2. The first two values of each row are the original binary's identifier and its original frequency of hash generation. Then, for our baseline algorithm, each row has the data for 10 variants generated during a successful oracle evasion. For each one of the variants, we include the correctness percentage and the hashes calculated per second. The last group of columns of the table contains the correctness percentage and the hashes calculated per second for the 10 variants generated with the MCMC algorithm. After each frequency of hashing measurement, we indicate the relative difference with the original frequency in parentheses. A difference larger or equal to 1.0 means that the variant is faster or as efficient as the original. On the contrary, the variant is slower if the difference is lower than 1.0. The table contains the information for 120 variants.

We use the backend miner pools (see Section 2.2) of the six cryptojacking to determine the validity of the hashes computed by all the programs. All correctness assessments for the 120 variants indicate that the miner pools do not detect any invalid hash when executing the WebAssembly cryptominer variants. This means that the variants synthesized by the baseline and the MCMC algorithms to evade the malware oracle can still systematically generate valid hashes.

We have observed that 23 of 120 variants are more efficient than the original cryptojacking. For example, for the `fbdd1efa` binary, all its variants are from 1.27 to 1.43 faster than the original hash generation frequency. This phenomenon occurs because wasm-mutate can perform transformations in the executable code, which work as optimizations. Our experiments have revealed two sources of faster WebAssembly variants: loop unrolling transformations and code replacements that lead to smaller binaries. We have also found that debloating transformations, which remove unneeded structures and dead code, results in more hashes being produced by the cryptominer in the first few seconds of mining, likely because of faster compilation.

In summary, all this is evidence that focused optimization is a good primitive for evasion. On the contrary, 97 out of 120 variants underperform compared to the original binary. The worst case is the binary `0d996462`. Its slowest variant has 0.20 of the original generation frequency. The main reason is that wasm-mutate also introduces non-optimal transformations regarding performance.

The variants generated by the baseline evasion algorithm tend to be slower than the MCMC evasion algorithm. The MCMC evasion algorithm triggers fewer oracle calls and can generate variants faster than the ones generated by the evasion algorithm. This phenomenon is a direct consequence of the MCMC evasion algo-

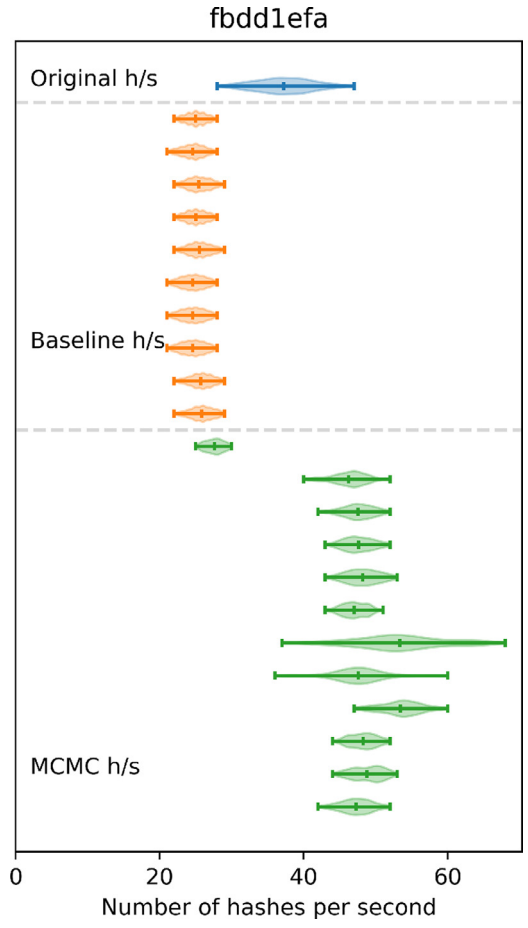

Fig. 5. Distribution of the number of hashes per second for the variants generated for the original `fbdd1efa` cryptojacking. In the figure we include the original binary (in blue), ten variants generated by the baseline algorithm (in orange), and ten variants generated by the MCMC evasion algorithm (in green). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

rithm implementation, which has a selective strategy when applying transformations on a binary (see Algorithm 2). In summary, the MCMC evasion algorithm produces variants that fully evade the VirusTotal oracle with lower performance overhead during execution. The worst performant variant is only 1.93 times slower for the MCMC evasion algorithm.

