TÉCNICO LISBOA

Cryptojacking Detection with CPU Usage Metrics

Fábio Miguel de Jesus Gomes

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. Dr. Miguel Nuno Dias Alves Pupo Correia

Examination Committee

Chairperson: Prof. Dr. António Manuel Ferreira Rito da Silva
Supervisor: Prof. Dr. Miguel Nuno Dias Alves Pupo Correia
Member of the Committee: Prof. Dr. Pedro Miguel dos Santos Alves Madeira Adão

September 2020

Acknowledgments

First and foremost I would like to sincerely thank my advisor Professor Miguel Pupo Correia for his guidance during the process of developing and writing these Thesis, for being always available to help as much as possible, for not quitting on me and for being patient with me even tho most of the times I was not the model student. I've never known someone so dedicated to its students and with this much patience.

I would also like to thank my parents (Alice Antunes and António Gomes) and girlfriend (Joana Nogueira) who have always unconditionally supported and motivated me throughout this whole process, and to whom I owe my every achievement, since without their motivation and help I would probably not have completed this Thesis. My Theses is dedicated to them. I hope I can bring them as much joy and opportunities as they gave me my hole life.

I would also like to thank Integrity SA that gave me the time to pursuit my Masters and to see it through.

Abstract

Cryptojacking is currently being exploited and used by cyber-criminals. This form of malware runs in the computers of victims without their consent. It often infects browsers and does CPU-intensive computations to mine cryptocurrencies on behalf of the cyber-criminal, which takes the profits without paying for the resources consumed. Such attacks degrade CPU performance and potentially even shorten the hardware lifetime. We introduce a new cryptojacking detection mechanism based on the CPU usage of the visited web pages. Our detector monitors CPU usage metrics. This may look like a problematic way to detect the presence of mining malware since a lot of web resources are heavy computationally, which arguably would lead to false positives. However, by combining a set of CPU monitoring features and using machine learning, we manage to obtain metrics like precision and recall close to 1.

Keywords: Cryptojacking, Classifiers, Machine Learning, CPU, Malware

Contents

List of Tables

List of Figures

Nomenclature

API Application Program Interface

CPU Central Processing Unit

GPU Graphics Processing Unit

HIDS Host-based Intrusion Detection Systems

IDS Intrusion Detection System

PCM Performance Counter Monitor

PMU Performance Monitoring Units

Chapter 1

Introduction

The world has started to see a shift in the way revenue is generated online. A few years ago the only revenue stream for websites was provided by embedding advertisements (ads) on webpages and pop-ups that in some cases made websites annoying and others even unusable. This problem has led many administrators to start looking for other ways to make their websites more pleasant to be visited. This lead to the spread of ad blockers, Focused on a blacklist that would deny resources from specific URLs to be retrieved and shown to the user. This primitive way of blocking the pages was far from perfect, considering anyone can either re-route the traffic through a proxy or even just buy a new domain that was not in the blacklists yet and use it instead. After all, it was enough to deny most of the ads from being loaded and bringing the website revenues way down.

Although ads remain dominant, administrators started looking for other ways to monetize their pages and bring some of the lost revenue back. This is where *cryptocurrencies* came in to help. Cryptocurrencies exist for more than 10 years now. The first, Bitcoin, was introduced by Nakamoto circa 2008 [Nak08], but today there are many other [TS16, BMC+15b]. Nakamoto's vision was to create the first peer-to-peer currency eliminating third parties and financial institutions. The protocol would allow the coins to be *mined* [BMC+15a] by any user, but it would require a certain effort to be able to get the coins, in this case, CPU power. The previously mentioned protocol took advantage of a new technology called blockchain. Blockchain is a growing list of records, called blocks, which are linked using cryptography. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data. This ensures that the coins can not just be created out of thin air whenever someone wants. The idea behind the mining process is to calculate a hash using the exactly previous values of the chain and add a random value such that the resulting hash has to have a precise amount of zeros in the beginning. The amount of zeros required to find an acceptable hash would increase with time,

to make it harder and harder to mine new coins making the number converge to a finite amount, in the Bitcoin case 21 million. The Bitcoin the algorithm used for mining (Hashcash) was much more effective using GPUs with slightly modified software and using an ASIC, which is a chip designed for a special application, in this case for mining. Different mining algorithms were developed and applied to different coins with slight differences in terms of capabilities, usability, and profitability. Most of them based on the proof-of-work algorithm described early.

After a while, someone got the idea to mine coins on a web-browser instead of using thick clients. To make it possible the code needed to be implementable in JavaScript. So the pioneer Coinhive [coia] was born. Coinhive appeared as the first to make mining possible through the browser using JavaScript libraries. This was not possible until then since some low level controls over the hardware were needed and JavaScript usually does not allow it considering it runs on a sandboxed environment. It was only possible to run miners using the browser after the introduction of some new technologies, among them WebAssembly. WebAssembly stands as a binary instruction format for stack-based virtual machine. It was designed for a compilation of high-level languages.

The first cryptocurrency to be able to be mined using a web browser was Monero [SALY17]. Its mining process is based on a proof-of-work algorithm different from the one used by Bitcoin or Ethereum [Woo14], called CryptoNight [SJN+13]. It was now as effective to get coins using CPUs or GPUs making it the perfect candidate for the general public to be able to mine without the need to invest big sums of money.

As time flew by, the number of people that got to know cryptocurrencies and their capabilities increased. This made the intrinsic value of the coins to get higher and higher. This resulted in a huge bubble [Smi02], where the value of a single Bitcoin reached more than 20.000 dollars in value, crashing right after.

The process of mining takes a lot of time to generate a single coin and is expensive since the machine processing power consumed is high. Energy in most places is not cheap and high levels of processing power lead to a shorter life of the hardware, so the choice to do it must be weighed by each user individually since the investment may or not make it worth it depending also on the coin current value and evolution along time.

1.1 Motivation

Security-wise, everything exposed on the internet is vulnerable to some kind of issue and it is only a matter of time for it to be breached. Not all with the same severity but in most cases, the problems are easily spotted to someone with a little experience testing systems with multiple

technologies.

Cyber-criminals hack websites either for fun or profit. In the last case, the profits are often obtained by selling privileged information to entities who may benefit from it. Some of the companies in possession of this information and secrets are also those better protected.

Although, if an attacker just wants to do damage, there are some companies and public infrastructures like hospitals and water supplies that are great targets. A lot of them are not security educated, reuse weak passwords, and are afraid of updates (since they can break old routines/scripts). These are some of the facts that contributed to the wide-spread of ransomware in the last years. All these factors lead to a virus like Wannacry, a few years ago [HG02]. Wannacry leveraged a critical vulnerability on Windows SMB protocol to spread, lock all the content on a computer and ask for a ransom to unlock it. It spread initially by exploiting these services accessible publicly on the internet and in later iterations by phishing emails to further lock internal services still left unpatched. Criminals have been making millions paid in cryptocurrencies thanks to ransomware.

Nevertheless, the ideal target is the common user. There are billions of them instead of thousands/millions of companies. This brings us to a recent trend in the cyber-criminal world: *cryptojacking* [ELMC18, BBD19, PST19]. Cyber-criminals aw an opportunity here. If they could hack some webpages and embed there the mining scripts without getting detected they could capitalize on the millions of visits a day these pages get to further expand their profits.

Cyber-crime is trending, and easy to get into. With all the resources available today anyone can get skills in attacking webpages [LLR02]. With all this increasingly available knowledge, the companies need to keep up with their websites security, which currently is not done fast or well enough. Cyber-criminals are constantly getting new ways to monetize their efforts in attacking webpages. The old and most used way to do so used to be to steal information and sell it tho the highest bidder. But it is a dangerous game, with all the technology surrounding us there are dozens of ways to track illicit activities to someone trying to sell such information online. Also, someone selling the information never knows if the buyer is not someone belonging to a police force, making it a dangerous game to be involved in. Cryptomining is a new way of monetizing resources from thousands or potentially millions of users that visit compromised webpages each day. Crooks used only to have access to the compromised server and potentially an internal network and never the end-user machine. With cryptomining they managed to make all the users visiting webpages work for them and in a way compromise the end-users machines directly. Having a really good server at their disposition is good. But to have millions of users working for them seamlessly is even better. Also, it is not easy to pinpoint the origin of such an

attack. After the webpage owners find out about the ongoing attack the attackers have already made thousands in profit in virtual currency that can be moved around and converted into real money in online brokers. However, not only compromised webpages are making users mine for them. Some webpage owners are embedding mining scripts in their pages as a way to increase their profits. The issue is that the mining process should not happen without user consent for the reasons already mentioned. Needless to say, cryptojacking malware does not ask users for consent. There is an urging need to find a way to detect this scripts to the end-user to stop this resource takeover some webpage owners are currently doing, and a CPU active reading may be the response to the problem since one of the biggest tells is a huge CPU usage while not doing any CPU intensive task.

