7-2-2020

# A Deep-Learning Based Robust Framework Against Adversarial P.E. and Cryptojacking Malware

Faraz Amjad Naseem
*Florida International University*, fnase001@fiu.edu

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

# A DEEP-LEARNING BASED ROBUST FRAMEWORK AGAINST ADVERSARIAL P.E. AND CRYPTOJACKING MALWARE

A dissertation submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Faraz Naseem

2020

To: Dean John Volakis
College of Engineering and Computing

This dissertation, written by Faraz Naseem, and entitled A Deep-Learning Based Robust Framework Against Adversarial P.E. and Cryptojacking Malware, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Kemal Akkaya

Alexander Perez-Pons

A. Selcuk Uluagac, Major Professor

Date of Defense: July 02, 2020

The dissertation of Faraz Naseem is approved.

Dean John Volakis
College of Engineering and Computing

Andres G. Gil
Vice-President for Research and Economic Development
and Dean of University of Graduate School

Florida International University, 2020

DEDICATION

To my parents, Amjad Naseem and Fawzia Amjad,

my brother, Asad Amjad Naseem,

and

my loving partner, Katherine Delarosa

ABSTRACT OF THE DISSERTATION

A DEEP-LEARNING BASED ROBUST FRAMEWORK AGAINST ADVERSARIAL P.E. AND CRYPTOJACKING MALWARE

by

Faraz Naseem

Florida International University, 2020

Miami, Florida

Professor A. Selcuk Uluagac, Major Professor

This graduate thesis introduces novel, deep-learning based frameworks that are resilient to adversarial P.E. and cryptojacking malware. We propose a method that uses a convolutional neural network (CNN) to classify image representations of malware, that provides robustness against numerous adversarial attacks. Our evaluation concludes that the image-based malware classifier is significantly more robust to adversarial attacks than a state-of-the-art ML-based malware classifier, and remarkably drops the evasion rate of adversarial samples to 0% in certain attacks. Further, we develop MINOS, a novel, lightweight cryptojacking detection system that accurately detects the presence of unwarranted mining activity in real-time. MINOS can detect mining activity with a low TNR and FPR, in an average of 25.9 milliseconds while using a maximum of 4% of CPU and 6.5% of RAM. Therefore, it can be concluded that the frameworks presented in this thesis attain high accuracy, are computationally inexpensive, and are resistant to adversarial perturbations.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

# INTRODUCTION

**Malware and Adversarial Machine Learning:** Machine Learning (ML) techniques have been the de-facto solutions for several domains (e.g., speech recognition, natural language processing, computer vision, computer and network security, data mining, etc.) due to their ability to automatically generalize (i.e., classify or cluster) in both known and never before seen input samples [GMP18]. In fact, one of the main applications of ML in computer and network security has been the detection of ***mal***icious soft***ware*** (malware) [GMP20, MTICGN19, YLAI17]. Currently, ML-based models run in cloud environments in order to classify unknown samples [YLAI17].

Machine learning and especially its subset - deep learning (DL) - based models can provide superior performance over traditional methods (i.e., signature-based or heuristic-based) for malware detection [BBCC19]. However, recent research has shown that the efficiency of ML-based techniques can drop drastically due to adversaries attacking these systems via adversarially crafted/perturbed inputs. Such attacks have their roots in the computer vision domain with the study of Szegedy et al. [SZS+14], and then followed by others [GSS15, PMJ+16, CW17]. Researchers soon realized that it is possible to attack ML-models even without knowing the properties of the target classifier (e.g., features, classification algorithm, parameters, etc.) due to the transferability property [SZS+14, PMG16] of adversarial samples among different ML-based models. However, one needs to consider the domain-specific constraints while adopting such attacks. For instance, the adversarial manipulations to the samples should be imperceptible to the human eye [GMP18] in the computer vision domain, inaudible to the human ear in the audio domain [CW18], and should preserve semantics in the text domain [ERLD18], but should result in the sample evading the target classifiers.

In the malware detection domain, adversarial attacks to ML-based malware detectors involve adding carefully crafted perturbations to the malware samples that preserve the malicious functionality of the malware while allowing the samples to evade the target ML-based malware classifiers (i.e., modified malware samples are classified as benign). Researchers were able to craft adversarial malware samples and successfully evaded ML-based malware detection systems including Windows Portable Executable (PE)-based malware detectors [KDB+18, KBA+18, CRY+19], Android malware detectors [YKXG17, CHY17], PDF-malware classifiers [rL14, XQE16] and even cloud based proprietary anti-virus engines (e.g., Kaspersky, Eset, Sophos) [CSD19]. These examples clearly demonstrate that it is possible for attackers to evade state-of-the-art ML-based malware classifiers not by complex concealment techniques (e.g., polymorphism, metamorphism, encryption, packing), but by simple, minute adversarial perturbations. In order to defend ML-based malware classifiers from such attacks, researchers employed defense mechanisms such as adversarial training [SZS+14]. However, such mechanisms are computationally costly and also suffer from model poisoning and decreased detection accuracy [CRY+19]. Therefore, defending ML-based malware detection systems against adversarial attacks is still an open problem.

**Cryptojacking Malware:** In recent years, a new type of fileless malware that exploits the computational resources of end-users via browsers, has become increasingly common [CBOS20]. This new strain of malware, known as *Cryptojacking* (a.k.a., drive-by-mining) malware, performs unauthorized and covert cryptocurrency mining operations in browsers without the end-users' knowledge [ELMC18]. Both the tremendous rise in the monetary value of cryptocurrencies and the profitability of browser-based mining have been major driving forces behind the use of cryptojacking. As such, there have been a number of major cryptojacking incidents that have affected various popular services and websites in the past. For instance, some

of these cryptojacking incidents have affected popular streaming services and web applications like YouTube [you], Openload, Streamango, Rapidvideo, OnlineVideo-Converter [adg], Los Angeles Times [los], and some other organizational websites (e.g., US and UK government websites) [usu]. A prime example that made the news recently constitutes the episode of the Starbucks' WiFi network in Buenos Aires [sta], which was injecting cryptojacking malware through all its outgoing connections due to 1.4 million compromised MikroTik routers [BBD19]. Also, several researchers [ELMC18, MWJR18, KVM+18, MWJR19, KMM+19] confirmed that cryptojacking is prevalent in the wild based on their analyses' of websites in the Alexa and Zmap top 1 million lists.

The birth of cryptojacking can be attributed to a number of emerging technologies such as WebAssembly (Wasm), WebWorkers, and WebSockets. In general, these technologies have served to facilitate high-performing, scalable web applications running on browsers. In the context of cryptocurrency, Monero came forward with the promise of untraceable transactions, which caught the attention of malicious entities in the dark web [Kre18]. The Coinhive mining service provided WebAssembly-based Monero-mining scripts to website owners as an alternative source of income/profit [MWJR18, ELMC18, Kre18]. Thus, cryptojacking was born, a new cryptocurrency mining malware running on end-user browsers covertly that relies on the latest web technologies and easily reaching its victims via websites without requiring any software installation. The misuse of Coinhive scripts by malicious entities without the consent of end-users facilitated the shut down of Coinhive in March 2019 [VGOB20]. Although Coinhive is no longer maintained or operational, numer-

ous studies [BMZ20, VGOB20] indicate that cryptojacking malware is still in use in the wild. The findings of our study add further support to this statement.

## 1.1 Research Problems

This thesis has the following five major research components and problems investigated:

1. A detailed investigation into the use of image-based novel malware classification as a robust solution to adversarial attacks with high detection accuracy and low implementation overhead.

2. The introduction of two new adversarial attacks against ML-based malware classifiers that can evade state-of-the-art ML-based malware detectors while preserving the functionality of the modified malware.

3. The design of a novel cryptojacking detection mechanism that implements a Wasm binary classifier in a lightweight, computationally inexpensive, and incredibly fast end-to-end framework - MINOS

4. The use of a novel Wasm binary classification technique that utilizes gray-scale image representations of the binaries to train a convolutional neural network.

In general, to protect end-users from malware and adversarial attacks, a framework must ensure high accuracy in detecting the malware with low false positive and true negative rates. Additionally, the proposed framework must be robust to adversarial perturbations. Also, the proposed framework should be scalable and reproducible to ensure effectiveness against potential future malware threats. Finally, all the components of the proposed framework must work in real-time with low computational overhead to ensure efficient performance.

## 1.2 Organization of the Thesis

The rest of this thesis is organized as follows: In Chapter 2, we present pertinent background information that serves to support this thesis. In Chapter 3, the studies in the literature related to the work in this thesis, is presented. Chapter 4 introduces a novel deep-learning framework that is resilient to adversarial perturbations to P.E. malware. Then, in Chapter 5, we present a novel cryptojacking detection framework, MINOS. Finally, we conclude the thesis and propose future research directions in Chapter 6.

CHAPTER 2

# BACKGROUND

In this section, we provide background information pertinent to the context of the thesis. This includes a brief explanation regarding the structure of PE (portable executable) files, followed by an explanation of the process of representing a malware binary in the form of a gray-scale image. In addition, this section will provide necessary information regarding cryptojacking malware and the mechanisms it uses to operate. Further information can be found in the associated chapters.

## 2.0.1 Portable Executable Files and Visualization

The vast majority of malware on the Internet has the structure of Windows Portable Executable (PE) files and nearly 64% of malware detected by Symantec in 2018 were in PE format [sym19]. For this reason, we selected PE-based malware families for evaluation purposes in this study. In the following sub-sections, we will briefly describe the PE structure and also depict how a PE-based malware binary can be represented as a gray-scale image.

### Portable Executable (PE) Format

Portable Exectubale (PE) is the format used to create executable files, Dynamic Link Libraries (DLLs), common object files in 32-bit and 64-bit Windows operating systems (OS) [PEF20, GMP20]. It contains the necessary information needed by the OS for managing the executable file and provides an architecture-independent, and thus portable description. Each PE file consists of a PE header and various sections which are used by the linker in the loading process. The PE header possesses section, symbol, and optional headers' information. There can be several sections in a PE file, but the sections that are common in the majority of PE files are as follows:

- **.text** : encloses the program's actual code,
- **.rdata** : includes the read-only initialized data (e.g., strings, constants, etc.),
- **.data** : contains the initiliazed data,
- **.rsrc** : holds the resources utilized by the program, such as icons and images used by the program.

In addition to these sections, there are sections containing imported and exported symbols (i.e., .idata and .edata), uninitialized data (.bss), and thread-local storage (.tls) [PEF20].

### Visualization of Malware Binaries

A malware binary can be represented as a sequence of zeros and ones. Indeed, it is possible for this vector of binary values to be modified and transformed into an image [NKJM11]. Specifically, to enable such a conversion, the malware binary is represented as a vector of 8 bit unsigned integers (uint8) and then shaped into a two-dimensional array. The array is then divided by 255 to represent the array as a gray-scale image where the pixels take a value in the range of 0 to 255 (0 being black, and 255 being white). Fig. 2.1 shows an example of a malware binary from the Ramnit family being converted to an image using this technique. As shown in the figure, distinct regions in the gray-scale image of a PE malware binary correspond to specific sections in the PE structure. Examples of malware to image transformations of two unique malware families are shown in Figure 2.2. It can be observed that malware samples belonging to the same family of malware are visually extremely similar when converted to gray-scale images. Another observation is that the images of malware belonging to a specific family will be distinct from those belonging to a different family.

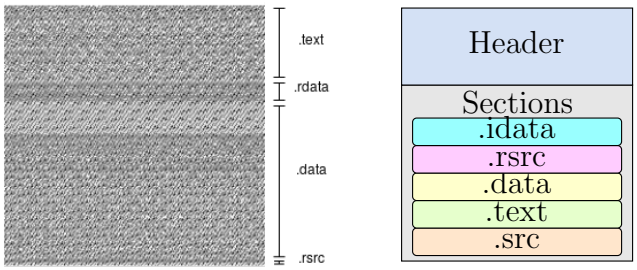


Figure 2.1: An image depicting a malware binary from the Ramnit family of malware (left) and PE file structure (right). Each section of the image is labelled corresponding to the respective section of the PE file excluding the PE Header.

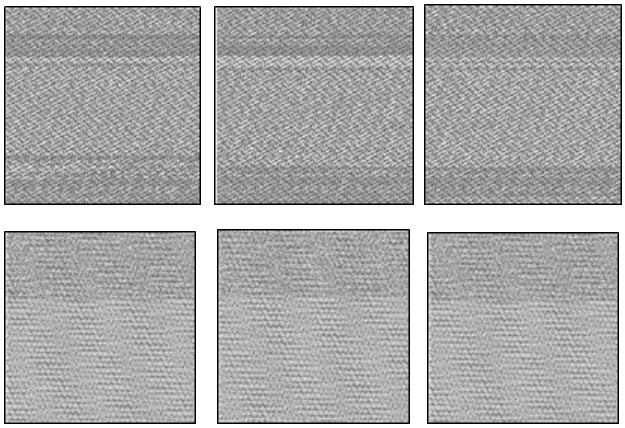

Figure 2.2: The first row represents gray-scale images of malware samples belonging to the Ramnit [RRF+18] family of malware while the second row represents gray-scale images of malware samples belonging to the Kelihos_ver3 [RRF+18] family of malware.