In Fig. 5, we plot the distribution of the hashes per second for variants generated for the `fbdd1efa` cryptojacking. In the figure, we include the original binary (in blue), the ten variants generated by the baseline algorithm (in orange), and the ten variants generated by the MCMC evasion algorithm (in green). Each violin plot corresponds to the hashes per second. We observe a normal distribution around the exact number of hashes per second. In this case, we have observed that the MCMC evasion algorithm provides a hash-per-second ratio better than the original. This phenomenon can be observed in the lines inside the violin plots, the green line is shifted to the right compared to the lines of the blue violin plot.

Answer to RQ3: Our algorithms synthesize WebAssembly cryptojacking variants that fully evade our malware oracle and that provide the same functionality as the original. The execution of evading malware systematically produces valid hashes, and the variations in performance are imperceptible. For 19% of the generated variants, we observe better performance, and in the worst case, the generated variant underperforms by five times the original binary.

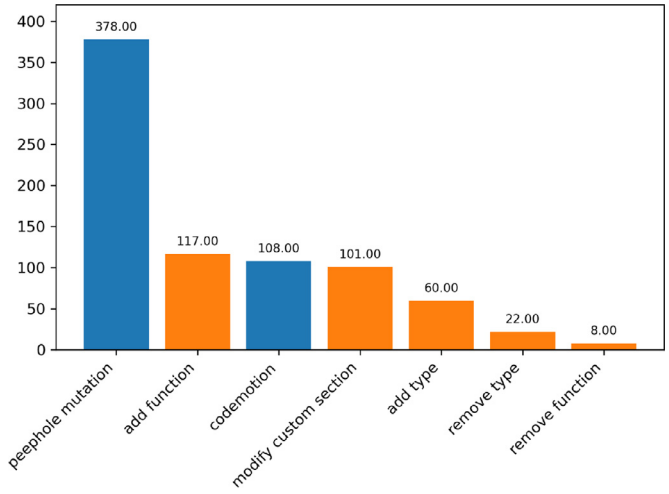

Fig. 6. Distribution of applied transformation for the MCMC evasion algorithm with $\sigma = 1.1$ (low acceptance). The x-axis displays the names of the transformations. The y-axis indicates the number of transformations found in the generated variants. We can observe which transformations perform better in order to provide total evasion.

6.4. RQ4. Individual transformation effectiveness

With RQ4, we investigate what individual transformations are the most appropriate to generate evading malware variants. Both attackers and defenders can leverage this information as follows. On the one hand, attackers know which transformation operators they can discard to speed-up the search for evading malware and minimize calls to the oracle. On the other hand, defenders can focus on the most effective transformations to evade antiviruses. For example, one way to defend against evasion is to use transformation as a preprocessing stage prior to detection. This can help to ensure that detection is more robust to potential evasion vectors.

In Fig. 6 we plot the distribution of transformations applied by the MCMC evasion algorithm with $\sigma = 1.1$ when it generates variants that fully evade the VirusTotal oracle. On the x-axis, we provide the name of the transformation, as wasm-mutate implements them. The y-axis is the absolute number of transformations found among the generated variants. The transformations are sorted in decreasing order of usage in the variants. We use two colors for transformations: transformations that affect the execution of the binary (in blue color) and transformations that do not (in orange color). For example, adding a new type definition does not affect the execution of the binary, while a peephole transformation does.

First, we concentrate on non-behavioral transformations in orange. The most effective transformation is to add a random function, which is present 117 times in the evading binaries. The next most present structural transformation is the `modify custom section` transformation. This highlights the sensitivity of malware detectors to the custom section. We note that custom section modification at the bytecode level provides advantages against source code obfuscation because we are sure that the compiler does not add metadata information that would help malware detectors. In other words, metadata information injected by compilers into WebAssembly binaries (Hilbig et al., 2021), could be removed from being part of possible detection.