1.2 Topic Overview

Web-browser mining was made possible with an objective in mind, to get internet monetarization to the next level. An alternative revenue stream. But where is money there is always bad intent.

This mining process should not happen without the users' consent, since it makes them have extra charges in electricity and it also increases the hardware usability reducing significantly its life. Ideally, a choice between mining or having pop-up advertisements must be given, but it rarely happens. Most of the pages (even the ones that were not hijacked) never give this choice to users they simply start mining with no consent. When cybercriminals find a way to get the mining script into the websites, they make it as concealed as possible most of the time, passing months until it gets discovered which gives plenty of time to make a good sum of virtual coins. This happens due to the complexity that webpages have now. They require dozens of scripts to be loaded from multiple sources and complexity is the enemy of security and detection.

So the search for a method to detect these scripts is still an ongoing work. Multiple ones have been purposed. But none good enough.

The first cryptojacking detection mechanism to be developed and used was based on blacklists [MWJR18] which should contain all the domains that are known to supply these mining scripts preventing them to be loaded by the client. This rudimentary way of preventing scripts to be loaded, although accurate by blocking known domains, is also easily bypassed for example by having some kind of domain rotation going on. So theoretically someone could register domains indefinitely to go around this measure and make the scripts be loaded from the newly registered domain.

There is also some very interesting work done in early 2018 based on detecting the Websockets traffic used by these scripts [RSD+18]. They compared the signature of the code with some

pre-populated and saved all the JavaScript retrieved so it could be analyzed and grepped for different strings, URLs, and code fingerprints, and in this way track the cryptojacking mining campaigns. Although getting good results when compared with blocklist this method failed to detect obfuscated code. With no doubt being an amazing advance since it provided a way to find multiple websites that belong to the same campaigns it did not track new campaigns with different communication patterns and different fingerprints.

1.3 Objectives

We introduce a new *cryptojacking detection mechanism* based on the CPU usage of the visited web pages. Our detector monitors work in real-time, while the attack is happening (if there is an attack). The detector obtains a set of CPU metrics and feeds them into a machine learning algorithm. By combining a set of CPU monitoring features and using machine learning, we manage to obtain metrics like precision and recall close to 1.

A naive approach for detecting cryptojacking in webpages would be to measure the CPU time consumed and generate an alarm if it goes above a certain threshold. This would produce bad results for two reasons. First, some web applications have high CPU usage for long time frames, e.g., video streaming or conferencing applications. These applications would lead to false positives in such a simplistic mechanism. Second, the malware may throttle the CPU usage to keep it below a certain level, in order to stay below the threshold. This would lead to false negatives. These two challenges were solved by *not using CPU time as the single metric*, but instead using a combination of features based on a set of CPU metrics. Moreover, we use a machine learning classifier, that has the ability to pick a good combination of features to detect the phenomenon.

Our solution ideally would adapt and improve throughout time by supplying it with new sets of data over time. We are well aware of the problems this could bring to the algorithm, with the major of them being the possibility of an adversary violating the statistical models in which the solution rests, by continuously feeding it with tampered data.

To configure the detector we obtained a large collection of measurements of the CPU metrics with mining algorithms running, with different loads and in different conditions. Then we applied different machine learning classifiers to see if it is possible to have a model based on CPU measures with good performance in terms of metrics like precision, recall, etc. We excluded the possibility that the malware might use the GPU for mining. We acknowledge this possibility but we found no cases of malware that exploited browsers and used the GPU for mining, probably because the JavaScript APIs that allow it are recent and the malware is based on Coinhive

[ELMC18], the pioneer of web mining (as a substitute for ads).

But this solution by itself will not be enough. There is a need to combine it with 1 or 2 extra detection methods that may not be good by themselves, but will provide an extra validation and have a higher level of confidence in the results.

Some extra problems may represent a challenge, for example, the fact that some of the mining scripts do not mine in all the browsers that load the page, others are not well implemented and never start mining while still being present. However, this does not mean that the websites do not have mining scripts embedded on them, it just means they are not mining at the moment. Our solution will not cover these specific problem all by itself, it will only detect when the client is actively mining cryptocurrencies.

Our research may have a more broad appliance than the browser miners. We base our work on CPU metrics and patterns which is a traversal and a recurrent behavior in miners that run using the CPU not only on the browser.

1.4 Thesis Outline

The following document is divided into 5 Chapters that were developed while the different stages of the research were being performed. Chapter 2 is where all the research performed in the existing related work that could be complementary to this document. This step was done before we tried to use each technology individually.

Chapter 3 discuss the Design, some technology choices , and problems that were found while implementing and using the tools that were later applied in this research.

Next, in chapter 4, we show the results we got from the different experiences we performed and conclusions that lead us to each subsequent experience.

Chapter 5 we evaluate the results we got from our experiences, and we give our final considerations of the performed work and some ideas for future researches and tools. Next, the References, which has the bibliographic registry of the quoted work along with this document, and finally the Appendix where some scripts used in this research are present.

Chapter 2

Related Work

The solution required a lot of research in different areas. There are five components that are the core of our work, so we decided to divide the chapter into the following five sections. We start with a background into the cryptocurrencies, next to some research on the actual Cryptojacking theme, what it is and what makes it a danger, also some machine learning work previously developed and how could it be applied to find patterns on the values we are going to supply to it. Next what is an Intrusion Detection System and what kind of them exist that could better fit our model, also more in-dept research on the CPU metrics and some APIs research that could possibly be of use to the present document , and finally a discussion of some of the already developed and researched solutions to flag cryptominers.

2.1 Cyptocurrencies

Cryptocurrencies appeared to try to eliminate the need for third-party institutions on money payments and transference online. Although this objective was not accomplished for now, since there is still the need to use services provided by centralizes companies.

As described by Satoshi Nakamoto [Nak08], the existent system is not a good fit for today's online conditions. It is not easy to reverse transactions, there is no way to keep anonymity in the current system, the costs of mediation paid to the bank institutions is too high for the frequency of the transactions, leaving small transference is unfeasible since the mediation payment would be massive when compared with the transaction value. A way to try to resolve all the problems above a cryptocurrency was born called Bitcoin. It is based on cryptography with algorithms that are yet waiting to be proven to be insecure. However, there was a need to have a decentralized control of the transactions so the users were requested to take part in this endeavor. The way to make sure there was a good number of systems always connected

to the network allowing for the transactions to be processed was by giving an incentive to the people that allowed their machine to be part of the swarm. The incentive was paid by giving Bitcoins to users, or fractions of Bitcoins, and they were only paid the to ones that found the hash matching some specific rules. After this hash is found the result is propagated to the rest of the network and would be from now on part of the blockchain.

At the beginning, Bitcoin did not have much traction and it took a few years to have it accepted by a large number of users. But after this phase, people saw the opportunities of the new system and also found a bunch of flaws that Bitcoin had. The flaws could be solved with some differences, so hundreds of new cryptocurrencies [coi15] started appearing with the emphasis on different problems, from faster transactions to better anonymity. This flourishing market has also lead to a lot of research in different aspects of cryptocurrencies, from different networks [Kin13, BMC+15b, TS16, GHM+17, ABB+18] to attacks [ES14, AZV17], consensus algorithms [CGLR17, Cor19], and many other topics.

But none good and complete enough to solve all the problems, replacing the current financial system which is flawed and prone to corruption. Extensive researches have been done in the cryptocurrency field.

2.2 Cryptojacking and its Detection

Cryptojacking, as previously mentioned, is a good idea – an alternative to ads – badly used for stealing computing resources. This kind of attack has been growing in the last year. In 2017 alone it has grown over 8500%, according to 2018 Symantec security annual report [Sym02] which is an amazing growth, allowing it to compete directly with the most widely spread attack nowadays, which is the ransomware.

A good way to detect it is still under development. Currently, some research has been done to understand what is the best way to do it but none good enough has been found.

Rauchberger et al.[RSD+18] mention that the best ways to identify a browser mining cryptocurrency, is by tracking the usage of the following mining specific technologies:

- WebAssembly;
- Web workers (a non-trivial amount of them);
- WebSockets (probably not the best one since it is widely used by other web apps).

The work developed by Rauchberger et al. is probably one of the best to take as a foundation for this research. They touch some of the most crucial problems and features, that need to be looked into to try to flag as accurately as possible the mining browsers.