### 2.0.2 Cryptocurrency Mining and Cryptojacking

Cryptocurrency mining started with CPU-bound PoW schemes that mine Bitcoin or Ether currencies [MWJR18]. When one of the miners computes a block successfully, a new block for the blockchain is generated by the miner, and in return, the miner receives a certain amount of cryptocurrency as a reward. Since the revenue that could be obtained was directly proportional to the amount of processing power, Application-Specific Integrated Circuits (ASICs), Field Programmable Gate Arrays (FPGAs), and Graphics Processing Units (GPUs) became the de facto platforms for CPU-bound PoW schemes instead of ordinary desktop computers [MWJR18, ELMC18]. In

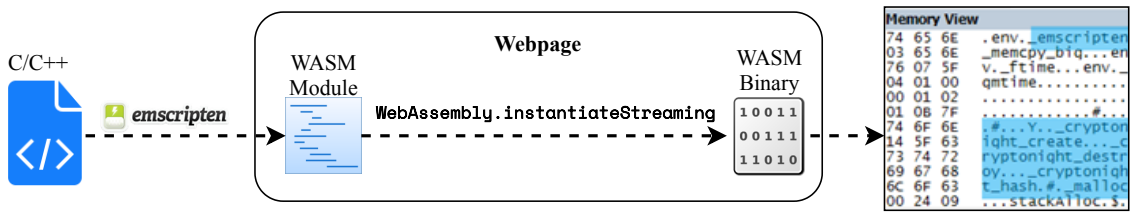


Figure 2.3: We illustrate the process of implementing malicious Wasm modules that mine cryptocurrecy in webpages. The right-most image shows the resulting Wasm module through a binary dissassembler. The highlighted portions of text (i.e. strings found in the binary) confirm that the module was compiled using Emscripten and that the binary is indeed executing cryptocurreny mining functions.

order to remedy this issue, new cryptocurrencies (i.e., Monero, Bytecoin, etc.) that utilize memory-bound PoW schemes emerged. These schemes were based on hash puzzles that required voluminous interactions with the memory rather than CPU power. Hence, ordinary computers started to be suitable mining environments for such cryptocurrencies.

Cryptocurrency mining in browsers first started with the Coinhive miner, which promised an alternative income opportunity to website owners in 2017 [ELMC18, MWJR18, Kre18]. In Coinhive mining, website owners were placing cryptocurrency mining scripts on their websites that would trigger the mining process within the visitors' browsers. There were also legitimate websites that would receive the consent of users to mine cryptocurrency. However, this extra income opportunity caught the attention of malicious entities. These individuals began to covertly mine cryptocurrency without the explicit consent of users. This phenomena is known as *cryptojacking*, or *drive-by mining/ coinjacking* [ELMC18]. In this new type of malware, attackers misuse the processing power of victims and derive revenue from it.

A malicious cryptocurrency mining script, namely cryptojacking malware, can be injected into websites in a number of ways [CBOS20,ELMC18]: i) Website owners can place such scripts in their websites and activate them taking consent of the visitors. ii) Third-party services can inject such scripts without informing neither website owners.

iii) Malicious browser extensions can run cryptocurrency miners in the background silently. iv) Attackers can breach servers, browser extensions, third-party services, and inject cryptojacking. v) Vulnerable network devices such as routers, access points, etc. can be exploited to inject cryptojacking to web traffic.

A browser-based cryptocurrency miner typically consists of a JavaScript code snippet that has the identification number of the script owner, and corresponding code that configures the mining process communicates with the cryptocurrency service provider and starts the mining operation. The identification number of the script owner discriminates the malicious entity that owns the script amid other entities from the cryptocurrency service provider's point of view. In this way, service providers can monitor and measure the total hashing power provided by script owners. The service provider communicates with miners through high-performance communication primitives, like WebSockets. In order to increase profit, cryptocurrency miners use WebWorkers to run the mining process in parallel via multiple threads. Further, to solve hash puzzles with high efficiency, they utilize miner implementations in WebAssembly instead of JavaScript [MWJR18].

### 2.0.3 WebAssembly and Cryptojacking Malware

WebAssembly (Wasm) is a low-level binary instruction format that promises to run code near native speeds in a stack-based virtual machine within the browsers [was20]. It is currently supported by four major, widely-used browsers, including Google Chrome, Mozilla Firefox, Microsoft Edge, and Safari. Its use of binary encoding results in efficiency in size and load-time, and execution speeds that are comparative to native machine code [was20]. Other principle features include it being easy to decode, hardware and platform-independent, and compact [RTH+18].

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

Advantageous features of WebAssembly make it suitable to implement and execute browser-based cryptocurrency mining functions that require substantial computational power such as *cryptonight_hash*. As such, a vast majority of browser-based cryptojacking malware implements Wasm to execute the cryptocurrency mining payload. This is apparent as Konoth et al. [KVM+18] reported that 100% of the 1,735 cryptocurrency mining websites they identified in their study utilized Wasm. In parallel to this study, Musch et al. [MWJR19] analyzed the prevalence of WebAssembly in the wild. They inspected Alexa top 1 million websites and realized that 0.16% of the websites employ WebAssembly. Their analysis revealed that more than half of the websites that employ WebAssembly are using it for malicious purposes. They highlighted that cryptojacking is the major application among malicious use-cases.

Crytojacking malware authors write code in C/C++ that performs mining functions including `cryptonight_create`, `cryptonight_destroy`, and `cryptonight_hash`. They then compile this to a Wasm module using the Emscripten toolchain. This Wasm module is then accessed through the JavaScript function, `WebAssembly.instantiateStreaming`. The `fetch()` method of the Fetch JavaScript API is used

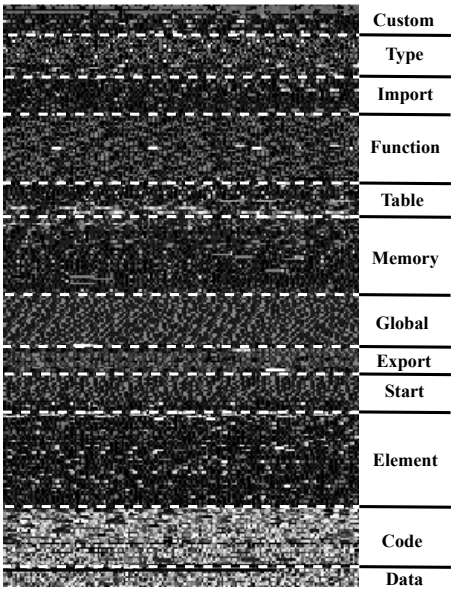


Figure 2.4: An image depicting a Wasm binary with each section of the image being labelled corresponding to each of the 12 respective sections described in Section 2.0.4.

as the function's first argument. This method fetches, compiles, and instantiates the module, enabling access to the raw byte code. During the compilation phase, the Wasm binary has already undergone optimization and can hook directly into the backend where machine code is generated and executed. This code performs mathematical operations that facilitate solving convoluted hash puzzles i.e., mining cryptocurrency. A visual depiction of this procedure can be seen in Figure 2.3.

### 2.0.4 Structure of WebAssembly Modules

A Wasm module comprises 12 distinct sections, each with its own section ID ranging from 0 to 11. The following list describes each module and includes each section's corresponding ID number (Figure 2.4).

0. **Custom section:** This section is either used for debugging purposes or by third-party extensions for custom purposes. It contains a name that identifies

the section as the custom section and a custom sequence of bytes for use by third-party extensions.

1. **Type section:** The type section decodes or defines a vector of function types. This component contains every function type utilized in the module.
2. **Import section:** This section decodes a vector of a set of imports that are required in order to confirm that the module is valid during instantiation. Each import definition consists of a module name and a name for an element within that specific module.
3. **Function section:** The function section consists of a vector of type indices representative of the *type* parameter of the functions defined in the module. Each function definition consists of 3 parameters: type, locals, and body. The locals and body parameters are encoded in the code section.
4. **Table section:** This section contains a vector of tables that are defined by their table type. A table consists of a vector of values of a specific element type.
5. **Memory section:** This component decodes into a vector of linear memories described by their memory type, consisting of raw, uninterpreted bytes.
6. **Global section:** The global section contains a vector of global variables used in the module. Each global variable definition consists of a single value describing its type.
7. **Export section:** This section consists of a vector of a set of exports that the host environment can access after module instantiation.
8. **Start section:** The start component defines the index of an optional start function that is invoked during module instantiation after tables and memories are initialized. This function is used to initialize the state of the module.

9. **Element section:** The element section is comprised of a vector of element segments that initialize a subrange of a table based on a specifically defined offset.

10. **Code section:** The code section contains a vector of code entries that consist of pairs of expressions and value types. The four value types that Wasm variables can take include 32 and 64-bit integers and 32 and 64-bit floating-point data. In the module's code, these types are represented by the terms $i32$ and $i64$, and $f32$ and $f64$ respectively. Each code entry is comprised of the size of the function code in bytes, and the function code. The function code, in turn, consists of local variables and the body of the function as expressions. These correspond to the local and body parameters of the sections of the function mentioned previously.

11. **Data section:** This section is comprised of a vector of data segments that initialize a range of memory based on a specifically defined offset.

CHAPTER 3

# LIERATURE REVIEW

In this chapter, we present the studies in the literature that are closely related to the research presented in this thesis.

## 3.1 Image-based malware detection, Adversarial Attacks to Malware Classifiers and Defense Methodologies

Adversarial Machine Learning and malware detection are two broad research areas and many prior studies exist in these fields. We do not go into the details of malware analysis and ML-based malware detection techniques in this thesis. Nevertheless, we refer the readers to the works of Elisan and Hypponen [EH13] and Sikorski and Honig [SH12] for malware analysis; Gilbert et al. [GMP20] and Ucci et al. [UAB19] for ML-based malware detection. Here, we briefly review the works on image-based malware detection, adversarial attacks to malware classifiers and defense methodologies.

### 3.1.1 Image-based Malware Detection

In 2011, Nataraj et al. [NKJM11] converted malware binaries into gray-scale images and realised that malware samples belonging to the same family look very similar to each other. Based on this, they extracted the features of gray-scale malware images and used K-Nearest Neighbor for classification, which enabled them to achieve 98% classification accuracy. Since then several studies (i.e., [BSK19], [FXW$^+$18], [KM13], [HLI13] and [NQZ18] to name a few) employed malware images and visualization techniques for malware classification.

### 3.1.2 Adversarial Attacks to Machine Learning Systems

Adversarial attacks to ML systems emerged in the computer vision domain with the work of Szegedy et al. [SZS+14]. They realized that it is possible to cause a deep learning model to misclassify an image simply by adding perturbations that are imperceptible to the human eyes. Szegedy et al. used gradient-based optimization algorithms to find optimum perturbations that maximize the prediction error and cause the model to misclassify the input. After this work, several studies were performed in computer vision using gradient-based approaches. Some of the well-known adversarial attacks include L-BFGS [SZS+14], FGSM [GSS15], JSMA [PMJ+16], DeepFool [MFF16] and C&W [CW17]. Researchers also proposed defense and detection mechanisms to counter these attacks (adversarial training [SZS+14], defensive distillation [PMW+16], feature squeezing [XEQ18]). We refer the readers to [GMP18], [QLZW19] and [WLK+19] for a comprehensive review of attacks (training time and test time attacks), detection and defense mechanisms.

### 3.1.3 Adversarial Attacks to Malware Classifiers

Adversarial attacks in computer vision are relatively easy to implement since the domain-specific constraint is that the generated perturbations should be imperceptible to the human eye [GMP18, LCBDR19, AHHO18]. However, in malware domain, it becomes much harder since the perturbations have to preserve the functionality of the **functionality** malware executable while bypassing the ML-based malware classifier [LCBDR19, AHHO18, CSR19, CSD19, AKF+18, KBA+18, SCJ19].

Adversarial attacks to malware classifiers can be grouped into three classes based on the type of malware: Windows Portable Executable (PE) malware, Android malware, and Portable Document Format (PDF) malware. To the best of our knowledge,

adversarial attacks to GNU/Linux malware and Mac OS malware do not exist. We focus on Windows PE malware in this thesis since the majority of malware on the Internet has the structure of PE files [CYB17, CSR19, RSRE18] and nearly 64% of malware detected by Symantec in 2018 were in that format [sym19].

In terms of PE malware, several studies proposed adversarial attacks to ML-based malware classifiers. These attacks can be grouped into three categories: *1) Addition of Bytes:* This refers to the addition of crafted bytes to the end of the malware samples [KDB+18, KBA+18, SCJ19, CRY+19], to unused sections [KBA+18], or modification of slack bytes that are added by compilers for alignment purposes [SCJ19]. Bytes can be crafted randomly, using parts of benign binaries, or via one of adversarial attacks applied to ML systems in computer vision (e.g., FGSM, JSMA, CW, etc.). *2) Modification of Features:* This method involves the addition or removal of API call features to bypass ML-based malware classifiers [RSRE18, CYB17, AHHO18, AKF+18, FRZ+, CSD19, CSR19, LCBDR19, HT17], or the use of a Generative Adversarial Network (GAN) to craft adversarial malware [HT17]. *3) Other Modifications:* This process includes using a tool like the LIEF library [lie20] to perform changes on the malware which do not intend to break the functionality [AKF+18, FRZ+, CSD19, CSR19, LCBDR19], and malicious code injection to benign files [FRZ+] using a tool like ROPInjector [PNX].