Transformations `remove function` and `remove type` also affect malware detection. This novel observation indicates that VirusTotal is looking for code common in malware yet dead code. For example, malware detectors probably check for code that does not affect the original functionality of the binary. Thus, if this information is removed, the detector misses the malware. In other terms, by removing code, the detection surface is reduced.

```
...
local.get 0



i32.const 4
i32.add
...
```

Listing 4. Original piece of code for the 089a3645c WebAssembly binary.

```
...
local.get 0
i64.const -461681990785514485
drop
i32.const 0
i32.shr_u
i32.const 4
i32.add
...
```

Listing 5. Peephole mutations of Listing 4. Only with this mutation VirusTotal passed from 2 initial detectors to 0 in our experiments.

The most significant transformation to generate evading malware consists of peephole transformations (the largest bar of the figure at the left-hand side of the figure). The peephole transformations operate at the bytecode instruction level. For example, in Listings 4 and 5 we show a piece of the binary 089a3645c, and one peephole transformation applied, respectively. The transformation in the listings corresponds to a variant generated with the MCMC evasion algorithm. Only this transformation is required to fully evade for the 089a3645c binary.

Our results show that malware detectors should prioritize the detection of peephole transformations in WebAssembly, to increase the likelihood of detecting cryptojacking. For example, the transformation of Listing 5 can be reversed with static analysis.

When we answer our four research questions, we generate WebAssembly cryptominer variants by adding one transformation at a time (See Section 4.3). This method allows us to answer our research questions at a fine-grained level. For instance, the answer to RQ4 could only be possible if the transformations are analyzed one by one in the evasion process. Now, our method can be easily tuned to one-shot evasion: the algorithms could apply multiple transformations simultaneously to produce evading malware in one iteration. Consequently, the evasion process proposed in this work could be faster and more practical for a potential attacker. On the other hand, our algorithms stop as soon as one binary is diversified enough to provide total evasion. Since the overhead introduced by wasm-mutate is imperceptible, the transformation process can generate remarkably more binaries. Our approaches could escalate to infinite cryptojacking variants.

Answer to RQ4: Our experiments reveal that peephole transformations are the most effective for WebAssembly cryptojacking malware evasion. We also show that transformations on non-executable parts of WebAssembly binaries can contribute to evasion. These novel observations are crucial for cryptojacking malware detector vendors to prioritize their work on improving malware detection.

6.5. RQ5. Effectiveness against MINOS

To evaluate the effectiveness of MINOS at detecting diversified malware, we reuse the protocol of RQ1. We repeatedly stack random mutations to the original malware binary until MINOS is fully evaded or the maximum number of iterations is reached. We repeat this process 10 times for each binary. The results of our experiment are presented in Table 5. The table provides the identifier of the program and the mean number of iterations required to synthesize a variant that fully evades MINOS.

Table 5
The table provides the identifier of the program as its sha256 hash value and the mean number of iterations required to totally evade MINOS. Each binary was mutated with the baseline algorithm 10 times.

Hash	Mean #trans.	Hash	Mean #trans.
24aae13a	980.0	000415b2	960.0
59955b4c	38.0	119c53eb	1.0
fb15929f	1.0	5bc53343	33.0
47d29959	100.0	dc11d82d	115.0
a27b45ef	33.0	006b2fb6	1.0
942be4f7	29.0	7c36f462	85.0
0d996462	24.0	15b86a25	1.0
8ebf4e44	92.0	a74a7cb8	38.0
fbdd1efa	1.0	089dd312	68.0
65debcbe	38.0	aafff587	1.0
046dc081	33.0	6b8c7899	38.0
a32a6f4b	1.0	d2141ff2	81.0
68ca7c0e	38.0	dceaf65b	66.0
9d30e7f0	419.0	4cbdbbb1	1.0
643116ff	47.0	c1be4071	38.0
e09c32c5	15.0	f0b24409	33.0
89a3645c	108.0		

Our technique completely evades MINOS in all cases. In 2 cases out of 33, wasm-mutate needs more than 900 iterations to evade MINOS. The main reason is the application of symmetric mutations. For example, in some cases, wasm-mutate performs mutations that copy parts of the binary to another program location. Thus, when the binary is turned into a grayscale image, the embedding of the image is preserved, i.e., the code has changed, but the shape of the image has not. The contrary happens when non-symetric mutations are applied. For example, removing functions also removes embeddings of the grayscale image used by MINOS.