Most of the time when a mining script starts, it does not let the CPU go to sleep for more than two seconds between hash calculations, even with features like throttle introduced by Coinhive. This option lets the attacker set up a value that will be applied to slow down the mining activity so it passes a bit more unnoticed by the user or just slows down the hash calculation rate allowing for a more pleasant visit to a website that he explicitly allowed to mine, instead of watching advertisements. Even with the throttle value as high as possible, it is clear the high CPU usage, making this probably the best option to track the mining scripts. It is the only thing that can not be hidden or easily tampered with.

The biggest challenge faced by our research is the uncertainty of the values yield by the CPU for cryptomining detection. The outcome is very unpredictable. Many web pages have high CPU usages even without actively mining any cryptocoin. A lot of the inexperienced developers do not know how to make efficient websites and as a result, these pages will use a lot more CPU. The decisive tiebreaker lies in the prolonged high computation values, even after everything on the web page as already loaded. If it happens we are most likely in the presence of an active cryptojaking miner.

MiningHunter [RSD+18] is a framework developed to track mining scripts. It had an interesting concept in mind, where for every loaded website the metadata, all executed JavaScript, and raw Websockets traffic is recorded. The stored data is analyzed and compared with each other, looking for common strings and functions with the same signature are checked for patterns and keys that were repeated. After the previous step, the signatures were matched to try to figure out which ones belong to the same mining campaigns. Although it had good results when compared to the use of blacklists, it collects much data, which is problematic and avoidable.

Code analysis can be done fairly easily but it has significant drawbacks since as these authors found out all it is needed to make the algorithm not to detect much is to obfuscate the code delivered by the website resulting in a less effective analysis. There is although ways to improve the detection but it will not have as close as the same success rate as the one observed before the obfuscation was introduced.

Using a different approach is the solution developed by Rodriguez and Posegga [RP18]. They aimed to label system resources particular API calls and analyze when they were used. They had an amazing success rate of 99.99% detection which was an impressive result. Their system was trained using different classifiers and methods.

All the mentioned researches were done to tackle the cryptojacking campaigns that run on users browsers, but we propose to take it one step further, since the solution to be developed will be CPU based, it might be able to track other CPU based mining cryptocoin software that

does not run on the browser but it will only be possible if their patterns are the same. This has a more limited application since most of the mining algorithms are developed to have a heavy load on the GPU instead of the CPU.

Saad et al. [SKM18] have also put together a solution to analyze browser-based cryptojaking scripts. In their analysis, they also did some interesting findings in the way WebSockets are used by these scripts. They noticed that during cryptojacking script execution, the code would establish a WebSocket connection with a remote server and perform a bidirectional data transfer. The WebSocket communications can be monitored using traffic analyzers such as Wireshark. However, a major issue when using traffic analyzers is that browsers encrypt the web traffic during WebSocket communication making it not that easy to analyze.

Some patterns can be seen by carefully watching all the transferred Websocket packets (which use a single TCP connection). Some of the traded messages have a standard length that can be matched between authentication processes. The analysis done by these researchers found that the authentication message is 112 bytes long. The site key parameter which is sent to the server in the authentication process is used by the server to identify the actual user who owns the key and adds balance of hashes to the user's account. The server proceeds to authenticate the user and to respond. This message has a length of 50 bytes, it includes a token and the total number of hashes received so far from the client's machine. In the authentication message, the total number of hashes is 0, since the client has not sent any hashes yet. Then, the server sends the job message to the client. The job message has a length of 234 Bytes with a job id, blob, and target. The target is a function of the current difficulty in the cryptocurrency to be mined. The client then computes hashes on the nonce and sends a submit message back to the server. The submitted message has a payload length of 156 Bytes. The hash accept message is 48 Bytes long. So it is possible to write a detection system based on this length patterns but this will not be accurate enough since it is easy to mascaraed the traffic length and it differs for different miners.

One promising solution named Outguard [KMM+19] uses 12 different features to flag pages as being potentially running cryptojacking. These features ranged from easily detectable patterns to technologies used in multiple of these pages. Some of the methods were also used previously, like the presence of Web Workers and the use of WebAssembly, but also using events like "PostMessage Event Load" and "MessageLoop Event Load". They also added the CPU use as a detection feature, although the only thing that they checked was if the CPU load was over 50%, which is problematic as they acknowledge. Cryptojacking in most of its forms allows for the attacker to bend the CPU used percentage at the attacker's will. A page can keep its mining

activities at a low CPU consumption rate to avoid detection, but also keeping its profitability low. Also, the researchers found out a lot of the pages found in the wild have a high CPU demand due to multiple scripts loaded and multiple network operations which could add a lot of false positives if used as a standalone feature. Once again the researchers used a cross-check methodology, so verify if the CPU usage was high was a good way to have an extra verification step to make sure there was an intensive miner at work. This paper had great results, In the sense that it found twice the miners the blacklists were flagging. This was a more complete solution than some of the previously evaluated work.

Coinpolice is a more recent research that also uses multiple features [PIB20]. The authors used the following methods to cross-check the presence of a miner: 1) a baseline classifier that only uses a hard-coded 30% threshold over CPU usage, 2),3) two classifiers based on HPC counters (respectively using a convolutional and recurrent neural networks), 4),5) two classifiers based on JS/WASM function execution time series, and 6),7) two classifier based on the JS/WASM features and throttling-detection features. Expanding their CPU threshold to 30% allowed for them to have a broader and more complete mining detection but it also introducing the possibility of a higher false positive rate if used on its own which wasn't the case. They also had a result of flagged pages around the 6k that were running cryptojacking, from the 1M top Alexa pages having a what they claim to be a 97.8% on True Positive Rate and a 0.74% False Positive Rate. Besides that, they only profiled the page for 5 seconds (not counting the load time established at 10 seconds) which is a good time estimate for a user visited page.

A code analysis classifier that performed a dynamic analysis of the opcodes [COSB18] was developed using Weka to execute the classification algorithm. Opcodes are portions of a machine language instruction that specifies the operation to be performed. They took advantage of the 10-fold-cross-validation training method which seems to be the standard for this kind of machine learning training runs. The previous method was used to train their chosen algorithm (Random Forest) that would find the best way, given the supplied training data to tie them together and understand what was the constant across all the examples that allowed it to flag a malicious script. They concluded having as a basis the output of the Weka tool that dynamic opcode tracing is extremely effective at detecting cryptojacking. Claiming a 99.9% accuracy detecting cryptojacking. Also, their model can distinguish between cryptomining sites, weaponized benign sites, de-weaponized cryptomining sites, and real-world benign sites. They did get a good result using a single detection method, not needing an extra validation step to verify the prevalence of malicious scripts.

Munoz et al. present a solution to detect cryptojacking by inspecting network traffic [MSVBR19]. The detection is based on monitoring NetFlow/IPFIX flows that summarize traffic, so this approach can be placed outside of machines being monitored.

SEISMIC detects cryptojacking based on semantic signatures of behavior [WFX+18]. Moreover, it uses the fact that its detection mechanism provides low false positives to dynamically block scripts flagged as executing cryptojacking.

As Carreiro finds in his work [Car19], mining activities are definitively possible to flag just by observing the CPU and GPU activities depending on the kind of miner running on the system, since they have a recognizable pattern when running. Temperature is also a good way to detect the presence of a miner, since the CPU and GPU intense activity is linked to a higher temperature on these components. Although these are valid ways to flag mining operations, they can possibly be fooled or miss understood by other computationally intensive tasks.

2.3 Machine Learning

Machine learning is a reasonably recent technology. Although some of the techniques used by machine learning algorithms have been used for several decades its study and recognition as a separate study field is recent, so much is still under heavy research.

As Brownlee [Bro16] states we can divide machine learning algorithms into two big groups: the supervised and the unsupervised algorithms. The supervised machine learning is where you have input variables (x) and an output variable (Y) and you use an algorithm to learn the mapping function from the input to the output.

$$Y = f(X)$$

The goal is to approximate the mapping function in such a way that when you have new input data (x) that you can predict the output variables (Y) for that data. Unsupervised learning is where you only have input data (X) and no corresponding output variables. These are called unsupervised learning because unlike supervised learning above there is no training data. Algorithms are left to their own devises to discover and present the interesting structure in the data.

Machine learning classifiers are a form of supervised learning, where an input is given and a specific output is expected. Pereira et al. [PMB09] described the classifiers usability as the name indicates is way to classify the input variables in a way to predict accurately the outcome.