When we consider these attacks, we see that preservation of malware functionality is not always achieved. Attacks that append bytes to the end of a malware binary do not touch the executable code part of the binary, and therefore guarantee the preservation of functionality. Modifying slack bytes also results in the preservation of functionality, however it has a low evasion rate [SCJ19] since the amount of such bytes is quite small. Among the set of feature modification attacks, the only attack that preserves the functionality is proposed by Rosenberg et al. [RSRE18]. Their attack

is complicated as it does not alter the malware, but uses API hook mechanism and wraps it with proxy code and operating system Dynamic-link Libraries. The majority of the other modification attacks use the LIEF library to modify the malware PE, but nearly all of the studies using this method indicated that the functionality of the modified malware is not always preserved. In addition, malicious injection which uses ROPInjector, has several limitations and can not always inject malicious samples.

### 3.1.4 Defense Mechanisms Against Adversarial Attacks to Malware Classifiers

There exist only a few defense studies against adversarial PE-based malware perturbations in the literature. Al-Dujaili et al. [AHHO18] proposed a secure learning framework, which performs training by considering the adversarial loss of malware samples and natural loss of benign samples. As a defense mechanism, Chen et al. [CYB17] proposed SecDefender which progressively retrains the classifier using adversarial examples and also tries to maximize the cost of evasion for the attackers. The last study [CRY+19] analyzed the performance of the adversarial training approach of Szegedy [SZS+14] and proposed a pre-detection mechanism. However, adversarial training has high computational cost, can lower the detection performance and can also be affected by model poisoning [GMP18].

### 3.1.5 Adversarial Attacks to Image-based Malware Classifiers

The three categories of adversarial attacks described earlier are all against non-image-based classifiers. On the other hand, adversarial attacks to image-based malware clas-

sifiers started to be a focus of research recently. *A crucial factor that must be taken into consideration when generating adversarial malware to evade image-based classifiers, is creating an adversarial sample whose* ***image representation can evade the classifier****, while* ***retaining its malicious functionality****.* There are four studies in the literature which aim to evade image-based malware classifiers but cannot achieve both of these conditions. Park et al. [PKY19] employed FGSM and C&W attacks to generate an adversarial image of a malware sample first. Then, using their algorithm, they inserted semantic no-operation (NOP) instructions into the original malware sample to make it appear like the an advesarial sample. Although adding NOP instructions do not change the actual logic of a binary, adding instructions changes the section size and addresses, and therefore breaks the executable. Liu et al. [LZLL19] converted the malware binaries to images and then generated an adversarial image using the FGSM attack which can evade the image-based classifier. However the resulting file may have a series of unmeaningful character sequences which can break its functionality.In the work of Vi et al. [VNNT19], a malware binary is converted into an image, and then the resource section of the image is determined and perturbed via FGSM to evade the classifier. Following this, the perturbed pixels of the resource section is converted back to binary and is used to modify the original malware's resource section. This approach can cause the Windows PE loader to fail to load the malware, since this section has to follow a certain structure for successful parsing [pec20].Khormali et al. [KAC+19] proposed COPYCAT that uses adversarial example (AE) padding and sample injection attacks. AE padding generates an adversarial image of a sample using well-known attacks (e.g., FGSM, C&W, etc.) and then converts the image to bytes and appends the generated bytes to the end of the original malware, essentially doubling the size of the sample. The sample injection attack injects targeted class samples after the exit code of the malware which has

a high probability of breaking the malware PE due to changing the offsets of the sections after the code section of the malware.

## 3.2 Cryptojacking Detection Systems

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

### 3.2.1 Browser-based Mining (Cryptojacking) Detection

Several researchers proposed highly accurate detection systems against cryptojacking malware. Hong et al. [HYY+18] proposed CMTracker that uses the cumulative time spent on hashing operations and stack characteristics of threads. Wang et al. [WFX+18] proposed SEISMIC, as a semantic signature-matching-based detection system that instruments the Wasm modules to count specific instructions in runtime. Rodriguez and Posegga [RP18] proposed RAPID, a Support Vector Machine (SVM) based classifier that uses CPU and memory events, and network traffic features. Kharraz et al. [KMM+19] proposed OUTGUARD, that builds an SVM classifier using features of runtime, network, mining, and browser events. Bian et al. [BMZ20] proposed MineThrottle, which uses a block-level profiler and dynamic instrumentation of the Wasm code that is pointed by the profiler at compile time. Kelton et al. [KBRS20] proposed CoinSpy which utilizes computation, memory, and

network features. Conti et al. proposed [CGLP19] that uses Hardware Performance Counters (HPC) data to detect cryptojacking. Konoth et al. [KVM+18] proposed MineSweeper that firstly analyzes the Wasm modules and counts number of specific instructions. It tries to find out how similar the analyzed module is to cryptojacking malware Wasms. In addition, it monitors cache events during runtime.

### 3.2.2 Host-based Stand-alone Mining Detection

In terms of malicious cryptocurrency miners not targeting browsers but directly hosts, Darabian et al. [HHD+20] proposed a detection system that uses opcode sequences and system calls of Windows Portable Executable (PE) files. Berecz and Czibula [BC19] employed both static features (i.e., entropy, and header, section, and function information) and API calls. Vladimír and Žádník [V9] presented two technique, in which the first technique uses a decision tree on the flow features, and the second technique acts like a miner client probing the miner server.

### 3.2.3 Network-based Detection

As a network-level detection mechanism, Neto et al. [NLFM20] proposed MineCap, a network-flow-based detection and blocking mechanism to protect the network of devices controlled by the SDN controller. MineCap relies on Apache Spark Streaming library and incremental ML model to detect the cryptocurrency mining flows.

### 3.2.4 Challenges and Need for a New Detection System

Cryptojacking detection is challenging. Benign web applications frequently use the technologies that are also used by cryptojacking malware (e.g., WebAssembly, WebWorkers, WebSockets). Similar to cryptojacking scripts, benign web applications can

consume high CPU and memory resources. All of these characteristics can affect the accuracy and false positive rates of detection systems. In addition, end-users browse the web through a variety of browsers using various operating systems that require solutions to be platform-independent. Moreover, end-users expect pages to load quickly and run flawlessly, which force detection solutions to be lightweight and efficient. In addition, attackers can throttle the resource consumption of their malware, change function names and strings, use proxies, dynamically generated domain names, and encrypted communications that make the detection increasingly challenging.

However, considering the challenges and approaches proposed in prior work, cryptojacking detection can still be improved drastically. Existing detection approaches, albeit useful, have several drawbacks. First of all, cryptojacking is moving from JavaScript to WebAssembly for various reasons (e.g., performance, hardware support) [KMM+19]. However, only a small portion of prior studies [WFX+18,KVM+18, KMM+19] take this change into account. Secondly, cryptojacking detection systems [WFX+18,KVM+18,KMM+19,RP18,BMZ20,KBRS20] relying on dynamic analysis features can suffer from high computational overhead, reduced measurement accuracies due to noise caused by other processes and false positives resulted from benign websites using the same technologies. For this reason, practical applications of such schemes may cause quality-of-experience issues for end-users. Moreover, attackers can easily find ways to circumvent existing detection systems. To be more specific, fixed threshold values used by CMTracker [HYY+18] can be evaded easily. The techniques of Vladimír and Žádník [V9] are not effective against private mining pools or mining pools hidden behind proxies. Cryptojacking scripts can use encrypted communication, dynamic domain names, alternative communication primitives (e.g., XMLHttpRequest), and proxies to bypass detections systems that use network-based features [MWJR18]. In addition, some detection systems have drawbacks that may

limit their acceptance and usage by end-users. For instance, one of the features used by OUTGUARD [KMM+19] (i.e., MessageLoop event) is browser-dependent. MineCap [NLFM20] can be utilized only by operators which employ SDN in their networks. Therefore, individual users cannot use it. Instrumentation code added by SEISMIC [WFX+18] may severely affect the performance of legitimate web applications that use Wasm, which may degrade the quality of experience of end-users.

CHAPTER 4

# A LIGHTWEIGHT IMAGE-BASED MALWARE CLASSIFICATION SYSTEM RESISTANT TO ADVERSARIAL MACHINE LEARNING ATTACKS

## 4.1 Introduction

In this chapter, we show that converting the malware detection problem into image-based malware classification problem provides robustness against adversarial perturbations. The underlying robustness stems from the fact that adversarial attacks, which are relatively easy to apply to background image in the computer vision domain, are extremely difficult to apply to transformed background image of malware samples. This is because such an operation that adds adversarial noise to a malware image has a very high possibility of breaking the functionality of the actual malware when the image is converted back to a malware binary. There have been a few adversarial attack attempts [PKY19, LZLL19, VNNT19, KAC+19] to image-based malware classifiers recently. However, most of the attacks fail to preserve the functionality of the adversarial malware sample. To prove the robustness of image-based malware detection against adversarial attacks, we select adversarial attacks that preserve the functionality of the malware sample while evading the state-of-the-art malware classifiers. In addition, we introduce two novel adversarial attacks that preserve the malware functionality after modifications. For our evaluation, we used the 2015 Microsoft Malware Classification Challenge dataset [RRF+18] which includes real malware samples from nine different malware families in the Windows Portable Executable (PE) format. We trained a convolutional neural network (CNN)-based classifier using the gray-scale background image of the malware in the dataset. We applied adversarial attacks including our two novel attacks to our image-based malware classifier. The

results of our evaluation show that the image-based malware detection approach is robust against adversarial attacks that can easily fool a state-of-the-art ML-based malware detector. The evasion rate of adversarial samples dropped to 0% in certain attacks. Furthermore, our tests demonstrate that even if an adversary increases the amount of adversarial perturbations by up to 20% of the malware sample's original size, our image-based malware detector still provides a detection accuracy of above 80%. Moreover, we analyzed the overhead incurred by implementation. The analysis indicates that the image-based malware detection technique provides a 70% decrease in training time and a three-fold reduction in RAM usage during the training process in comparison to a start-of-the-art ML-based malware classifier. Our extensive analysis shows that image-based classifiers are both efficient and also robust against adversarial attacks that preserve the functionality of the malware. For this reason, employing an image-based malware classifier does not require additional defense mechanisms, such as adversarial training; hence, it remains immune to model poisoning. To the best of our knowledge, this is the first work in the adversarial malware literature that demonstrates and analyzes the robustness of image-based classifiers against adversarial attacks.

***Contributions:*** In summary, the main contributions of this chapter are as follows:

- Proposition of image-based novel malware classification as a robust solution to adversarial attacks with high detection accuracy and low implementation overhead,
- Introduction of two new adversarial attacks against ML-based malware classifiers that can evade state-of-the-art ML-based malware detectors while preserving the functionality of the modified malware.

## 4.2 Differences from Prior Works

Considering the related work, we can see that although adversarial attacks to ML systems and image-based malware classifiers exist, whether image-based malware classifiers are robust against adversarial attacks remains an open question. Most of the proposed attacks cannot preserve the functionality of the malware. The defense mechanisms have high computational costs and suffer from model poisoning and reduced detection accuracy issues. This study, however, analyzes the robustness of image-based malware classifiers against adversarial PE-based malware perturbations. Our analysis shows that image-based classifiers are robust against adversarial attacks that preserve the functionality the malware. For this reason, employing an image-based malware classifier does not require adversarial training; hence, it remains immune to model poisoning. To the best of our knowledge, this is the first work in the adversarial malware literature that demonstrates and analyzes the robustness of image-based classifiers against adversarial attacks.

## 4.3 Image-based malware classification and adversarial attacks

In this section, we describe the structure of our evaluation architecture in which we convert malware binaries to gray-scale background image, train a CNN classifier, apply adversarial attacks and test the performance of malware classifiers (our image-based classifier and the state-of-the-art ML-based classifier, Malconv) against adversarial malware samples. We also propose two novel black-box adversarial attacks that utilize brute-force techniques to generate adversarial samples, followed by our threat model.

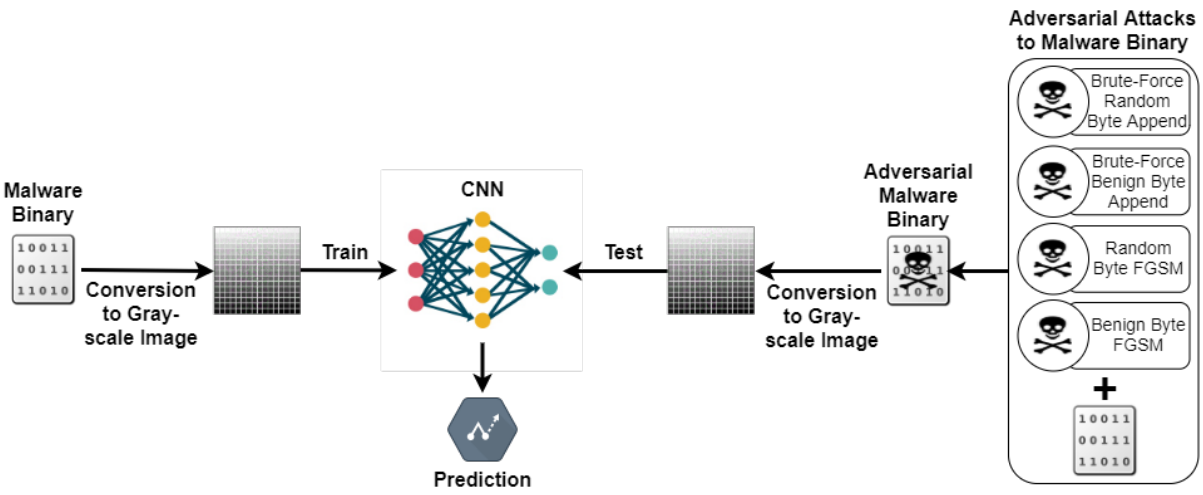


Figure 4.1: An overview of the architecture of our proposed approach. Malware binaries are converted into gray-scale background image before being fed to the CNN for the training process. A total of 4 adversarial attacks are applied to malware binaries, each appending bytes to the end of the samples. Similar to the training process, these samples are converted to background image and then fed to the model as input in order to classify them according to the malware family they belong to.