Remarkably, this experiment shows that WebAssembly diversification requires fewer iterations to evade MINOS than VirusTotal, meaning that it is easier to evade MINOS. The minimum number of iterations needed overall for evading VirusTotal are 118 for the baseline algorithm Table 2 and 11 for the MCMC algorithm Table 3, while for MINOS, wasm-mutate totally evades detection for 8 out of 33 binaries in one single iteration. This shows that the MINOS model is fragile wrt binary diversification. According to those results, VirusTotal can be considered better than MINOS wrt to cryptojacking detection.

To further enhance the detection capabilities of MINOS, we believe in binary canonicalization (Bruschi et al., 2007). By creating a canonical representation of the malware variant before training and inference, one would help the classifier to better generalize. This is feasible as it is a preprocessing step in the pipeline. We believe this is an interesting direction for future work.

Answer to RQ5: Our approach fully evades detection by the WebAssembly antivirus MINOS. In our study, we achieve evasion for all cryptojacking binaries in our dataset. wasm-mutate needs fewer iterations to totally evade MINOS compared to VirusTotal, validating VirusTotal as a baseline.

7. Discussion

In this section, we discuss the key challenges we faced, in order to help future research projects on similar topics.

Dataset size The dataset is smaller than other similar works for malware evasion. However, the related work does not consider WebAssembly – e.g. Ling et al. (2023) focus on Windows. For example,

while Tekiner and colleagues consider cryptojacking (Tekiner et al., 2021), we entirely focus only on WebAssembly cryptojacking malware. In this context, to the best of our knowledge, wasmbench is the most complete dataset of WebAssembly binaries. We analyze this dataset through the lens of VirusTotal and systematically extract all the cryptojacking malware it includes. Despite novelty, we acknowledge that the limited size of our malware dataset poses a challenge in terms of the generalizability of our results. Some types of malware might be absent from our dataset. To address this issue, one solution for future work would consist in expanding the dataset by using the inherent diversity found within the popular Cryptonight library. One could utilize the release history of their GitHub repository to compile and mine distinct, yet semantically equivalent malware instances. This approach would entail the exploration of a broader range of variations of cryptojacking malware Yet, this is considered as a new research paper per se as the process of mining and compiling code from a repository's release history is both time-consuming and computationally demanding. This is due to the need to analyze, filter, and compare a vast number of code commits and releases, as well as to validate their semantic equivalence. This process is even more complex in the case of malware reproduction due to the complex architecture in which this type of software operates (cf. Fig. 1.)

VirusTotal observations The final labelling of binaries for VirusTotal vendors is not definitive (Zhu et al., 2020). For example, a VirusTotal vendor could label a binary as benign and change it later to malign after several weeks. This phenomenon creates a time window in which slightly changed binaries (fewer iterations in our case) sometives evade the detection of numerous vendors. Also, we have observed that when our evasion algorithms manage to evade, some VirusTotal vendors result in timeout in several cases. This suggests that the evasion effectiveness is also due to performance constraints on the VirusTotal side.

Lack of abstraction A WebAssembly cryptojacking can only exist with its web complements. As we previously discussed, a browser cryptojacking needs to send the calculated hashes to a cryptocurrency service. This network communication is outside the WebAssembly accesses and needs to be delegated to a JavaScript code. We have observed that, the imports and the memory data of the WebAssembly binaries have a high variability in our dataset. The imported functions from JavaScript change from binary to binary. Their data segments also differ in content and length. This suggests that the whole JavaScript-WebAssembly polyglot package is the right direction for cryptojacking detection.

Mitigation As we noted in our response to RQ4 and RQ5, we believe that code canonicalization and is a promising mitigation technique if they are applied directly to WebAssembly. One way to do this would be to modify a diversifier such as wasm-mutate into a binary optimizer completely based on e-graph. This would provide canonicalization through a compact representation of WebAssembly code. In turn, malware variants with the same ancestor would be more seen as the "same" program, from its canonical representation.

8. Conclusion

We have demonstrated the potential for WebAssembly cryptojacking malware to be diversified and evade detection by leading malware detectors, such as VirusTotal and MINOS. Our generated variants are functional, performant, and do not trigger malware detection. Our evaluation of the technique against 60 state-of-the-art antiviruses through the meta-tool VirusTotal highlights the superiority of meta-antiviruses over single tailored defenses, even when the latter are specifically designed for WebAssembly cryptojacking binaries such as MINOS. By studying effective code transformations for evading cryptojacking detection, our work provides valuable insights and guidance for researchers in the field to better mitigate evasion.