A classifier has a number of parameters that have to be learned from training data. The learned classifier is essentially a model of the relationship between the features and the class label in the training set. After the training, the classifier has to be tested against a different data set to make sure it captured the relationship between the input and output.

Some books have already been written on the subject, in giving crucial understanding on how to use the existing libraries of classifiers and adapt them to our individual needs.

Most of the more widely known libraries are written in Python, like Numpy [lp05] and Scipy [lp01], the first one being a data manipulation tool and the second one has modules more centered in machine learning algorithms. Python is a versatile and powerful language, that although not super fast it is very intuitive and easy to develop on.

There is also a very interesting tool written in Java called Weka [wek08]. Java is a heavy RAM using language and when compared with more recent languages is not as good as they develop, but this tool has a great advantage that puts it in a better position than Python which is the ease to swap between evaluation algorithms. It will greatly increase the speed at which this research is done and since the aim of the detection algorithm is not to use as few resources as possible and taking into account the ingenuity of the researcher to machine learning, anything that facilitates the algorithm choice and reduces the probability of error seems like a logical choice.

Richert [Ric13] defines some steps which the development of a machine learning algorithm should go through, shortly described next:

1. Collect data — The first step to go through is the data collection which is going to be used in all the following steps, so the data must be reliable;

2. Preprocessing and data cleaning — There is a preprocessing and data cleaning step that aims to reduce the existing error of the samples, and the values that have a big deviation from the majority will be the ones to be removed. The number of removed items can never be big when compared with the totality of the samples, otherwise, the accuracy may be compromised;

3. Chose model and learning algorithm — This is where the heavy lifting will be done by the previously mentioned Java tool. Here the model and learning algorithm is chosen, also known as a classifier. It is chosen mostly by trial and error, and the idea is to get the line that better fits the collected data while having the smallest error possible. More than one line may be needed, to have an accurate model and in this case it will be better to divide the data into multiple groups. If no model is found to fit well on the gathered data we will

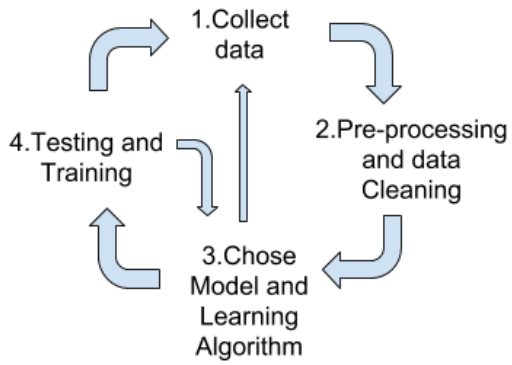

Figure 2.1: Machine learning algorithm choice cycle

be required to go back to step 1 and collect a new data set;

4. Testing and Training — Finally all the previous work is tested and from here on out the algorithm will go into a continuous learning state. At this point, there is a comprehensive review of the results and an algorithm evaluation to understand if step 3 needs to be repeated.

The cycle described above is represented in Figure 2.1.

The classifier choice is not that easy or straight forward. It may be required to take into account multiple attributes (ex. CPU usage, CPU temperature, CPU cycles ...) to better find a suited classifier. There are multiple factors to take into account like which one of the classifier attributes will have the most discriminatory power.

The attribute selected in this research will always take into account continuous monitoring over-time of the CPU metric we chose, so we can not get the value of the metric once per web page, it will need to be collected in different moments and with regularity. However, it will most likely not be enough if we use one metric by itself so there may be the need to collect a variety of them.

A good way to measure the performance of the classifier is by having a binary result attributed to each analyzed page, where 0 means incorrect/not present and 1 valid/ cryptominer present. The average of these two after testing would indicate how accurate they were when compared with the blacklist success rate. By itself this will not mean much, so a group of other values may need to be calculated in order to get a better perspective of the classifier performance.

An interesting way to train the algorithm is also described in the previously mentioned book and is called the “leave one out”. This happens when the data is divided into multiple groups chosen randomly at each iteration, with the same length (usually 5 to 10) and one of the samples is left out for testing purposes, while the rest of the samples are used to train the algorithm.

Which, is the way our classifier will be trained.

2.4 Intrusion Detection

Wagner and Soto [WS02] mention that there are broadly speaking two kinds of intrusion detection systems: network intrusion detection systems and host-based detection systems. The difference is mostly based on the fact that the host-based solution is capable of monitoring all or parts of the dynamic behavior and the state of a computer system, based on how it is configured. One can think of host-based detection systems as an agent that monitors whether anything or anyone has circumvented the systems' security policy. Network IDSs (NIDSs) look at network data as source for their analysis. NIDSs work in real-time to find out the malicious traffic making it faster, when compared to a host-based detection system that works by analyzing suspicious activity on the host itself.

Our interest lies with host-based intrusion detection systems (HIDS) since they are the most adequate for what we aim with this research. Host-based intrusion detection system can be further divided into two categories: signature-based schemes and anomaly-based. Usually, signature-based detection systems are easily bypassed with minimum variations on it.

Wagner and Soto say that signature-based detection is easy to bypass when compared to anomaly-based ones. However web applications are all different among them and there is no way to create a pattern that would match all of them watching their CPU usage, and get an anomaly out of it.

Therefore we decided on a signature-based intrusion detection system, with signatures created automatically using machine learning. We could create a signature not of all applications (because it would probably be impossible) but of the mining algorithms and flag a website as malicious once the pattern is detected. It is possible with this solution to analyze what is happening with the web application without requiring anything from the user, allowing him to be able to keep using his browser without noticing any difference.

The intrusion detection mechanisms are defined by models with all their capabilities and definitions. The intrusion detection mechanism will determine when there is a security-related issue, what was the trigger, and what should be done after the trigger was issued.

Intrusion detection models as mentioned by Denning et al. [Den87], are based on the premise of the intrusion or exploitation of a system vulnerability that involves the unwanted and abnormal use of that system. Therefore, intrusion attempts can be detected by abnormal system patterns. The deal breaker is the ability to find out what metrics and patterns to take into account, to catch these security events as fast as possible, and to seize them with the same

celerity.

The IDES (intrusion-detection expert system), proposed by Denning et al. goes into big details on what people defining the detection systems need to have into account when designing the system, making it as reliable as possible and in what way this must proceed so the events are handled correctly.

One of the definitions to take into account is the profile definition. The profiles define what is an expected behavior for the security event. So it works like a signature that can have some deviation but not so much that turns this signature too permissive. In other words, it must have a statistical metric and model that supports it. Here the metric definition comes into play, it is a variable representing a quantitative measure accumulated over a time frame. The time frame is one of the three metrics that need to be used, and it will be fixed at a specific value, that may vary from some seconds to several minutes. The smaller the time frame we get the faster the security issue will be caught, but it comes at a price. Smaller time frames means that it is a lot harder to find patterns even for automated systems. Longer time periods mean that the threat that is attacking the system will be taking advantage of it for a longer time frame.

Chari et al. [CC03], mention that one of the factors that provoked the shift from network-based detection systems to host-based is the fact that now communications are being encrypted. A network IDS would not be able to know what is being transferred on the network if it is encrypted, making the shift to the host a required measure. Their host-based solution was developed at the kernel level and is based on a set of rules/policies that are applied depending on the host behavior. The system policies are meant for the control access to system resources. It was a limited solution. It had a set of policies and it was not possible to establish policies for all the processes. We aim to surpass this limitation by having the machine learning algorithm to decide what is a bad behavior and allowing it to have a deviation from the defined pattern, to better suit multiple miners.

In our case, we do not know how long will be needed to find a pattern that suits the detection of cryptojacking but taking into account that most websites are visited for short time frames (except for pages that supply media content like videos or games) probably any pattern recognizable under 1 minute can be called a good pattern since the damage of the attack is reduced heavily.

The statistical model will be defined by a machine learning algorithm that will be trained using multiple pattern signatures of known web pages actively mining, but the model that better detects the scripts will be chosen by trial and error. The model will have a standard deviation taking into account known default values (learned by the algorithm) and this deviation will

be calculated to suit our needs. It will be based on a multi-variable model, which consists of considering multiple variables on its analysis.

The profile structure mentioned here is very important since it will be on the pattern match core. Each time the pattern recorded by analyzing the CPU metrics matches as close as possible to a known cyptominer software, it will trigger an alert to the user and record the needed metric to improve itself. This is also known as a knowledge-based intrusion detection [DDW00] since we are trying to track known system vulnerabilities and to match them to the existent patterns. Any other behavior is going to be acknowledged as acceptable. For its completeness usually, the algorithms are required to update the attacks' info regularly. In our case, we plan to have the algorithm feeding itself new measures to keep learning and improving without human aid.