### 4.3.1 Overview

Our proposed method consists of a three-stage process. A visual depiction of the architecture of our approach is illustrated in Figure 3. In the first stage, each malware binary undergoes pre-processing during which each binary in our dataset is converted to an array of unsigned 8-bit integers and normalized to a common size. These arrays, which represent the binaries as gray-scale background image, are then used to train a convolutional neural network (CNN) in the second stage. In the third stage, adversarial examples are then generated using each of the 4 attack vectors outlined in Section 4.4, converted to gray-scale background image and tested against the trained CNN. The CNN then makes a prediction as to which class of malware the adversarially crafted malware samples belong to. The following sub-sections describes each stage of our method in detail and includes how the binaries are converted into background image to be used to train the model, details of the structure of the model and the

tools utilized to create the model and finally, the methods used to craft adversarial malware.

### 4.3.2 Preprocessing: Conversion of Malware Binary to Image

**Algorithm 1:** Malware Binary to Gray-scale Image

**Input :** Malware Binary
**Output:** Gray-scale Image Array of Malware Binary

```
1  for file in getCwd() do
2      f → open(file)
3      ln → getSize(file)
4      width→ math.pow(ln, 0.5)
5      rem → ln%width
6      a → array('B')
7      a.fromfile(f, ln − rem)
8      f.close()
9      g → reshape(a, (len(a)/width), width)
10     g → uint8(g)
11     h → resize(g, size, size)
12     h → h/255
13 return h
```

Before the model can be trained, the dataset needs to be preprocessed in order to convert the data into a format that the model is able to use as input. This involves converting each malware binary in the training set to a gray-scale image and then resizing it to a common size. The details of this procedure are depicted in Algorithm 3.

In Line 1, a for loop ensures that each file in the training set directory is visited with each iteration of the loop. Lines 3-5 calculate size parameters ensuring that the final array will have a relatively similar length and width. In line 6, the file is converted to an array of unsigned integers. Lines 2-8 convert the malware binary to

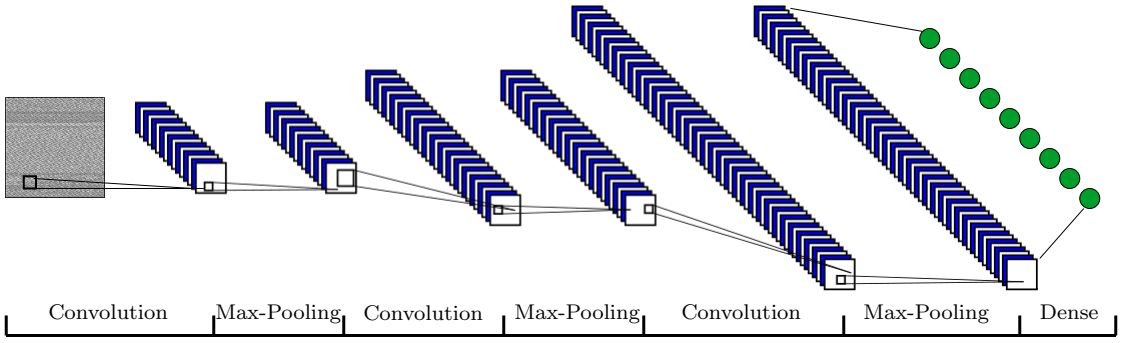


Figure 4.2: The structure of the convolutional neural network used to classify malware binaries.

an array of integers. Lines 9-10 reshape the created array and convert it into an array of 8 bit unsigned integers (uint8) that range in values from 0 to 255. The value of each integer in the array represents the brightness of a pixel ranging from black to white (0 to 255). In lines 11-12, the image array is resized to a common size of 100 by 100 and the pixel values are normalized to a range of 0 to 1 by dividing the array by 255. This normalization is done as it is easier for the model to process input arrays with a smaller range of values. This process continues for each file in the directory until each file has been successfully converted to an image array.

### 4.3.3 Convolutional Neural Network

The malware classifier consists of a convolutional neural network (CNN) built using the TensorFlow software library, specifically using TensorFlow's high-level API, Keras. The CNN was written in Python 3 using TensorFlow version 1.13.1. The model was trained on a system running Ubuntu 16.04.01 with dual Intel Xeon E5-2630 V4 processors and 62 GB of available RAM. The system has a total of 20 cores with each processor running a base frequency of 2.20 GHz.

The structure of the CNN consists of 3 sets of convolution layers followed by max-pool layers with an increasing number of filters in each successive convolution layer

(16, 32 and 64). The kernel size used in each convolution layer is set to (3,3) while the pool size of each max-pool layer is set to (2,2). These layers are followed by two dense layers with the final output being a vector representing the probability that a sample belongs to each of the 9 classes in the dataset. A visual depiction of the overall structure can be seen in Figure 4.

The model is trained on labeled, pre-processed malware samples from our dataset with a validation split of 0.2, i.e., the model was trained on 80% of the samples while the remaining 20% are used to validate or test the accuracy of the model. The samples are labeled 0-8 according to which malware family they belong to. The CNN is trained with the number of epochs being set to 100 and the batch size set to 32. The model achieved an accuracy of 98% when tested against the malware samples in the validation set.

### 4.3.4 Adversarial Malware Generation

As we explained in the Section 2.3, the functionality of the modified malware is guaranteed only for a subset of byte append attacks and one of the feature modification attacks. Taking this into consideration, we apply four byte-append attacks to generate adversarial samples that are able to evade MalConv [RBS+18], a state-of-the-art ML-based malware classifier. MalConv is a CNN-based malware classifier that analyzes the raw bytes of PE-based malware samples. It is a popular malware detector that is used in various other studies as a target model to create adversarial samples from PE-based malware files [KBA+18, KDB+18, FRZ+, SCJ19, CRY+19].

From the adversarial ML point of view, our attacks can be organized into two categories: black-box and white-box attacks. It should be noted that the following attacks considered in this study guarantee that malware functionality remains intact as each of the attacks appends bytes to unreachable portions of the binary. In this case, all

attacks append bytes to the end of the binary, and therefore the executable code in the binary remains unaffected, section sizes and offsets don't change, and malware functionality is preserved. As an additional constraint, given a malware binary $x$ of size $s$, appended by byte perturbations $p$, $p$ can not exceed 10% of the original sample size. This is because from an adversarial point of view, the added perturbations are meant to be small, seemingly undetectable additions to the original malware binaries, relative to the original size of the binary. Other works in the literature append a maximum of only 1% of the original sample size [CRY+19], [KDB+18], [SCJ19], therefore, a maximum upper bound of 10% is suitable. In other words, the the generation of adversarial samples through each method is bounded by the equation:

$$x + p <= 1.1s. \tag{4.1}$$

**Black-box Attacks:**

In this case, it is assumed that the adversary has no knowledge regarding the internal parameters or the structure of the target/victim model. The only information the adversary has access to is the final classification result of the model with respect to a given input file. The following two black-box attacks are novel methods that utilize brute-force techniques to generate adversarial malware samples.

**Brute-Force Random Byte Append**: In this attack, randomly generated bytes are appended to the end of a malware binary with each iteration until it is classified as benign or the size of the resulting binary reaches the threshold set in Equation 4.1 is . In the event that the adversarial sample, $x'$, generated through this method is still being classified correctly once this threshold is reached, 10-byte increments are extracted from random points in $x$ and appended to the end of the binary. This iterative process continues until the sample is classified as benign. A combination of

both of these random byte-append techniques ensures that adversarial samples are generated for all of the malware binaries in our validation set.

**Brute-Force Benign Byte Append:** In this scenario, portions of benign files are appended to the end of a malware binary $x$ until it is either classified as benign or reaches the upper bound in equation 4.1. To accomplish this, with each iteration of the attack, a file is chosen at random from the set of benign files and a section of 10 bytes is extracted from a random location in that file, and appended to the end of the malware binary. If the adversarial sample generated reaches the upper bound, and is still classified correctly, the perturbations are removed and the process is repeated with another random benign file being selected from the data set, until $x'$ is misclassified as a benign file.

**White-box Attacks:**

In this case, the adversary has complete access to the structure of the victim model, including the internal parameters, hyperparameters and weights for the convolutional neural network.

**Random Byte FGSM:** This method is an adaptation of the FGSM approach originally proposed by Goodfellow et al. for image based deep learning classifiers [GSS15]. FGSM creates adversarial malware samples by using the gradients of neural network. The gradient of the cost function used to train the model, $J(\theta, x, y)$, with respect to an input malware binary, is used to generate a new binary that maximizes loss. This can be represented using the following equation:

$$x' = x + \epsilon * sign(\nabla_x J(\theta, x, y)). \tag{4.2}$$

Here, $x$ is the original malware binary, $y$ is the binary's original label, $\epsilon$ is a constant multiplier used to control the size of the perturbations, $\theta$ are the model's parameters and $J$ is the loss function with respect to original malware binary. The main goal here is to create a new malware binary $x'$ that maximizes the loss function. This is achieved by appending a certain number of bytes, namely $numBytes$ in the form of random bytes to a malware binary and updating their values (as dictated by Equation 4.2) in an iterative fashion with the binary moving further away from its original label with each iteration. In this case, The number of bytes appended with each iteration was set to 100, while the number of iterations was similary set to 100. With *MalConv*, the model is not differentiable end-to-end as the input bytes are mapped to an 8-dimensional vector in the embedding layer, and therefore computing the gradient is not possible. To overcome this issue, as proposed in [KDB+18] and [SCJ19], the gradient-based updates of the appended bytes are performed in the embedding space and then the updated byte value is mapped to the nearest byte value along the direction of the embedding gradient.

**Benign Byte FGSM:** This attack is very similar to the aforementioned FGSM attack except that instead of adding $numBytes$ in the form of random bytes, it adds benign byte portions from a randomly selected file from the set of benign files. The byte values are then updated iteratively over $numIterations$ using Equation 2.

### 4.3.5 Threat Model

The adversary, in the context of this study, can be classified as an individual that attempts to add minimal perturbations (bytes) to the end of malware samples, in such a way that they are able to bypass byte-based malware classification models. They are able to do so as these added perturbations result in the sample being misclassified as benign. The adversary can either have complete (white-box) or partial access (black-

box) to the classification model and its internal parameters/hyperparameters. In each of these two scenarios, the crafting of adversarial samples is governed according to the following set of adversarial goals:

- The perturbations are appended to the end of the malware, past the executable code portion of the samples.
- The crafted adversarial malware sample is able to retain its malicious functionality after the addition of adversarial perturbations.
- The added perturbations or bytes result in the target model misclassifying legitimate malware samples as benign.

Taking these goals into consideration, the adversary applies the two black-box and two white-box attacks outlined in the previous section.

## 4.4 Performance Evaluation

In this section, the methodology used to carry out our evaluation is presented, followed by the results of the evaluation and the overhead analysis.

### 4.4.1 Methodology

To evaluate the performance of the image-based malware classifier, adversarial samples were generated utilizing each of the four attack methods outlined in Section 4.4 and tested against the classifier to see what percentage of the samples were successfully able to evade it (evasion rate) and cause a misclassification. These results were compared to the evasion rate of the attacks when tested against MalConv.

For the purpose of the evaluation, MalConv was trained with the same validation split as the image-based classifier i.e., it was trained and tested on the same samples

as the image-based classifier. In addition, the methods used to create adversarial samples were applied to the validation set so as to ensure that adversarial samples were created from malware binaries that the model has not been trained on.

### 4.4.2 Dataset

The dataset used in our study is the 2015 Microsoft Malware Classification Challenge dataset [RRF+18]. It contains 10869 labelled malware samples from 9 different families of malware that including: Ramnit, Lollipop, Kelihos_ver3, Vundo, Simda, Tracur, Kelihosver1, Obfuscator.ACY, and Gatak. The files consist of hexadecimal representations of the binary content of malicious PE files with the exception of the PE header. The PE headers are removed from each binary, pertaining to the rules of the classification challenge, so that classification of the malware is made a more difficult task. This set of labelled malware was divided into training and validation sets using an 80/20 split, respectively. For the benign samples, 10642 executables are taken from pure installations of Windows 10 (64 bit) and Windows 7 (64 bit) operating systems.