As future work, we will improve the evasion fitness functions by including malware program properties, w.r.t. both evasion and malware execution performance. Some argue that the future of malware detection lies in machine learning. In future work, we believe in using our diversification technique to provide data augmentation for better malware detection.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

Afianian, A., Niksefat, S., Sadeghiyan, B., Baptiste, D., 2019. Malware dynamic analysis evasion techniques: asurvey. ACM Comput. Surv. 52 (6). doi:10.1145/3365001.

Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., Kruegel, C., 2020. When malware is packin' heat; limits of machine learning classifiers based on static analysis features. In: Proc. of NDSS.

Aslan, O.A., Samet, R., 2020. A comprehensive review on malware detection approaches. IEEE Access 8, 6249–6271. doi:10.1109/ACCESS.2019.2963724.

Bhansali, S., Aris, A., Acar, A., Oz, H., Uluagac, A.S., 2022. A first look at code obfuscation for WebAssembly. In: Proc. of Conf. on Security and Privacy in Wireless and Mobile Networks doi:10.1145/3507657.3528560.

Bian, W., Meng, W., Zhang, M., 2020. Minethrottle: defending against wasm in-browser cryptojacking. In: Proceedings of The Web Conference 2020. Association for Computing Machinery doi:10.1145/3366423.3380085.

Bostani, H., Moonsamy, V., 2021. Evadedroid: a practical evasion attack on machine learning for black-box android malware detection. CoRR abs/2110.03301https://arxiv.org/abs/2110.03301.

Botacin, M., Ceschin, F., de Geus, P., Grégio, A., 2020. We need to talk about antiviruses: challenges & pitfalls of AV evaluations. Comput. Secur. 95. doi:10.1016/j.cose.2020.101859.

Botacin, M., Domingues, F.D., Ceschin, F., Machnicki, R., Zanata Alves, M.A., de Geus, P.L., Grégio, A., 2022. Antiviruses under the microscope: a hands-on perspective. Comput. Secur. 112. doi:10.1016/j.cose.2021.102500.

Bruschi, D., Martignoni, L., Monga, M., 2007. Code normalization for self-mutating malware. IEEE Secur. Privacy 5 (2), 46–54. doi:10.1109/MSP.2007.31.

Bytecodealliance, 2021. wasm-mutate. https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate.

Cabrera Arteaga, J., Laperdrix, P., Monperrus, M., Baudry, B., 2022. Multi-variant execution at the edge. In: Proceedings of the 9th ACM Workshop on Moving Target Defense. Association for Computing Machinery doi:10.1145/3560828.3564007.

Cabrera-Arteaga, J., Malivitsis, O.F., Pérez, O.L.V., Baudry, B., Monperrus, M., 2021. Crow: code diversification for WebAssembly. In: Proceedings of MadWEB doi:10.14722/madweb.2021.23xxx.

Castro, R.L., Schmitt, C., Dreo, G., 2019. Aimed: evolving malware with genetic programming to evade detection. In: 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE). IEEE, pp. 240–247.

Chua, M., Balachandran, V., 2018. Effectiveness of android obfuscation on evading anti-malware. In: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy. Association for Computing Machinery doi:10.1145/3176258.3176942.

Cohen, F.B., 1993. Operating system protection through program evolution. Comput. Secur. 12 (6), 565–584.

Dasgupta, P., Osman, Z., 2021. A Comparison of State-of-the-Art Techniques for Generating Adversarial Malware Binaries. arXiv e-printsarXiv:2111.11487.

Demetrio, L., Biggio, B., Lagorio, G., Roli, F., Armando, A., 2021. Functionality-preserving black-box optimization of adversarial windows malware. IEEE Trans. Inf. Forensics Secur. 16, 3469–3478.

Egele, M., Scholte, T., Kirda, E., Kruegel, C., 2008. A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. 44 (2). doi:10.1145/2089125.2089126.

GoogleLLC, 2022. Virustotal enterprise. https://assets.virustotal.com/vt-360-outcomes.pdf.

Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J., 2017. Bringing the web up to speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.