The usual advantages of this kind of system are the low false positive rate but in our case, it is precisely the opposite, the number of false negatives are expected to be low and the false positives to be higher. Most of the JavaScript scripts can require high CPU usages and set an early alarm on our algorithm, however, if we look at longer time frames this misleading alarm can be discarded since after loading the web pages do not require a relevant CPU usage.

The disadvantages are usually tied with the difficulty of gathering new data, we do not foresee a great deal in this field since the learning process will be retroactive. Although bypassable if a real out of the ordinary way of mining is found that does not require high and continuous use of the target CPU like the proof-of-stake mining algorithms. Also, these detection mechanisms are usually closely tied to a certain operating system, version platform etc. The choices being taken will try to avoid these situations and make the solution as global as possible. A machine Learning Intrusion System is not a breakthrough or a something new since a lot has been done [BG16] and with a lot of new additions were brought since Internet traffic classification [NA08]. Although our analysis has some internet basis is far from traffic analysis, but some research can be applied to our model.

2.5 CPU Monitoring

CPU monitoring is not new and multiple CPU metrics can be provided to the user, either native by the hardware firmware, by the operating system, or by APIs written for this purpose. Some times even taking into account more sensors than those present by default on the CPU (like those on the motherboard). These sensors allow the user to get different information and some times better reads than those already given by the CPU. The ideal scenario would be to get metrics provided by default on most operating systems or hardware, and have with it universal coverage. By taking this into account, the final solutions that would depend on the provided metrics would

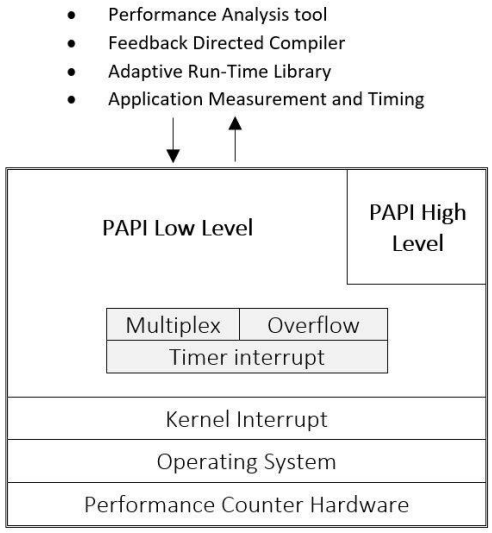

Figure 2.2: PAPI architecture [TJYD10].

work on the highest number of systems possible without the need for modifications.

As mentioned earlier one of the most noticeable changes in a system when it starts running these mining algorithms based on the browser, is the CPU high usage by the browser process. If it happens via a website it will load malicious JavaScript to the client, otherwise, if the user is not infected via browser it would be an independent process and the malicious code will be in a different language.

To gather the required metrics we found after a while digging an API, known as PAPI. The referred API provides a lot of measurements of the CPU for example:

1. CPU temperature;
2. CPU cache;
3. CPU usage, among others.

The API was written several years ago now and has been updated by multiple times [TJYD10]. In 2009 there were several major updates on PAPI-C, which allowed it to get more accurate measures on multi-core processors with different architectures. This module allows for a more complete set of metrics to be used with a software interface, and it works in a great variety of hardware. Probably one of the best ways to get values from the sensors.

Going away from the previously mentioned API, we got the PCM which is better known and had in the last years one of the most relevant advances, the PMU (Performance Monitoring Units). Which was introduced by Intel in its latest processors. It implements a basic set of routines with a high-level interface that is callable from user C++ application and provides various CPU performance metrics in real-time.

The PCM (Performance Counter Monitor) has all this technology incorporated in Intel CPUs, starting from version 1.5. PCM contains a Windows service, based on Microsoft .Net 2.0 or better, that will create performance counters that can be shown in the "perfmon" program that is delivered with the Microsoft Windows OS. It is interesting since Windows is the most widely used operating system in the world and a wide support is right what we are looking for. Besides that the PCM module has also a Linux and Mac OS X support. Although not present natively the module exists and works. Thanks to the abstraction layer that the library provides, it has become very easy to monitor the processor metrics inside the applications.

In its later versions 2.0 it also shows the energy usage info by socket, that may be used as one of the indicators to train the machine learning algorithms. These algorithms are also known to have high energy usage thanks to their high processing output required.

The only disadvantage is the fact that the feature (PMU) is restricted to the latest Intel CPUs, the Intel Xeon E5 series processors. These processors are not widely used at the moment but with a little, more time will probably be since Intel is the biggest player in the market and their processors are the ones used by numerous users.

After all considered, the PCM API was though to be the chosen technology but was later discarded for reasons described in the next chapter. It had a wide support across multiple operating systems and its Github repository [opc15] shows continuous development. The different metrics provided are also something useful, that can be used to understand which one is the most adequate to detect the scripts running on client side. And even though the PMU supplied counters may not be used all the other counters may be of great use.

One of the latest works done in the CPU monitoring area was done by Phung et al. [PLZ18]. In the research the main goal was to make use of the Running Average Power Limit (RAPL) feature that is commonly found in today's Intel CPUs to track the energy consumption and applications efficiency. One of the main issues that cryptocurrencies have is the power consumption. They use a proof of concept algorithm that requires high processing power from the CPU. In their research they were able to create a very accurate power consumption model that had only a 1.63% error. They were able to get this values using only a software-based virtual power meter. It looks like a good fit since we would probably be able to track cryptocurrency mining. However, it is not an optimal solution. Their research got an awesome result that based all the solution in a simple equation as shown next:

$$P_{system} = aE_{CPU} + b \tag{2.1}$$

The values a and b are variable from CPU to CPU so the solution would not be general

enough since these values would need to be calculated to every case. Also, it requires a very recent technology to be present in the CPUs which is only present in the latest ones produced by Intel leaving a lot of users out of a solution. The amount of users covered by the solution although being important it is not a strict rule.

This chapter presents the background research work done for each of the components that were part of the solution developed during this project, before going into the actual design of our solution and the selection of final technologies we would use in later phases.

Chapter 3

The Intrusion Detection Mechanism

The chapter ahead is divided into three logic sections which were taken to develop the Cyptojacking detector. First, Section 3.1, where we describe some design choices taken before developing this work. Next, Section 3.2, here there is a deeper development of what kind of problems were found while trying to choose the technologies and why some got chosen instead of others, that were described in earlier chapters. There were quite a few roadblocks that needed a solution and this was the way we managed to solve them. Section 3.3 describes our approach to categorize each result gotten from our solution and understand what each result means and how should it be interpreted.

3.1 Detector Overview

Our intrusion detection tool was designed to run inside virtual machines. For now, it is set up to check if a webpage executes cryptojacking scripts. For that purpose, the detector is started by executing a Python script composed of four modules, represented in Figure 3.1. This architecture is modular and it aims to work well once exported and used in different machines.

The script will first run a *Webpage crawler* that will visit the page and download all its scripts. While the page visit is happening our tool will be running a *CPU monitoring* module that gathers all the selected metrics that are later supplied to the *machine learning classifier*. Before feeding the data into the classifier, it is put in the format the classifier consumes by the *Arff parser*, that returns a file in the Attribute-Relation File Format (Arff) format. The job of the Classifier is to assign to a class – cryptojacking / no cryptojacking – the input obtained from the CPU monitoring component with the help of a classification algorithm. We tested many of such algorithms, so we used the implementations available in the Weka tool [HFH+09]. Weka computes with the chosen algorithm a result with the presence or not of any kind of

cryptojacking miner that would later be supplied to the user.

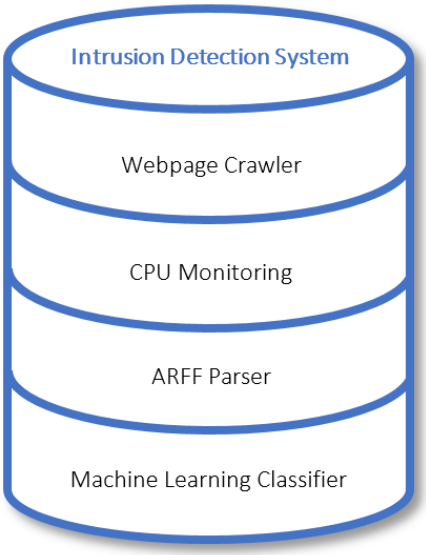

Figure 3.1: Intrusion detector architecture

To reach the previously mentioned solution there were multiple steps to take into account, which are described in the next Section 3.2 and further developed in Section 3.3.