Although there are various other popular resources from which malware samples can be collected including VirusShare and VirusTotal, the reason this particular data set is chosen is due to the difficulty in finding labelled malware samples that indicate which unique malware family they belong to. The aforementioned resources provide unlabelled malware samples in this regard and labelling must be done manually. Even still, most malware samples from these resources are classified under various different malware families making it difficult to correctly classify them into a single family of malware.

### 4.4.3 Results

Both classifiers were tested considering numerous accuracy metrics including accuracy, precision, recall, and F1-score. Table I illustrates the result of calculating each of these metrics for the classifiers. Our image-based classifier exceeds MalConv's performance in every category. Both accuracy and F1-scores are both approximately 10% higher while precision and recall are 11% and 10.3% higher respectively. The prime reason for this is that our classifier is able to be make accurate predictions even in the absence of the PE header information. In the original MalConv study, it is indicated that the PE-header is the most important feature in the model's classification of malware, accounting for its weaker performance.

Table 4.1: Accuracy Metrics for both Classifiers

| | **Accuracy (%)** | **Precision (%)** | **Recall (%)** | **F1-score (%)** |
|---|---|---|---|---|
| **MalConv** | 87 | 89.2 | 87.2 | 87 |
| **Image-Based Classifer** | 98 | 98.3 | 97.5 | 98.2 |

In Figure 5, the evasion rates of each of the four attacks are shown when tested against MalConv and the image-based classifier. The brute-force random and benign byte-append attack methods completely failed to cause a misclassification, resulting in a 0% evasion rate for the image-based classifier. In the case of MalConv however, these attacks were able to achieve a 100% evasion rate as the brute-force strategy used to craft the adversarial malware, by design, guarantees misclassification for each crafted adversarial malware sample. The adversarial examples generated using the FGSM random byte-append and FGSM benign byte-append techniques performed slightly better against the image-based classifier but were still only able to obtain an evasion rate of 2% and 3% respectively. Against MalConv, the evasion rates of both FGSM attacks were 43% and 51% respectively, indicating that, similar to the other attacks, the image-based classifier outperformed MalConv. Given these results, it is evident that the image-based classifier performed substantially better than MalConv

and remained robust to adversarially generated perturbations across all adversarial malware generation methods.

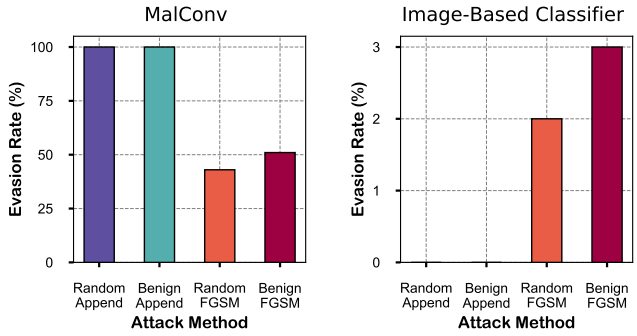


Figure 4.3: A side-by-side comparison of the evasion rates of the attacks used in this study when applied to MalConv and our Image-Based Classifier. It is evident that the vast majority of adversarial examples generated using the 4 attacks methods failed to misclassify the image-based classifier.

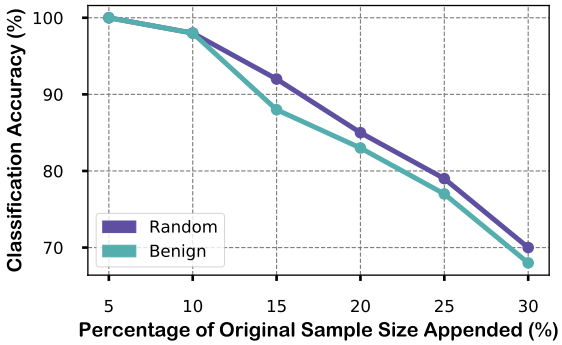


Figure 4.4: The classification accuracy of the image-based classifier with respect to the amount of bytes appended to the samples as a percentage of the original sample size.

In the case of our image-based classifier, after a certain amount of bytes are appended, relative to the original file size, the classification accuracy begins to decrease. We performed an analysis of this phenomenon on the brute force random and brute force benign byte append attacks as they achieve the highest evasion rate of all the attack methods. Figure 6 shows a near linear relationship between the percentage of bytes appended and the classification accuracy for both attacks. Every 5% increment

in the number of bytes appended past 10% results in a steady decline in classification accuracy. This is due to the fact that the pre-processing stage involves scaling the image down to $100 * 100$ and as more bytes are appended to the end of the binary, the portion of the binary containing the original malware sample is squeezed, resulting in a loss of features.

### 4.4.4 Overhead Analysis

Here, the overhead incurred in the implementation of our image-based malware classifier is described and compared to the overhead of MalConv. The majority of the overhead was incurred during the training and pre-processing stages of the CNN. The pre-processing stage, where the malware binaries were converted to vectors of 8 bit unsigned integers took a total of 3 minutes and 24 seconds. This preprocessed dataset was stored in virtual memory as a array of arrays, taking up a total of 87 Kb of space. The total time taken to train the model on our system was 16 minutes and 40 seconds with the average RAM usage during training being 2.8% or 1.7 GB.

With MalConv, the total time to train the model on the same dataset was 27 minutes and 52 seconds with the average RAM usage during training being 9% or 5.58 GB. This is approximately a 70% increase in training time and a 300% increase in RAM usage as compared to our image-based classifier.

## 4.5 Summary & Benefits

Here, some key points are discussed including the underlying reason for the performance of the image based classifier, the reasoning behind the crafting of the adversarial samples and choice of attacks, and finally, the benefits of our study.

**Underlying Concept:** According to the results obtained, it is apparent that the image-based malware classifier remained robust to adversarial malware samples. The underlying reason for this is that the perturbations are added to the end of a malware binary. As a result of this, when the binary is converted into a gray-scale image, the majority of the image remains identical to the original unperturbed sample. This allows the classifier to correctly predict the class of malware the adversarial sample belongs to. However, as seen in our performance evaluation, after a certain amount of appended bytes relative to the original file size, the classification accuracy begins to decrease.

**Preserving Malware Functionality:** Another interesting point to note is the reasoning behind the methods utilized to generate the adversarial samples used in this study. Adversarial samples are created against MalConv as this method ensures that the actual malware binary is modified and that malware functionality remains intact. In the case of the image-based classifier, if adversarial samples are created for the image-based classifier, they would be adversarial *image* samples and not adversarial *malware* samples. Even if these adversarial *images* were converted back to binaries, there is no guarantee that the original functionality of the malware is preserved. This is because adversarial attacks on images are not localized to specific regions of the image, i.e perturbations can be added in any region of the image and this may break malware functionality as these perturbations could correspond to adding bytes to executable portions of malware code.

**Choice of Attacks and Novelty:** Although there are a variety of attacks in the literature that claim to generate adversarial malware, the attack methods chosen for this study are the only ones that undoubtedly retain malware functionality. In addition to retaining malicious functionality, the two black-box attacks used in this study are novel techniques not seen in previous literature. Methods aside

from the byte-append attacks described in Section 4.4, such as feature modification attacks [AKF+18, FRZ+, CSD19, CSR19, LCBDR19] and malicious code append attacks [FRZ+,PNX] do not guarantee the preservation of malware functionality. Moreover, the testing of functionality for adversarial malware samples crafted from these methods would be difficult as it would require dynamic analysis in a sandbox environment and a significant portion of malware samples do not run in such virtualized environments [XQE16, RSRE18]. This is done in order to hinder dynamic analysis and prevent malware testers from extracting run-time features of malware samples.

**Benefits:** Our image-based malware classifier outperformed MalConv, a widely used raw-byte based malware classifier substantially in all recorded accuracy metrics. It remained robust to adversarially crafted malware samples across all 4 attack scenarios while entailing significantly lower overhead as compared to MalConv. In addition, since all adversarial malware samples retained their malicious functionality, our image-based classifier was tested under realistic circumstances. This means that this method's potential applicability in the real-world is promising.

## 4.6 Conclusion

As the number of malware samples in the wild continue to increase at an alarming rate, adversaries continue to discover means to mask malware with perturbations so as to evade malware classifiers. To remedy this, we proposed a novel application/use of image-based classifiers to the malware domain in order to develop a model that remains robust to adversarial perturbations. For our evaluations, we used real adversarial malware samples from nine different families of malware in the wild. The results of our study indicate that our image-based classifier outperformed the state-of-the-art ML-based malware classifier, MalConv, in all regards including detection accuracy,

evasion rate of crafted adversarial samples, and computational overhead. Therefore, our proposed technique is resilient to adversarial malware samples and can pave the way for the development of other malware detection mechanisms that are resilient to adversarial perturbations. In addition, it does not require adversarial training; hence, it remains immune to model poisoning. Further, to our knowledge, this is the first work in the adversarial malware literature that demonstrates and analyzes the robustness of image-based classifiers against adversarial attacks.

CHAPTER 5

# A NOVEL AND LIGHTWEIGHT REAL-TIME CRYPTOJACKING DETECTION SYSTEM

## 5.1 Introduction

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

We designed and implemented the MINOS lightweight cryptojacking detection system as an end-to-end framework. We trained and evaluated the underlying CNN using one of the most comprehensive collected datasets on cryptojacking malware samples in the wild. The results of this evaluation conclude that the classifier achieves extremely high accuracy with no false positives or negatives. Our analysis of the MINOS framework shows that MINOS is able to accurately and almost instantaneously identify the presence of mining activity/cryptojacking malware while consuming minimal computational resources.

***Summary of Contributions:*** The main contributions of this chapter are noted as follows:

- A novel cryptojacking detection mechanism that implements a Wasm binary classifier in a lightweight, computationally inexpensive, and incredibly fast end-to-end framework - MINOS
- A novel Wasm binary classification technique that utilizes gray-scale image representations of the binaries to train a convolutional neural network.
- Our extensive evaluation demonstrates that MINOS is capable of detecting Wasm-based cryptojacking with 100% accuracy and very low overhead. Specifically, the proposed detection mechanism successfully detected all cryptojacking malware in our dataset within 25.9 milliseconds on average with only 6.5% and 4% maximum utilization of RAM and CPU, respectively.

## 5.2 Difference from Prior Works

Our work differs from the existing work in several ways: (1) It does not rely on dynamic analysis features, hence it does not require the mining samples to run for a specified period of time for feature collection and detection purposes. Also, (2) the performance of MINOS is not affected by third-party applications running on the host or on the browser do not affect the performance of MINOS. In addition, (3) common evasion techniques used by adversaries (e.g., CPU throttling, dynamically generated domain names, proxies, encrypted communication, etc.) are not effective to bypass MINOS's detection. Unlike existing detection systems, (4) the proposed detection technique does not have high runtime overhead, thus promises a better quality of web surfing experience for end-users compared to other schemes. Further, (5) MINOS does make use of browser or operating system specific features/tools, which makes

it platform independent. Moreover, unlike earlier work, (6) it is not necessary for MINOS to work to have administrative privileges to run on any specific platform. Finally, (7) MINOS represents the first work in the literature that classifies malicious and benign Wasm binaries using gray-scale image representations of the cryptojacking malware.

## 5.3 Threat Model

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

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

In addition to the methods used to inject cryptojacking scripts, the attacker can use dynamically generated domain names, proxies, alternative communication protocols (e.g., XMLHttpRequest) and encrypted communications to obfuscate the network-level behavior of his/her script. Moreover, the attacker can configure his/her script to throttle the CPU usage of cryptojacking. In addition, the attacker can obfuscate the strings in his/her cryptojacking scripts and change the function names. However, we do not assume that the attacker applies any further obfuscation attempts, since it may negatively affect the mining performance of cryptojacking malware. In

fact, this assumption was validated in SEISMIC [WFX+18], where Wang et al. noted that the only obfuscation techniques they encountered were name obfuscations.

The attacker is assumed to have the major implementation of the cryptojacking malware in WebAssembly due to the performance superiority of WebAssembly over JavaScript. We find this assumption reasonable since many studies [KVM+18, MWJR19, KMM+19] pointed out the fact that almost all of the cryptojacking scripts that were detected in the wild were based on WebAssembly. For this reason, although it is possible, we do not consider the attacker employing pure JavaScript-based cryptojacking implementation. In addition, we do not target legitimate cryptocurrency mining operations (e.g., UNICEF [uni]) in the browsers that take informs the end-users and asks for their consent.