Hastings, W.K., 1970. Monte carlo sampling methods using Markov chains and their applications. Biometrika 57 (1), 97–109. http://www.jstor.org/stable/2334940

Hilbig, A., Lehmann, D., Pradel, M., 2021. An empirical study of real-world WebAssembly binaries: security, languages, use cases. In: Proceedings of the Web Conference 2021.
Kalash, M., Rochan, M., Mohammed, N., Bruce, N.D.B., Wang, Y., Iqbal, F., 2018. Malware classification with deep convolutional neural networks. In: 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), pp. 1–5. doi:10.1109/NTMS.2018.8328749.
Kaspersky, 2022. The state of cryptojacking in the first three quarters of 2022. https://securelist.com/cryptojacking-report-2022/107898/.
Kelton, C., Balasubramanian, A., Raghavendra, R., Srivatsa, M., 2020. Browser-based deep behavioral detection of web cryptomining with coinspy. In: Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2020, pp. 1–12.
Kharraz, A., Ma, Z., Murley, P., Lever, C., Mason, J., Miller, A., Borisov, N., Antonakakis, M., Bailey, M., 2019. Outguard: detecting in-browser covert cryptocurrency mining in the wild. In: The World Wide Web Conference. Association for Computing Machinery doi:10.1145/3308558.3313665.
Konoth, R.K., Vineti, E., Moonsamy, V., Lindorfer, M., Kruegel, C., Bos, H., Vigna, G., 2018. Minesweeper: an in-depth look into drive-by cryptocurrency mining and its defense doi:10.1145/3243734.3243858.
Lachtar, N., Ibdah, D., Khan, H., Bacha, A., 2023. Ransomshield: a visualization approach to defending mobile systems against ransomware. ACM Trans. Priv. Secur. 26 (3). doi:10.1145/3579822.
Le, V., Afshari, M., Su, Z., 2014. Compiler validation via equivalence modulo inputs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery doi:10.1145/2594291.2594334.
Li, D., Li, Q., Ye, Y.F., Xu, S., 2021. Arms race in adversarial malware detection: asurvey. ACM Comput. Surv. 55 (1). doi:10.1145/3484491.
Ling, X., Wu, L., Zhang, J., Qu, Z., Deng, W., Chen, X., Qian, Y., Wu, C., Ji, S., Luo, T., et al., 2023. Adversarial attacks against windows pe malware detection: a survey of the state-of-the-art. Comput. Secur. 103134.
Liu, L., Wang, B., 2016. Malware classification using gray-scale images and ensemble learning. In: 2016 3rd International Conference on Systems and Informatics (ICSAI), pp. 1018–1022. doi:10.1109/ICSAI.2016.7811100.
Lu, G., Debray, S.K., 2013. Weaknesses in defenses against web-borne malware - (short paper). In: Rieck, K., Stewin, P., Seifert, J. (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment - 10th International Conference, DIMVA. Proceedings doi:10.1007/978-3-642-39235-1_8.
Lu, L., Yegneswaran, V., Porras, P., Lee, W., 2010. Blade: an attack-agnostic approach for preventing drive-by malware infections. In: Proceedings of the 17th ACM conference on Computer and communications security, pp. 440–450.
Lundquist, G.R., Mohan, V., Hamlen, K.W., 2016. Searching for software diversity: attaining artificial diversity through program synthesis. In: Proceedings of the 2016 New Security Paradigms Workshop, pp. 80–91.
Monero, 2022. Monero. https://www.getmonero.org/.
Moser, A., Kruegel, C., Kirda, E., 2007. Limits of static analysis for malware detection. In: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pp. 421–430. doi:10.1109/ACSAC.2007.21.
Mozilla, 2019. Protections Against Fingerprinting and Cryptocurrency Mining Available in Firefox Nightly and Beta. https://blog.mozilla.org/futurereleases/2019/04/09/protections-against-fingerprinting-and-cryptocurrency-mining-available-in-firefox-nightly-and-beta/.
Mozilla, 2022. Using web workers. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.
Musch, M., Wressnegger, C., Johns, M., Rieck, K., 2019a. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. 10.1007/978-3-030-22038-9_2
Musch, M., Wressnegger, C., Johns, M., Rieck, K., 2019. Thieves in the browser: web-based cryptojacking in the wild. In: Proceedings of the 14th International Conference on Availability, Reliability and Security. Association for Computing Machinery doi:10.1145/3339252.3339261.