3.2 Configuration

3.2.1 Webpage Crawler

The logical division we did was the following. First, there was the need to crawl the web pages. For each of the use cases (for training and testing), different lists were supplied in both cases. In the training set, the results of each dataset were already known, but in the second list (testing set) the results were not known. The testing set was a group of web pages accessible online. The crawling was performed by a simple script, which went through each page individually and loaded all the resources of the initial page, just like a web browser. The engine used to perform this action was Chromium-based.

3.2.2 CPU Monitoring

Metrics are gathered by retrieving the CPU values for each webpage when it is loaded. For training purposes, each page is loaded then left running for a specific time before it is halted. While the page is being loaded the second stage is happening. The CPU behavior is being recorded and all the chosen metrics are being stored to individual files, so they can be further used and compiled in later stages to files that are later processed by the machine learning algorithm (Arff files).

3.2.3 Arff Parser

The next step of the process was to develop a script that transforms the data stored on the individual files to treat it choosing the interesting values, put them in a format that Weka understands and interprets as valid.

Weka needs to receive the metrics from a specific file format, known as an Arff file. Arff files are divided into a header part that describes how the data is subdivided into the next section. The file header is as big as the number of fields we are sending for evaluation. The name of the fields does not matter, just needs them to be different from each other. Next, the data, where there is a label as a first positional element. After it the data we are trying to evaluate, the number of fields will vary depending on the number of seconds we are trying to evaluate and the number of CPU metrics we want to evaluate. The last positional value of each line is the classifier that in our case will be a binary value, 1 which marks the presence of a miner and 0 otherwise. It could be a list of values and it could be composed by strings instead of integers, but in our case, there was no advantage to this approach.

3.2.4 Machine Learning Classifier

This is the last and one of the most important stages. We use Weka to evaluate which classifier better fits our training values so that later we could apply the chosen classifier to classify the testing values. The classifier values (1 or 0) will only be attributed to training values: 1 will mean a miner is running and 0 will mean no miner is running. The algorithm will need to know the result of what it is evaluating to create a pattern and further classify unknown values. The last position of the testing values has the character "?" attributed, since the result is unknown. The schema of the previously described steps is present in Figure 3.2.

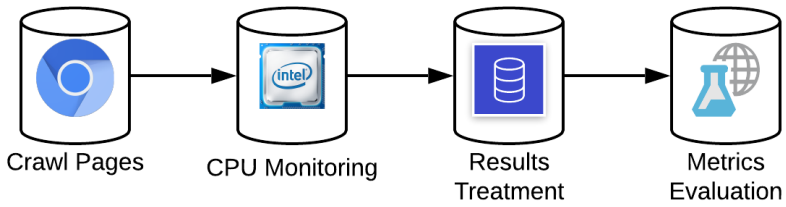

Figure 3.2: Steps to take between the data gathering step and its evaluation.

To choose the classifier that better fits the data, a number as big as possible of classifiers available within Weka, that can be applied to our data, will be used. Their precision rates will be taken into account when choosing the best option.

3.3 Implementation

There were quite a few architecture changes since some of the researched topics and technologies had problems not anticipated in the research chapter. Starting by the way metrics are acquired. The `PCM API` that expanded the already deployed process monitoring tool in Windows has proven not to be very reliable. On Windows, there were always compilation problems even with all the requirements met. This could be due to multiple reasons starting from the machines used or the support of the Windows version used. Most of the features advertised by the API did not work as intended on our second option to make this option doable (by running it in MAC OS). The `PCM` allowed us to get accurate metrics only a fraction of the times, others it just stopped working or thrown inaccurate results. The machine running the API ended up crashing multiple times probably since it was working at this low level, directly with the CPU calls. These observations revealed that it was probably not such a good idea to be calling routines at this level thousands of times throughout the experiment. We want something to be usable by users, easily deployed, and trustworthy enough not to crash.

We found there was a decent number of tools developed for Linux that got a lot of CPU measures that did not even get considered. Some evaluated tools are the following: `IOSTAT`, `MPSTAT` and `VMSTAT`. Each of these tools had their advantages, `VMSTAT` for example was customizable but had an output problem, it was not reliable or readable if not printed to stdout. The way the output was saved to a file was different in each write and it was not possible to write a parser on top of it with the output schema changing every time. There was no good way to parse it. It was a widely requested feature in multiple Github issues and pages throughout the internet. There was no solution for it yet. `IOSTAT` although outputting CPU metrics, they were a lot less than other tools we have analyzed until now. `IOSTAT` also had a big disadvantage when compared with the one we ended up choosing. It did not have the possibility of gathering the metrics of each core separately. We ended up choosing `mpstat` to monitor the CPU. It monitors the activity of each processor core independently. It also calculates the averages among all the processors which can be really helpful to some of our observations. The chosen tool outputs a good amount of CPU metrics, it is customizable in terms of metrics by core and time interval at which the reading should be done. Also had a fantastic output format, and best of all it worked out of the box without crashing in Linux and Mac OS. Windows was unsupported, and we dropped it as a supported operating system for the solution since at this point there was not a good option to cover all the operating systems. `MPSTAT` allowed for the following metrics to be extracted (%usr, %nice, %sys, %iowait, %irq, %soft, %steal, %guest, %gnice, %idle) from the different CPU cores independently, and an average could be drawn between all of them. Next

the explanation of what each metric means:

- **%usr**– the percentage of CPU utilization that occurred while executing at the user level (application).
- **%nice** – the percentage of CPU utilization that occurred while executing at the user level with nice priority.
- **%sys** – the percentage of CPU utilization that occurred while executing at the system level (kernel). Note that this does not include time spent servicing hardware and software interrupts.
- **%iowait** – the percentage of time that the CPU or CPUs were idle during which the system had an outstanding disk I/O request.
- **%irq** – the percentage of time spent by the CPU or CPUs to service hardware interrupts.
- **%soft** – the percentage of time spent by the CPU or CPUs to service software interrupts.
- **%steal** – the percentage of time spent in involuntary wait by the virtual CPU or CPUs while the hypervisor was servicing another virtual processor.
- **%guest** – the percentage of time spent by the CPU or CPUs to run a virtual processor.
- **%idle** – the percentage of time that the CPU or CPUs were idle and the system did not have an outstanding disk I/O request.

We could probably get the same metrics developing a tool ourselves using "ps" since it is a very complete command and most of the CPU metric tools use it in the background. But `MPSTAT` was a well built tool and already presents all the capabilities we were looking for, saving us some time to develop our solution.

While the initial tools were being chosen and some of them developed, there was a major setback. The cryptominer in which most of the research was sited upon was Coinhive [coia], which ended up closing operations. It was the most widely spread way to mine coins using the browser. The shutdown was announced giving the reason the value of Monero (the coin mined by Coinhive miner) value as decreased as much as 85% and the coin was getting harder and harder to mine due to the latest forks at the time which lead the company not to be profitable to continue its activity. Other companies existed that provided the same kind of service. Some of them even getting it more attacker/user-friendly providing some script examples, that could by default warn the user there was a cryptominer starting on the webpage. It also had the capability

of adding ads on top of the miner to extra profitability, we can also change the amount of CPU usage the miner was allowed to use among others.

After a careful evaluation a replacement was found, one that allowed for the initial requirements to be met as closely as possible. The best choice after Coinhive's shutdown was CoinImp [coib]. CoinImp had a very complete interface and the ease of use became the number one miner in no time. Also, the miner allowed for the amount of CPU power used while mining to be set to any percentage, which would really help in understanding if our solution was reliable in lower CPU mining loads. It was a required feature to be able to throttle the CPU to different percentages and check its behavior, and with this make sure we could also detect a miner at low but constant CPU uses. CoinImp also provided a number of monitoring solution on their web interface that allowed understanding which of the attacker is controlled web pages were generating more mining power as well as coins.

Also mid way through the implementation phase there was another small change off the design. CoinImp stopped supporting Monero. The virtual coin Monero suffered some new changes that made it even harder and impracticable to mine, there was no way to generate profit from it leading CoinImp and others to halt his mining process. CoinImp provided an alternative coin called MintMe [Min]. The coin had a behavior similar to the one yield by Monero while mining (high CPU usage), actually the same behavior as most of the coins that took advantage of clients machines throw the browser using Javascript. Also, Webchain kept the ability to throttle the usage of the client CPU with these and keeping in mind our requirements there was no visible change between both while mining.