## 5.4 MINOS Framework

### 5.4.1 Inherent Similarity of Cryptojacking Malware

Cryptojacking malware implementations have several inherent similarities as highlighted by Wang et al. [WFX+18] and Musch et al. [MWJR19]. They are constrained by optimized implementations of specific proof-of-work (PoW) schemes based on memory or CPU-bound hash puzzles. Although cryptojacking malware authors can employ various tactics to prevent their malware from being detected, they still have to implement the same PoW schemes with the same hashing algorithms. This is in fact, one of the unique characteristics of cryptojacking that distinguishes it from other malware families. Hence, we hypothesized that their implementations should share similar characteristics and maybe even look syntactically and semantically similar to each other. In fact, the semantic similarity of cryptojacking malware strains was confirmed by Wang et al. [WFX+18]. In order to verify the validity of our hy-

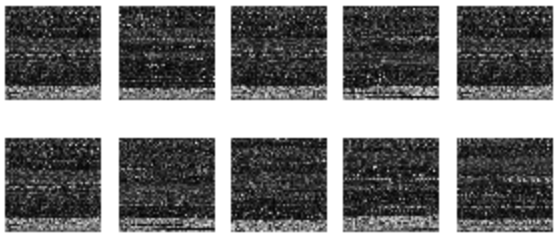

Figure 5.1: Gray-scale background image of Wasm binaries that belong to cryptojacking samples.

pothesis, we collected Wasm binaries of both benign and cryptojacking samples and converted them to gray-scale background image, as depicted in Fig. 5.1. As shown in figure 5.1, gray-scale background image of Wasm-binaries belonging to cryptojacking samples (i.e., the gray-scale background image in the first row) are visually very similar to each other. Based on this observation, and motivated by the drawbacks of existing detection mechanisms, we propose MINOS, a novel, lightweight cryptojacking detection system, in this study.

### 5.4.2 System Model

The architecture of MINOS framework, as shown in Figure 5.2, consists of four primary components: *Wasm module auto-collector, preprocessor, Wasm classifier, and notifier.*

The first component of MINOS is the Wasm Module Auto-Collector. As the user is browsing the Internet, this component checks if the website being visited currently produce any Wasm binaries, and if so, downloads them to a specified folder (❶). The second part is the preprocessor, which reads the specified folder where the Auto-collector downloads the Wasm binaries, and converts each binary in the folder to a gray-scale image (❷). It further preprocesses this image into a format that can be read by the next component. In the third part, the transformed binaries are input to

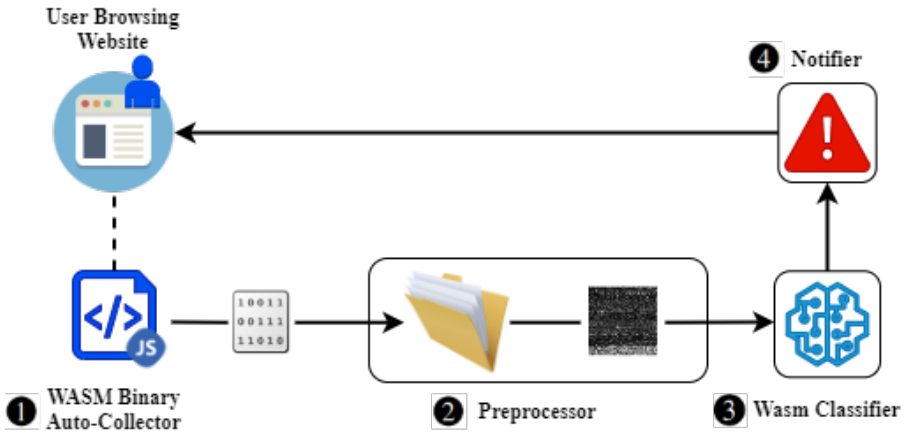


Figure 5.2: Overview of the MINOS's framework detailing its four components. The Wasm binary auto-collector downloads Wasm binaries to a designated folder. The preprocessor then converts the binaries to gray-scale background image and

the Wasm Classifier, a pre-trained CNN that classifies each preprocessed binary as either malicious or benign (❸). Finally, the notifier receives the classification results from the CNN and, based on those results, will either alert the user of malicious mining activity or do nothing (❹).

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

**Preprocessor:** Before the Wasm classifier can perform its task, the extracted Wasm binary needs to be preprocessed to convert the data into a format that the neural network is able to use as input. This involves converting each Wasm binary to a

gray-scale image and resizing it to a common size. The preprocessor converts the binary into an array of integers with each integer representing a pixel of a gray-scale image and then normalizes and reshapes the resulting array. The final reshaped array is then passed on to the Wasm classifier.

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

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

## 5.5 Implementation of MINOS

This section provides details regarding the dataset used in the study, as well as the technical implementation of the MINOS framework. The framework is implemented

as an application written in Python 3 in only 90 lines of code, with a supplemental script written in node.js. The implementation is performed and evaluated on a system running Ubuntu 18.04 with an Intel Core i7-9700K processor and 32 GB of available RAM. The system has a total of 8 cores, with each processor running a base frequency of 3.6 GHz. The current implementation is designed to work specifically with Google Chrome, as it is the world's most used web browser. The following subsections provide an in-depth look into each component and outline details, including algorithms involved and libraries/APIs utilized.

### 5.5.1 Dataset Collection & Breakdown

The dataset used to train the Wasm Classifier consists of 150 malicious and 150 benign Wasm binaries that were obtained from numerous studies and resources. A large portion of the dataset consists of binaries that were collected and used in the following other studies in the literature: SEISMIC [WFX+18], MineSweeper [KVM+18] and Musch Et al. [MWJR19]. The remainder of the binaries were collected manually using resources such as VirusTotal [virb] and VirusShare [vira], NoCoin [noc] and MadeWithWasm [mad]. A breakdown of the number of binaries collected from each resource, including how many of them were benign and malicious, can be found in table 5.1 below. This is followed by a brief description of the other resources and how the binaries were collected from each of them.

**VirusTotal and VirusShare:** VirusTotal and VirusShare are two popular websites that are used for malware research and practice purposes (e.g., scanning, sharing). Both websites provide various malware samples along with their detection results with respect to several antivirus engines. In order to obtain the newest samples that were detected by antivirus engines, both VirusTotal and VirusShare were used in a way that complemented each other. In this respect, the VirusTotal API enables researchers to

Table 5.1: DATASET BREAKDOWN

| | **Benign** | **Malicious** |
|---|---|---|
| **SEISMIC [WFX+18]** | 6 | 4 |
| **MineSweeper [KVM+18]** | 4 | 34 |
| **Musch et al. [MWJR19]** | 105 | 45 |
| **VirusTotal and VirusShare** | 0 | 53 |
| **NoCoin [noc]** | 0 | 4 |
| **MadeWithWasm [mad]** | 35 | 0 |
| **Total** | **150** | **150** |

automatically check the detection results for a specific sample (up to 4 queries per minute) but does not allow downloading of such samples. VirusShare, on the other hand, enables researchers to download large packages of malware, but does not have an API. To obtain the newest malware samples, a 73.15 GB malware package shared with the community on the 21st of April, 2020 was downloaded from VirusShare, and hash values of the samples that have the file type of HTML were extracted. Since HTML samples may have various malware (e.g., redirector, downloaded, ramnit, trojans, etc.) inside, the VirusTotal Premium API was used to determine the ones labeled as malicious cryptocurrency mining script. Using a bash script the hash values obtained from VirusShare samples were checked using the VirusTotal specific API in every 15 seconds. If the VirusTotal API indicates that the sample was identified as cryptojacking malware, then the sample was automatically extracted from the local VirusShare package and using the Puppeteer API [pup], opened in Google Chrome in incognito mode with developer options to examine if it compiles and runs Wasm for cryptojacking.

Using the developed script, we were able to determine 34 unique websites (adult, streaming, forum, etc.) that still perform cryptojacking operation using Wasm. We would like to note that the same web pages and also different pages of the same websites were detected cryptojacking positive by antivirus engines at different time intervals. For instance, antivirus engines detected *piratebay*'s individual torrent search

result pages to be employing cryptojacking. However, that website was injecting the same cryptojacking script to every page of it. Therefore, we omitted multiple occurrences of same websites and also multiple webpage occurrences of the same websites and counted them as 1. We analyzed the samples and found out the addresses of 8 unique mining services still operating. Table 5.2 outlines the list of active mining services that use Wasm. When we checked the domain names, we found out that except for *bimeq.com.vn* and *monero.cit.net*, rest of the domains already reside in the list of NoCoin [noc]. In addition, we realized that the VirusShare malware package that was shared with the community in April 2020 has several cryptojacking samples that try to reach the already down Coinhive mining services. Since it is not available anymore, that samples do not perform any mining operations. Nevertheless, antivirus engines seem to detect them as cryptojacking malware since they have specific keywords (i.e., Coinhive, miner, mine, etc.). We also found out that some samples which do not perform any mining operation and which do not have any mining-related script declarations were falsely detected as cryptojacking samples merely probably due to having specific keywords in the actual HTML text.

Table 5.2: Cryptojacking Services Extracted from VirusTotal and VirusShare Samples

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

**NoCoin:** NoCoin [noc] is a browser extension available on Chrome, Firefox, and Opera, that aims to block websites that mine cryptocurrency, using a blacklist. Each website on the blacklist was visited using Google Chrome, one-by-one, to check for

the presence of mining activity. This was achieved by opening Chrome DevTools and manually checking for the instantiation of Wasm modules in the "Sources" tab. Since the blacklist is relatively old, most of the websites no longer exist or are not mined in cryptocurrency. Four of the websites still functioned, and as such, the Wasm modules were downloaded. These modules are downloaded in the Web Assembly textual format rather than in binary form, and therefore must be converted to a binary. The conversion is performed using the WebAssembly Binary Toolkit (WABT), specifically, the `wat2Wasm` tool.

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

### 5.5.2 Wasm Module Auto-Collector

A script written in node.js provided by the authors of [MWJR19] automatically collects and downloads Wasm binaries to a specified folder as the user is browsing the web. It utilizes Puppeteer, a Node library that is able to communicate with and manipulate/control Google Chrome over the DevTools Protocol, through a high-level API. The code wraps JavaScript functions such as `WebAssembly.instantiateStreaming`, that instantiate a WebAssembly module, and logs the module's binary file to the NodeJS backend.

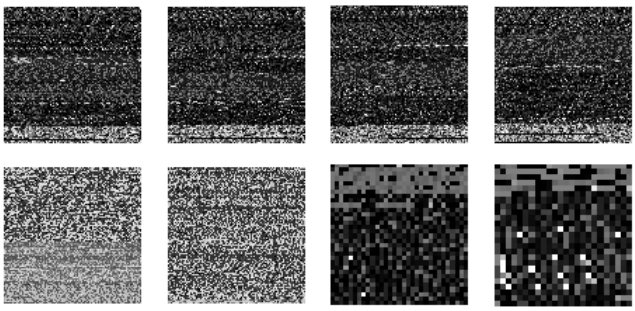

Figure 5.3: The first row of gray-scale images belong to Wasm binaries of cryptocurrency mining webpages. The images in the second row represent Wasm binaries of benign webpages, primarily games that employ Wasm.

### 5.5.3 Preprocessor

A Wasm module binary consists of a sequence of hexadecimal numbers. This vector of hexadecimal values can be modified and transformed into a gray-scale image. To facilitate such a conversion, the Wasm binary is first converted into a vector of 8-bit unsigned integers (uint8) and then reshaped into a two-dimensional array. This reshaped array is then divided by 255 to represent each integer as a pixel that takes a value ranging from 0 to 255 (with 0 being black, and 255 being white). These pixels together form a gray-scale image representation of the Wasm module binary. depicts an example of a gray-scale image of a Wasm binary that utilizes the CoinHive cryptocurrency mining service. As shown in the figure, visually distinct regions in the gray-scale image correspond to specific sections in the Wasm binary structure. Examples of binary to image transformations of malicious and benign webpages were shown in Figure 5.3. As it can be observed from Figure 5.3, binaries that mine cryptocurrency are visually extremely similar when converted to gray-scale images. This observation also holds true for benign binaries. Another observation is that the images of malicious Wasm binaries are distinct from those belonging benign webpages.

The visualization procedure is implemented using a recursive function written in Python 3, the details of which are depicted in Algorithm 3. In Lines 2-3, a while loop

**Algorithm 2:** Preprocessor

```
1  def preprocess():
2      while len(Wasm_directory) == 0 do
3          time.sleep(1)
4      if len(Wasm_directory)! = 0 then
5          Wasm_images → []
6          for file in Wasm_directory do
7              f → open(file)
8              ln → getSize(file)
9              width → math.pow(ln, 0.5)
10             rem → ln%width
11             a → array('B')
12             a.fromfile(f, ln − rem)
13             f.close()
14             os.remove(file)
15             g → reshape(a, (len(a)/width), width)
16             g → uint8(g)
17             h → resize(g, size, size)
18             h → h/255
19             h → h.reshape(−1, 100, 100, 1)
20             Wasm_images(h)
21     classify(Wasm_images)
22 return preprocess()
```

is constantly checking whether the destination folder for extracted Wasm binaries is empty at 1 second intervals (i.e., it is checking every second). In Lines 4-7, once a binary is collected and added to the folder, it is opened and ready for further preprocessing. In line 5, the variable $Wasm_images$ is declared, which will store the converted images. Line 6 ensures that if a website loads multiple Wasm modules, each downloaded module is visited in each iteration of the for loop and is preprocessed. Lines 8-10 calculate length and width parameters, ensuring that the final resized array will have a relatively similar length and width. In Line 11, the file is converted to an array of unsigned integers. In Lines 13-14, after the array is reshaped and the file is closed, the file is deleted from the directory so that once the function has executed, it does not preprocess the same module repeatedly. Lines 15-16 reshapes the created

array and converts it into an array of 8-bit unsigned integers (uint8) that range in values from 0 to 255. The value of each integer in the array represents the brightness of a pixel ranging from black to white (0 to 255). In Lines 17-18, the image array is resized to a common size of 100 by 100, and the pixel values are normalized to a range of 0 to 1 by dividing the array by 255. This normalization is done as it is easier for the model to process input arrays with a smaller range of values. Line 19 reshapes the array to a 4-dimensional array that the CNN can accept as input, and Line 20 appends the array to the $Wasm_images$ list. Once the binary or binaries are converted to images and added to this list, the $classify()$ function is called, which takes the images as an argument. This function will be referenced to and discussed in the following subsections. The *preprocess* function ends with a recursive call ensuring that it continues to check the directory at 1 second intervals for new input.