Naseem, F., Aris, A., Babun, L., Uluagac, S., Tekiner, E., 2021. MINOS: A Lightweight Real-Time Cryptojacking Detection System. Ndss doi:10.14722/NDSS.2021.24444.
Payer, M., 2014. Embracing the new threat: towards automatically self-diversifying malware. In: The Symposium on Security for Asia Network, pp. 1–5.
Peng, P., Yang, L., Song, L., Wang, G., 2019. Opening the blackbox of virustotal: analyzing online phishing scan engines. In: Proceedings of the Internet Measurement Conference. Association for Computing Machinery doi:10.1145/3355369.3355585.
Ren, X., Ho, M., Ming, J., Lei, Y., Li, L., 2021. Unleashing the hidden power of compiler optimization on binary code difference: an empirical study. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. Association for Computing Machinery doi:10.1145/3453483.3454035.
Rokicki, T., Maurice, C., Botvinnik, M., Oren, Y., 2022. Port contention goes portable: port contention side channels in web browsers. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security. Association for Computing Machinery doi:10.1145/3488932.3517411.
Romano, A., Lehmann, D., Pradel, M., Wang, W., 2022. Wobfuscator: obfuscating javascript malware via opportunistic translation to webassembly. In: Proceedings of the 2022 IEEE Symposium on Security and Privacy (S&P 2022), pp. 1101–1116.
Romano, A., Zheng, Y., Wang, W., 2020. Minerray: semantics-aware analysis for ever-evolving cryptojacking detection. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1129–1140.
Schkufza, E., Sharma, R., Aiken, A., 2012. Stochastic superoptimization. ACM SIGPLAN Notices 48. doi:10.1145/2451116.2451150.
Tekiner, E., Acar, A., Uluagac, A.S., Kirda, E., Selcuk, A.A., 2021. In-browser cryptomining for good: an untold story. In: 2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS), pp. 20–29. doi:10.1109/DAPPS52256.2021.00008.
VirusTotal, 2020. VirusTotal - Home. https://www.virustotal.com/gui/home/search.
Wang, W., Ferrell, B., Xu, X., Hamlen, K.W., Hao, S., 2018. Seismic: secure in-lined script monitors for interrupting cryptojacks. In: Lopez, J., Zhou, J., Soriano, M. (Eds.), Computer Security. Springer International Publishing, Cham, pp. 122–142.
Wang, W., Sun, R., Dong, T., Li, S., Xue, M., Tyson, G., Zhu, H., 2021. Exposing weaknesses of malware detectors with explainability-guided evasion attacks. arXiv preprint arXiv:2111.10085.
Willsey, M., Nandi, C., Remy Wang, Y., Flatt, O., Tatlock, Z., Panchekha, P., 2020. EGG: fast and extensible equality saturation. arXiv e-printsarXiv:2004.03082.
Xia, M., Gong, L., Lyu, Y., Qi, Z., Liu, X., 2015. Effective real-time android application auditing. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy. IEEE Computer Society.
XMRIG, 2016. Xmrig. https://github.com/xmrig/xmrig.
Zhu, S., Shi, J., Yang, L., Qin, B., Zhang, Z., Song, L., Wang, G., 2020. Measuring and modeling the label dynamics of online anti-malware engines. 29th USENIX Security Symposium (USENIX Security 20). USENIX Association. https://www.usenix.org/conference/usenixsecurity20/presentation/zhu

Javier Cabrera-Arteaga is a Ph.D. student at KTH Royal Institute of Technology. His research interests are in the fields of Automated Software Engineering, Automated Testing and Software Diversification.

Martin Monperrus is Professor of Software Technology at KTH Royal Institute of Technology. His research lies in the field of software engineering with a current focus on automatic program repair, AI on code and program hardening. He received a Ph.D. from the University of Rennes, and a Master's degree from Compiègne University of Technology.

Tim Toady is a programmer analyst living in Merise, Estonia. His research interests include taint splatter analysis, Easter eggs and the usage of fakes in computing.

Benoit Baudry is a Professor in Software Technology at the KTH Royal Institute of Technology. His research focuses on automated software engineering, software diversity and software testing. He favors exploring code execution over code on disk.