After the initial observations and due to some bad results on the 15 second measures the time frame used ended up being extended to 60 seconds, since with these there was a bigger window allowing more patterns to be discovered (if there were any). By evaluating the results discussed in the next chapter there was the need to perform extra modifications and steps on the metrics gathering schema to ensure as much ground as possible was covered. We also chose to add to the training set of values some crawling results of pages publicly available online, which were ruled out as dangerous pages. These pages were previously checked for high CPU usage and were found to be miner free. We were also careful to choose websites that were having a higher CPU usage than most webpages and for longer time frames. With these web pages we were trying to provide some real world examples that could be miss interpreted as running cryptojacking, allowing for a better trained model.

The web pages we chose to use for our new training model were the following:

- http://tmall.com

- http://sohu.com
- http://taobao.com
- http://jd.com
- http://alipay.com

We can see in Figure 3.3, the behavior of a CPU while the page sohu.com is downloaded and running. The mentioned page was chosen for this purpose since it did not have any trace of a cryptominer running. The page takes close to 15 seconds to load and the CPU consumption is not steady. By doing some research on the topic the average page takes between 10 and 15 seconds to fully load. These timings can be influenced by the distance between the website hosting site and where in the world we are requesting the resources from. The kind of scripts being executed can also increase the time a page takes to fully load. CPU time shows multiple peaks and some reduction on average, except for peaks that continue to appear when a routine or script is executed. We recorded the CPU behavior for 60 seconds, checking the metrics once each second.

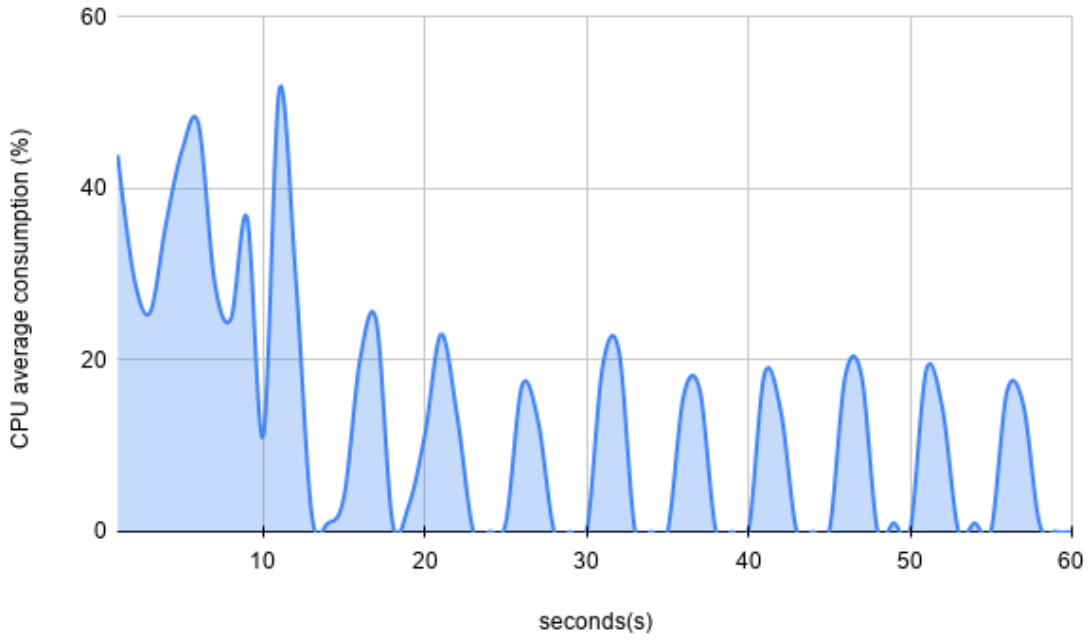

Figure 3.3: CPU behavior while the sohu.com page is being loaded

The pages mentioned loaded a lot of scripts and images, to the client-side. These pages were known to be heavier and have a higher post-processing load. Some times even taking higher time frames to respond and get the resources. Most of them at time were hosted far from the crawler machine. These were all good factors to consider. Once a miner started, even if the

machine lost connection to the server or it took longer to respond there was no impact on the miner without reloading the page. It would just keep mining.

For the crawler we chose to use a project publicly available on GitHub called Puppeteer-Cluster [tho20]. The project takes advantage of another project called Puppeteer which is a Node.js library that provides a high-level API to control Chromium and has the capability to spawn a pool of a given number of Chromium instances. We did not have the intention of spawning multiple instances of chromium tabs since we needed the tabs to be running by themselves to get good values to use in our training module. Having multiple tabs running at the same time could lead us to some wrong conclusions and bad training values. We could if needed at a later stage of development to capitalize on it to have a great crawler even multi-threaded if there was a need for it, which ended up not being the case. The project also had a good way of handling the top Alexa list [Ale]. Which is the list we are going to use to perform the crawl trying to find pages containing cryptojacking infected pages. The top 1 million Alexa list is the result of a rank that rates a million of websites by order of popularity. It takes into account an estimated average of daily unique visits each website gets and also the number of page views of the previous 3 months to get a good estimate of how many hits and how popular each page is compared to other ones. Using the previously described observations a rank is attributed to each website where the rank 1 is the most visited and popular web page of the list. The puppeteer-cluster project even had some good examples of previously developed crawlers that we could use as a solid base to develop our solution. We just needed to perform some modifications to adapt the example to our needs, and also add a routine that always spawns our chosen tool to capture the CPU values (`MPSTAT`) to extract the metrics that are later to be evaluated.

We also did run a blacklist through our alexa top page sample trying to understand if it was possible to detect any cryptojacking on our sample. We chose to use a tool written in golang which was fast gathering all the URLs present in the chosen websites called gospider [jp20]. This would be run simultaneously with one of the crawls and after getting all the data a blacklist extracted from a repository adblock-nocoin-list [hos20] and compared to the gathered URLs. We can then understand if there were any miner traces. This was not the core of what the research was about, and we did not put much emphasis on the values gotten this way. Since this is a widely known way to discover miners, pages with a heavy hit counter are probably extra careful to make sure a miner is not detected by such trivial methods. And in fact none of our sample pages tested positive for being actively running a miner with these blacklist method.

There was also some hand evaluation of the scripts loaded by a sample of pages looking for

keywords that could indicate something is running, like “coin”, “throttle”, “miner”. As well as a by hand load of these pages observing the CPU values looking for high and constant values. This is not by any means the best way to be sure there are no scripts mining since the human being is not perfect and errors are always possible to exist with this kind of observations. But no solution exists to date that gives an error free result, and the best we can do is to perform as many tests as possible with a defined methodology to have a cross-check among all of them to be able to rule out the possibility of a miner being present at a given web page. Also, there are extra measures an attacker can use to bypass all of these checks, but some of them can lead to lower generated profit from the compromised web page (assuming that the web page administrator is not the one injecting this kind of monetary gratification on their pages).

We set up an account with Coinimp which is a crypto mining service that provides a Javascript API easy to embed in any web page. We started testing it with our CPU measures running in background, changing the percentage of CPU used at each iteration, and we observed the amount of CPU mentioned in the Coinimp script was directly tied to the amount of CPU used by the browser and consequently the computer. The percentage chosen was not the percentage of CPU that the browser had attributed to it but the overall CPU usage existing on the computer. By choosing a 100% load on a mining script the user computer would slow down to a close to unusable scenario since there was not much CPU available to perform other tasks. This was the most greedy and careless scenario an attacker could perform, and although most likely not to be found on the wild at the moment, it was still a valid scenario that could exist, and should not be dismissed.

There were differences in the CPU metric that spiked using Firefox and Google Chrome. Using Chrome the value that got higher while mining was the (%usr) value while in Firefox the value that indicated the presence of a miner was (%nice). We decided to proceed using Chrome since it was part of the initial design to use a Chromium crawler. This does not exclude the possibility of also checking if the solution is also valid for Firefox but this was not our main focus for now.

After making sure there was a detectable high CPU consumption, which could indicate the presence of a miner, we could step into the next phase of developing a crawler. Figure 3.4 represents the CPU consumption of a mining script running in a browser configured with different throttles. From being limited to 25% of the CPU to having all the CPU for him.

Since the work was time sensitive we decided to reduce the sample substantially and go with just the top 12500 web pages from Alexa top 1 million.