### 5.5.4 Convolutional Neural Network & Notifier

**Algorithm 3:** Classification Retriever

```
1 def classify(images):
2         results → []
3         for ima in images do
4             results.append(model.predict_classes(ima))
5         if 1 in results then
6             notify_user()
7 return
```

The Wasm classifier is built using the TensorFlow software library, specifically using TensorFlow's high-level API, Keras. The CNN is written in Python 3 using TensorFlow version 1.13.1. The model is trained across 100 epochs using the RMSprop optimizer, with the learning rate manually set to 0.0001.

In Lines 24-27 in the $classify()$ function of Algorithm 3, the classification result of each binary collected and converted to images by the *preprocess* function, is re-

trieved. In Line 2, the variable *results* is declared, which will store the results of the classification of each binary. In line 3, the `model.predict_classes()` Keras function is used to classify each image in $Wasm\_images$. The function returns an integer representing whether the sample is benign (0) or malicious (1). Each classification result is appended to the *results* list that the notifier will use to perform its task.

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

## 5.6 Performance Evaluation

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

### 5.6.1 Performance of Wasm Classifier

The performance of the Wasm classifier was evaluated based on a number of metrics, including accuracy, optimization loss, and true positive and false positive rate. The dataset was divided into training and testing sets using an 80/20 split. In figure 5.4(a), it can be seen that the model converges to 100% accuracy on both the training and test sets after 30 epochs. The optimization learning curve in figure 5.4(b) shows that both the training and testing loss decrease to a point of stability after approximately

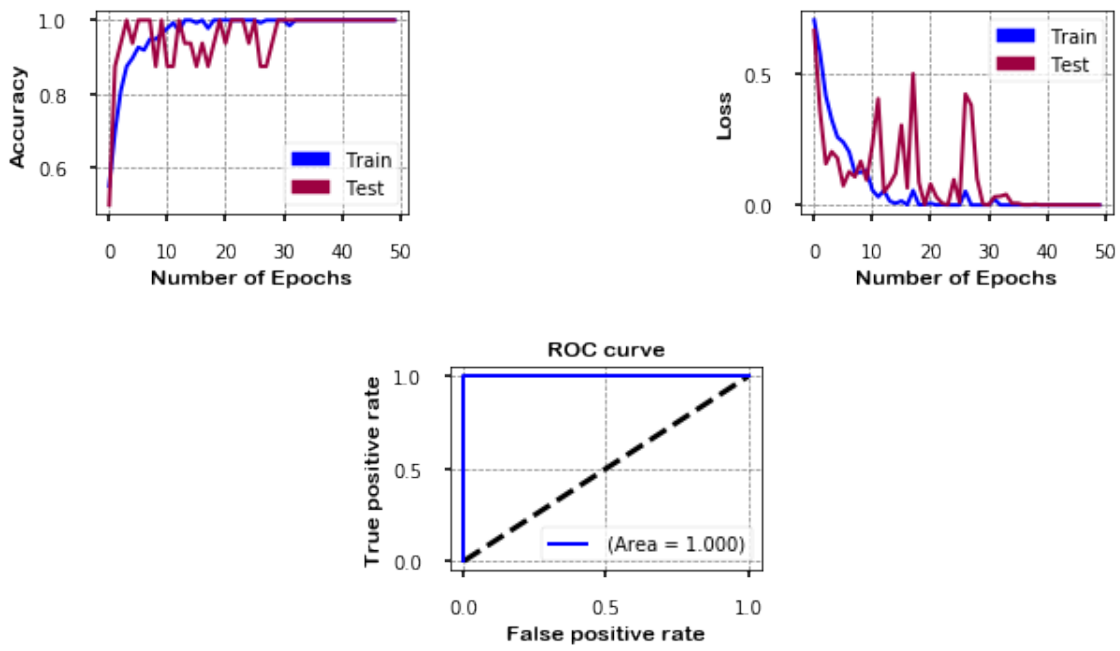


Figure 5.4: Performance metrics of MINOS framework. Accuracy, learning curve, and ROC curve of MINOS are depicted respectively.

30 epochs. This indicates that the model is neither overfitting nor underfitting and therefore is able to generalize effectively. Figure 5.4(c) displays the receiver operating characteristic (ROC) curve for the classifier. The area under the ROC curve is 1, indicating that the model's ability to distinguish between the two classes (benign and mining) is perfect. In addition, this also signifies that the test set's classification results contained no false positives or false negatives.

### 5.6.2 Overhead Analysis of MINOS Framework

**Training Wasm Classifier:** In order to train the classifier, each Wasm binary in the dataset was converted into a gray-scale image. During this preprocessing stage, when the Wasm binaries are converted to images, 4.3% of RAM is utilized with the conversion of all binaries, taking 0.497 seconds. The entire dataset is stored in virtual memory as an array of arrays taking up a mere 1.448 Kb of space. The total time

taken to build, compile, and train the model in 50 epochs was 24.8 seconds. During the training process, a maximum of 4.1% of RAM was used, and no more than 52% of the CPU's processing power was in use.

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

## 5.7 Discussion and Benefits

**Lightweight Runtime Overhead:** As is evident by the results of our performance evaluation, the overhead incurred during the implementation and actual runtime of the Minos framework is extremely minimal. While other detection systems in the literature report similar accuracy, their detection methods are based on the analysis of dynamic features. This means that in both the data collection stage, as well during the implementation of these detection methods, the webpages that use cryptojacking

malware were allowed to run and effectively mine cryptocurrency until the required features are extracted or until the respective detection method is able to detect the presence of mining activity. Further, the actual collection of features in such dynamic-analysis based systems requires additional resources or supplemental applications and programs. Since MINOS does not use dynamic features, it does not require the webpages that instrument cryptojacking malware to run for a certain amount of time to extract relevant features.

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

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

**Platform Independence:** MINOS does not utilize browser-specific or operating system-specific features/tools. Although the majority of existing detection systems are similar to MINOS in this respect, not every detection system fulfills this condition. For instance, OUTGUARD [KMM+19] relies on browser-specific features which limits its application in other widely used browsers.

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

**Quality of Web Surfing Experience:** MINOS has extremely low runtime overhead and successfully detects the cryptojacking scripts even before they start the mining process. However, existing detection systems either instrument and watch the Wasm modules of every web application, or continously monitor CPU, memory and network events which may affect the quality of web surfing experience of end-users. We believe that quality of web surfing experience is a crucial metric for the success of any cryptojacking detection system. Cryptojacking detection systems should not sacrifice from the quality of web surfing experience of end users. In addition, since more and more benign applications (e.g., Autocad [aut]) are moving to web thanks to WebAssembly and other technologies (e.g., WebWorkers, WebSockets), performance of such benign web applications needs to be ensured.

## 5.8 Conclusion

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

# CHAPTER 6

# CONCLUDING REMARKS AND FUTURE WORK

In this thesis, we introduced novel deep-learning based frameworks that are robust against adversarially crafted P.E. malware as well as cryptojacking malware. First, we provided detailed preliminary information regarding the current ecosystem of adversarial attacks and different classifiers in the current literature. This also included cryptojacking detection systems and the shortcomings that those systems have. Based on this, we developed two novel frameworks that are able to accurately and efficiently detect P.E. and cryptojacking malware that are resilient to adversarial perturbations, and result in minimal computational overhead.

To build a system that is robust to adversarial perturbations applied to P.E malware, we propose the use of image-based malware detection. We show that converting the malware detection problem into image-based malware classification problem provides robustness against adversarial perturbations. The underlying robustness stems from the fact that adversarial attacks, which are relatively easy to apply to images in the computer vision domain, are extremely difficult to apply to transformed images of malware samples. This is because such an operation that adds adversarial noise to a malware image has a very high possibility of breaking the functionality of the actual malware when the image is converted back to a malware binary. To evaluate our proposed solution, we trained a convolutional neural network (CNN)-based classifier using the gray-scale images of the malware in the dataset. We then applied adversarial attacks including our two novel attacks to our image-based malware classifier. The results of our evaluation show that the image-based malware detection approach is robust against adversarial attacks that can easily fool a state-of-the-art ML-based malware detector.

Moreover, in order to overcome the problems associated with current cryptojacking detection systems in the literature, we developed MINOS, a novel lightweight cryptojacking detection system that employs the use of gray-scale image representations of Wasm-binaries in web browsers. We trained and evaluated the underlying CNN using one of the most comprehensive collected datasets on cryptojacking malware samples in the wild. The results of this evaluation conclude that the classifier achieves extremely high accuracy with no false positives or negatives.

We present several key directions for future research:

• In this thesis, we presented a novel framework that remains robust to adversarial P.E. malware. Although we implemented the framework on a system in real-time using real P.E. samples, adversarially crafted inputs are still an ongoing issue in other types of malware. We believe that further lresearch should be done in implementing this framework to be used with other malware such as Android and PDF malware.

• A novel cryptojacking detection framework, Minos, is introduced in this thesis that is capable of detecting the presence of cryptojacking malware nearly instantly and with negligible overhead. The entire framework in its current form, consists of a Python application that runs in the background of the user's computer. However, we must consider that there will be users that do not have Python installed on their systems and/or are unable to run the application on their system. To remedy this, for future considerations, the Minos framework should be incorporated as a browser plug-in, thereby removing the need for any prerequisites that could hinder the operation of the framework and will allow for automatic interruption of webpages that are mining cryptocurrency.

• Finally, this thesis presents a novel cryptojacking detection technique that uses image representations of Wasm binaries of webpages that incorporate cryptojacking malware. This image classification technique works with the underlying assumption

that the majority of cryptojacking malware execute the same intructions on a binary level in order to compute hashes and therefore mine cryptocurrency. However, this study primarily deals with a single form of cryptocurrency, Monero. There is a strong likelihood that other cryptocurrencies may also be mined in a similar fashion in the future. Therefore, in the future, with the advent of other cryptocurrencies, the proposed detection should be used to classify malicious Wasm binaries according to the specific cryptocurrency being mined.

# BIBLIOGRAPHY


[adg] Crypto-streaming strikes back. `https://adguard.com/en/blog/crypto-streaming-strikes-back.html`. Accessed: 2020-06-02.

[AHHO18] A. Al-Dujaili, A. Huang, E. Hemberg, and U. O'Reilly. Adversarial deep learning for robust detection of binary encoded malware. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 76–82, May 2018.

[AKF+18] Hyrum S Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. Learning to evade static pe machine learning malware models via reinforcement learning. *arXiv preprint arXiv:1801.08917*, January 2018.

[aut] Tutocad webassembly: Moving a 30 year code base to the web. `https://www.infoq.com/presentations/autocad-webassembly/`. Accessed: 2020-04-13.

[BBCC19] Daniel S. Berman, Anna L. Buczak, Jeffrey S. Chavis, and Cherita L. Corbett. A survey of deep learning methods for cyber security. *Information*, 10(4), 2019.

[BBD19] Hugo L. J. Bijmans, Tim M. Booij, and Christian Doerr. Just the tip of the iceberg: Internet-scale exploitation of routers for cryptojacking. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 449–464, New York, NY, USA, 2019. Association for Computing Machinery.

[BC19] Gabriel Jozsef Berecz. and Istvan-Gergely Czibula. Hunting traits for cryptojackers. In *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications - Volume 2: SECRYPT,*, pages 386–393. INSTICC, SciTePress, 2019.

[BMZ20] Weikang Bian, Wei Meng, and Mingxue Zhang. Minethrottle: Defending against wasm in-browser cryptojacking. In *Proceedings of The Web Conference 2020*, WWW '20, page 3112–3118, New York, NY, USA, 2020. Association for Computing Machinery.

[BSK19] I. Baptista, S. Shiaeles, and N. Kolokotronis. A novel malware detection system based on machine learning and binary visualization. In *2019 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6, 2019.

[CBOS20] D. Carlin, J. Burgess, P. O'Kane, and S. Sezer. You could be mine(d): The rise of cryptojacking. *IEEE Security Privacy*, 18(2):16–22, 2020.

[CGLP19] Mauro Conti, Ankit Gangwal, Gianluca Lain, and Samuele Giuliano Piazzetta. Detecting covert cryptomining using hpc. *arXiv preprint arXiv:1909.00268*, 2019.

[CHY17] Lingwei Chen, Shifu Hou, and Yanfang Ye. Securedroid: Enhancing security of machine learning-based detection against adversarial android malware attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, page 362–372, New York, NY, USA, 2017. Association for Computing Machinery.

[CRY+19] B. Chen, Z. Ren, C. Yu, I. Hussain, and J. Liu. Adversarial examples for cnn-based malware detectors. *IEEE Access*, 7, 2019.

[CSD19] R. L. Castro, C. Schmitt, and G. Dreo. Aimed: Evolving malware with genetic programming to evade detection. In *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)*, pages 240–247, Aug 2019.

[CSR19] R. L. Castro, C. Schmitt, and G. D. Rodosek. Armed: How automatic malware modifications can evade static detection? In *5th International Conference on Information Management (ICIM)*, March 2019.

[CW17] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57, May 2017.

[CW18] N. Carlini and D. Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 1–7, May 2018.

[CYB17] L. Chen, Y. Ye, and T. Bourlai. Adversarial machine learning in malware detection: Arms race between evasion attack and defense. In *2017 European Intelligence and Security Informatics Conference (EISIC)*, pages 99–106, Sep. 2017.

[EH13] Christopher C. Elisan and Mikko Hypponen. *Malware, rootkits botnets: a beginners guide*. McGraw-Hill, 2013.

[ELMC18] S. Eskandari, A. Leoutsarakos, T. Mursch, and J. Clark. A first look at browser-based cryptojacking. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pages 58–66, 2018.

[ERLD18] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. HotFlip: White-box adversarial examples for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 31–36, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[FRZ+] William Fleshman, Edward Raff, Richard Zak, Mark McLean, and Charles Nicholas. Static malware detection & subterfuge: Quantifying the robustness of machine learning and current anti-virus. In *Proceedings of the AAAI Symposium on Adversary-Aware Learning Techniques and Trends in Cybersecurity (ALEC 2018) Arlington, Virginia, USA, October 18-20, 2018*, pages 3–10.

[FXW+18] J. Fu, J. Xue, Y. Wang, Z. Liu, and C. Shan. Malware visualization for fine-grained classification. *IEEE Access*, 6:14510–14523, 2018.

[GMP18] Ian Goodfellow, Patrick McDaniel, and Nicolas Papernot. Making machine learning robust against adversarial inputs. *Communications of the ACM*, 61(7):56–66, June 2018.

[GMP20] Daniel Gibert, Carles Mateu, and Jordi Planes. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, 153:102526, 2020.

[GSS15] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA*, 2015.

[HHD+20] H. HDarabian, S. Homayounoot, A. Dehghantanha, S. Hashemi, H. Karimipour, R. M. Parizi, and K. K. Raymond Choo. Detecting cryptomining malware: a deep learning approach for static and dynamic analysis. *J Grid Computing*, 2020.

[HLI13] KyoungSoo Han, Jae Hyun Lim, and Eul Gyu Im. Malware analysis method using visualization of binary files. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, RACS '13, page

317–321, New York, NY, USA, 2013. Association for Computing Machinery.

[HT17] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on GAN, 2017.

[HYY+18] Geng Hong, Zhemin Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Haixin Duan. How you get shot in the back: A systematical study about cryptojacking in the real world. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1701–1713, New York, NY, USA, 2018. Association for Computing Machinery.

[KAC+19] Aminollah Khormali, Ahmed Abusnaina, Songqing Chen, DaeHun Nyang, and Aziz Mohaisen. Copycat: Practical adversarial attacks on visualization-based malware detection, 2019.

[KBA+18] Felix Kreuk, Assi Barak, Shir Aviv, Moran Baruch, Benny Pinkas, and Joseph Keshet. Deceiving end-to-end deep learning malware detectors using adversarial examples. In *NeurIPS 2018 Workshop on Security in Machine Learning, Montreal, Canada, December 7*, 2018.

[KBRS20] Conor Kelton, Aruna Balasubramanian, Ramya Raghavendra, and Mudhakar Srivatsa. Browser-Based Deep Behavioral Detection of Web Cryptomining with CoinSpy. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[KDB+18] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *26th European Signal Processing Conference (EUSIPCO)*, Rome, 09 2018. IEEE.

[KM13] K. Kancherla and S. Mukkamala. Image visualization based malware detection. In *2013 IEEE Symposium on Computational Intelligence in Cyber Security (CICS)*, pages 40–44, 2013.

[KMM+19] Amin Kharraz, Zane Ma, Paul Murley, Charles Lever, Joshua Mason, Andrew Miller, Nikita Borisov, Manos Antonakakis, and Michael Bailey. Outguard: Detecting in-browser covert cryptocurrency mining in the wild. In *The World Wide Web Conference*, WWW '19, page

840–852, New York, NY, USA, 2019. Association for Computing Machinery.

[Kre18] Brian Krebbs. Who and what is coinhive? `https://krebsonsecurity.com/2018/03/who-and-what-is-coinhive/`, March 2018. Accessed: 2020-06-03.

[KVM+18] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1714–1730, New York, NY, USA, 2018. Association for Computing Machinery.

[LCBDR19] Raphael Labaca-Castro, Battista Biggio, and Gabi Dreo Rodosek. Poster: Attacking malware classifiers by crafting gradient-attacks that preserve functionality. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2565–2567, New York, NY, USA, 2019. Association for Computing Machinery.

[lie20] Library to instrument executable formats. https://lief.quarkslab.com/, 2020. [Online; accessed 2-February-2020].

[los] Cryptojacking attack found on los angeles times website. `https://threatpost.com/cryptojacking-attack-found-on-los-angeles-times-website/130041/`. Accessed: 2020-06-02.

[LZLL19] Xinbo Liu, Jiliang Zhang, Yaping Lin, and He Li. Atmpa: Attacking machine learning-based malware visualization detection methods via adversarial examples. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[mad] Made with webassembly. `https://madewithwebassembly.com/`. Accessed: 2020-06-03.

[MFF16] S. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2574–2582, 2016.

[MTICGN19] Javier Martínez Torres, Carla Iglesias Comesaña, and Paulino J. García-Nieto. Review: machine learning techniques applied to cybersecurity. *International Journal of Machine Learning and Cybernetics*, 10(10):2823–2836, Oct 2019.

[MWJR18] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. Web-based cryptojacking in the wild. *CoRR*, abs/1808.09474, 2018.

[MWJR19] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. New kid on the web: A study on the prevalence of webassembly in the wild. In *Proc. of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2019.

[NKJM11] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. Malware images: Visualization and automatic classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, VizSec '11, New York, NY, USA, 2011. Association for Computing Machinery.

[NLFM20] Helio N. C. Neto, Martin Andreoni Lopez, Natalia Castro Fernandes, and Diogo M. F. Mattos. Minecap: super incremental learning for detecting and blocking cryptocurrency mining on software-defined networking. *Ann. des Télécommunications*, 75(3-4):121–131, 2020.

[noc] Nocoin adblock list. `https://github.com/hoshsadiq/adblock-nocoin-list`. Accessed: 2020-06-03.

[NQZ18] Sang Ni, Quan Qian, and Rui Zhang. Malware identification using visualization images and deep learning. *Computers Security*, 77:871 – 885, 2018.

[pec20] Intentional PE Corruption. `https://blog.malwarebytes.com/cybercrime/2012/04/intentional-pe-corruption/`, 2020. [Online; accessed 15-April-2020].

[PEF20] Pe format. https://docs.microsoft.com/en-us/windows/win32/debug/pe-format, 2020. [Online; accessed 8-February-2020].

[PKY19] D. Park, H. Khan, and B. Yener. Generation evaluation of adversarial examples for malware obfuscation. In *2019 18th IEEE International*

*Conference On Machine Learning And Applications (ICMLA)*, pages 1283–1290, 2019.

[PMG16] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples, 2016.

[PMJ+16] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 372–387, 2016.

[PMW+16] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 582–597, 2016.

[PNX] Giorgos Poulios, Christoforos Ntantogian, and Christos Xenakis. ROPInjector: Using return-oriented programming for polymorphism and av evasion. In *Black Hat USA, 2015*.

[pup] Puppeteer. `https://pptr.dev/`. Accessed: 2020-06-03.

[QLZW19] Shilin Qiu, Qihe Liu, Shijie Zhou, and Chunjiang Wu. Review of artificial intelligence adversarial attack and defense technologies. *Applied Sciences*, 9(5), 2019.

[RBS+18] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K. Nicholas. Malware detection by eating a whole EXE. In *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7*, pages 268–276, 2018.

[rL14] N. rndic and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *2014 IEEE Symposium on Security and Privacy*, pages 197–211, May 2014.

[RP18] Juan D Parra Rodriguez and Joachim Posegga. Rapid: Resource and api-based detection against in-browser miners. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 313–326, 2018.

[RRF+18] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *CoRR*, abs/1802.10135, 2018.

[RSRE18] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In *Research in Attacks, Intrusions, and Defenses*, pages 490–510, Cham, 2018. Springer International Publishing.

[RTH+18] Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. Bringing the web up to speed with webassembly. *Commun. ACM*, 61(12):107–115, November 2018.

[SCJ19] O. Suciu, S. E. Coull, and J. Johns. Exploring adversarial examples in malware detection. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 8–14, May 2019.

[SH12] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. No Starch Press, 2012.

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

[sym19] ISTR internet security threat report. Technical report, Symantec, February 2019.

[SZS+14] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16*, 2014.

[UAB19] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers Security*, 81:123 – 147, 2019.

[uni] The hope page. `https://www.thehopepage.org/`. Accessed: 2020-04-13.

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

[VGOB20] S. Varlioglu, B. Gonen, M. Ozer, and M. Bastug. Is cryptojacking dead after coinhive shutdown? In *2020 3rd International Conference on Information and Computer Technologies (ICICT)*, 2020.

[vira] Virusshare. https://virusshare.com/. Accessed: 2020-06-03.

[virb] Virustotal - free online virus, malware and url scanner. https://www.virustotal.com/. Accessed: 2017-5-30.

[VNNT19] B. N. Vi, H. Noi Nguyen, N. T. Nguyen, and C. Truong Tran. Adversarial examples against image-based malware classification systems. In *2019 11th International Conference on Knowledge and Systems Engineering (KSE)*, pages 1–5, 2019.

[V9] Vladimír Veselý and Martin Žádník. How to detect cryptocurrency miners? by traffic forensics! *Digital Investigation*, 31:100884, 2019.

[was20] Webassembly. https://webassembly.org//, 2020.

[WFX+18] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W Hamlen, and Shuang Hao. Seismic: Secure in-lined script monitors for interrupting cryptojacks. In *European Symposium on Research in Computer Security*, pages 122–142. Springer, 2018.

[WLK+19] Xianmin Wang, Jing Li, Xiaohui Kuang, Yu-an Tan, and Jin Li. The security of machine learning in an adversarial setting: A survey. *J. Parallel Distributed Comput.*, 130:12–23, 2019.

[XEQ18] Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[XQE16] Weilin Xu, Yanjun Qi, and David Evans. Automatically evading classifiers: A case study on PDF malware classifiers. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[YKXG17] Wei Yang, Deguang Kong, Tao Xie, and Carl A. Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, page 288–302, New York, NY, USA, 2017. Association for Computing Machinery.

[YLAI17] Yanfang Ye, Tao Li, Donald Adjeroh, and S. Sitharama Iyengar. A survey on malware detection using data mining techniques. *ACM Comput. Surv.*, 50(3), June 2017.

[you] Youtube shuts down hidden cryptojacking adverts. `https://www.telegraph.co.uk/technology/2018/01/29/youtube-shuts-hidden-crypto-jacking-adverts/`. Accessed: 2020-06-02.

VITA

FARAZ AMJAD NASEEM

| | |
|---|---|
| 2014 - 2018 | B.S., Electrical Engineering<br>Florida International University<br>Miami, Florida |
| May 2017 - July 2017 | Software Engineer Intern<br>Cavium Inc.<br>San Jose, California |
| 2018 - 2020 | M.S., Computer Engineering<br>Florida International University<br>Miami, Florida |

SELECTED PUBLICATIONS, PATENTS, AND INVENTION DISCLOSURES

(Under Review) Faraz Naseem, Ahmet Aris, Leonardo Babun, A. Selcuk Uluagac, "Appearance matters, even for malware! A Lightweight Image-based Malware Classification System Resistant to Adversarial Machine Learning Attacks" ACSAC 2020- Annual Computer Security Applications Conference, December 2020.

(Under Review) Faraz Naseem, Ahmet Aris, Leonardo Babun, A. Selcuk Uluagac, "MINOS: A Novel and Lightweight Real-Time Cryptojacking Detection System" Oakland SnP 2021- 42nd Annual IEEE Symposium on Security and Privacy, May 2021

Faraz Naseem, Ahmet Aris, Leonardo Babun, A. Selcuk Uluagac, "Cspower-watch: A cyber-resilient residential power management system," in 2019 IEEE International Conference on Green Computing and Communications (GreenCom) July 2019, pp. 768–775. , December 2020

(Under Preparation) Faraz Naseem, Kemal Akkaya, Mark Weiss, A. Selcuk Uluagac, "Design of a Bonus-Based IoT Security Class" IEEE Transactions on Education

(Under Preparation) Faraz Naseem, Ahmet Aris, Leonardo Babun, A. Selcuk Uluagac, "Cspower-watch: A cyber- resilient residential power management system," IEEE Transactions on Greem Computing and Communications

(Under Preparation) Faraz Naseem, Ahmet Aris, Leonardo Babun, A. Selcuk Uluagac, *A Novel Fileless Malware Detection Mechanism*, invention disclosure being prepared to be submitted to Florida International University, 2020.