Before the crawling process took place there was the need to understand if there was any

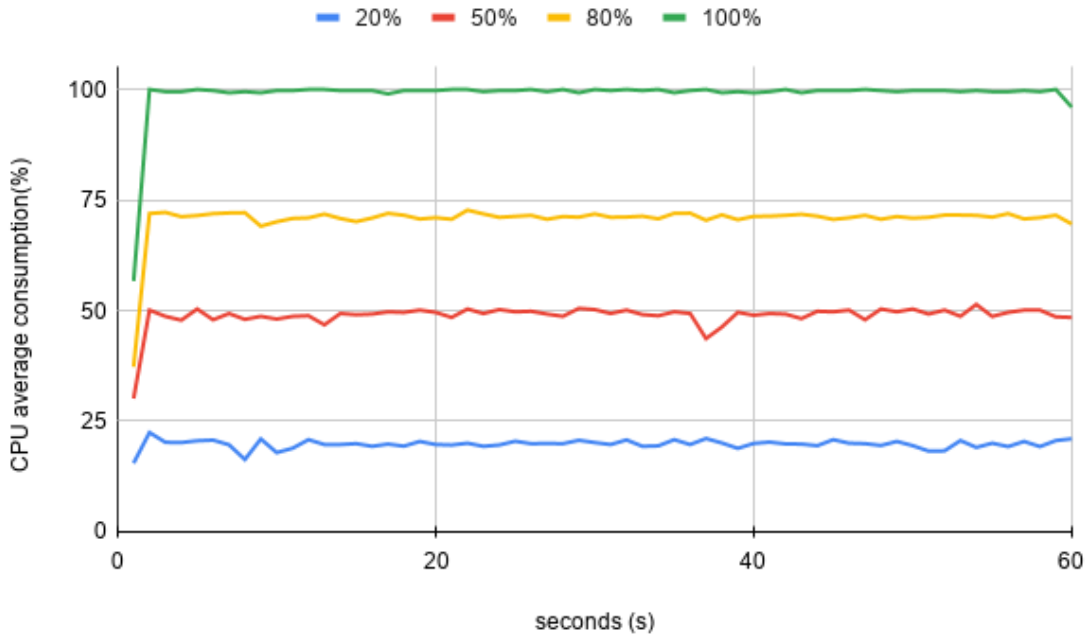

Figure 3.4: CPU behavior while mining at different with different throttling

difference between gathering samples in virtual machines or physical machines. There was none since the CPU metrics were taken at the Operating System level and it only matters how much of the existing percentage was used there was an abstraction on the overall existent hardware. The only visible difference is how many CPU cores are given to the virtual machine and even then it makes no difference in the mining process. The miner only looked at the percentage of CPU being used and not how fast it could go, or if it had 4,6,8 or more cores. The faster a CPU was the higher was the generated mining power in percentage. To better explain it lets take the following example. A CPU power is measured by the number of cores it has and the amount of GHz it can reach which is the velocity it can process its tasks so a CPU with higher core numbers and higher GHz will have more power than a lower-end processor. So when accounting for its percentage, 60 percent of a high-end CPU is inherently more powerful than 60 percent of the worst CPU. This meaning that by getting our metrics in terms of percentages we could abstract ourselves from the hardware that was providing the computational power and focus on a broad way to analyze the values and make some sense out of them.

To make the metric gathering faster, we chose to use Amazon EC2 instances, four to be more precise. All only with one CPU core, with 512 Mb ram and 40 Gb internal storage. Dividing our sample crawling between 4 machines that were images from the same initial instance, taking four times less to perform and it does not impact the metrics since the instances are exactly the same. `MPSTAT` had the options to read the CPU usage by core and to give an average of all the

cores, to be able to read in all the environments with different cores we ended up reading the average value for all the cores in any machine. The miner throttling system also referred to the average overall and not the average by CPU core.

At this point with the core tools developed and ready, we still needed to go through a few more points before doing the top web pages sweep.

We did a few tests to understand how long we need to capture a web page CPU consumption and make the classification algorithm recognize it as a cryptojacking. We also tried not to take too long in each page visit, since most of them are only visited for a few seconds before getting halted. We started by capturing it for 1 minute, then 30 seconds, and finally 15 seconds. During these captures, we tried to figure out the time it takes for the full page to get loaded and to have it CPU usage decreased. We figured 15 seconds was a perfect amount of time since it was not too long so the user would leave the page and it was long enough to have most pages load. Our measures found out it took between 3 and 10 seconds for most of the pages to fully load. Of course, this depended on multiple factors, and it could end up taking significantly more depending on the server load and the network speed. But in normal conditions our assumptions were solid.

There was the need to capture measures to train the algorithm, and for this, we needed to know if a page was indeed mining or not and do it in different environments. We made our crawler crawl a page served by us for 15 seconds each time and do it 60 times in each of the groups, divided into four different groups: first group we left the crawler run without any interaction a page containing a miner, second group we made it crawl a page mining while there was some normal user interaction with a computer (googling some pages, running some programs and so on), third we crawled a simple web page served by us without any mining action happening, and at last we crawl a non mining page while a user was performing some normal actions on the machine. We further decided to subdivide the two first groups into 4 more groups each which the only difference is the percentile of CPU used in the mining job, the values chosen were 25%, 50%, 75%, and 100%. With all these data points, half mining and half not mining in different conditions, we started searching for the classification algorithm that would give a result as good as possible. At the beginning there were no good algorithms, the precision was always bellow the 80%. The way Weka got the input results was by giving it Arff files. The files required a classifier at the end of each line of input that told it if it was classified as mining or not. We had 15 lines of values for each individual crawl which was leaving Weka to evaluate each line individually and not each crawl as a whole. This was probably why there were no good results, so we started using relational bags. Relational bags were only supplied

in a package called “mi” (which stands for multi-instance) and this package was only available in Weka developer release. These bags had some advantages when compared with the default Weka packages when applied to our training data being them:

- It has multiple instances in an example;
- Only one class label is observable for all the instances in an example.

Leveraging this package we would provide each crawl even as a whole and give it a classification of 1 if it was running a miner and a classification of zero if it was not.

There were, while this second crawl action was happening, 3 EC2 instances running the gospider getting the mentioned URLs to make sure the pages were not changed while the different crawlers were running. All the resources gathered were grepped against all the existent endpoints on the blacklist and all the pages having traces of miners were stored and compared with the results gotten from our algorithm.

3.4 Classification

This chapter discusses the classification process that would determine if a page was indeed running a miner or not. The first phase of this research was done by running a simple cryptominer self-served and using it to observe the metrics differences output by all the tools. A simple webpage containing the script that is triggering the mining process was added to a simple HTTP page that is present on the annexes of the present document. The script needed to be served using a HTTP service, for this we ended up choosing to serve it by using a pre-built Python module called SimpleHTTPServer which allows for a directory containing multiple pages to be accessed/served locally, using the localhost IP (127.0.0.1). The described module works just like a normal nginx/apache server serving just the present directory where the script was triggered from via terminal but with a lower complexity. Since the requirements were just to serve a simple HTML page containing some script tags this was a good fit.

We started by training our algorithm, using a cross-validation method [Bro09] since after some careful evaluation we got to notice that the cross-validation method is used more often and produces more pessimistic results which allow us to have a better perception of what our solution can and can not do. This training model is known as a k-fold cross-validation, where k is the only parameter and it refers to the number of groups the supplied data was divided into. The cross-validation method is better than others to our approach since it helps to have a better and more real estimate on how the model is expected to behave against data not used in

the training set. It gets us less optimistic results. According to Brownlee [Bro09] the procedure is the following:

1. Shuffle the dataset randomly;
2. Split the dataset into k groups;
3. For each unique group perform the following actions:
   (a) Take the group as a hold out or test data set;
   (b) Take the remaining groups as a training data set
   (c) Fit a model on the training set and evaluate it on the test set
   (d) Retain the evaluation score and discard the model
4. At last make a summary of the skill of the model using the sample of the model evaluation scores.

To sum it up these method starts by randomly dividing the set of observations into k groups of approximately the same size. The first group is treated as a validation set and the method is fit on the remaining k-1 folds.

The more folds we chose to do the higher the number of cross-checks would be and the longer it would take to compute a model. We ended up choosing a 10-fold cross-validation training set as the first option since it was widely used in other machine learning projects, had a good number of checks to start with and wouldn't take large amounts of time to compute.

During the experiments, we tested multiple classifiers to find those that provided better results given our dataset. The following classifiers were those that we ended up having better results with, although we evaluated many others. The first two algorithms were applied to multi-instances which allows for a single label to be applied to multiple values (relational bag), whereas the final two were applied to single instances.

- **TLC**, Two-Level-Classification [WFP03] uses a single decision tree to obtain propositionalised data. TLC represents regions with assigned attributes in its space. Each attribute represents the number of instances in the bag that can be found in the corresponding region. Together also considering the bag's class label, the meta-instance can be used with a standard propositional learner in order to learn the influence of each region on a bag's classification.
- **MISVM**, Multiple-Instance Support Vector Machine [ATH03] is a machine learning approach that leads to mixed-integer quadratic programs that can be solved heuristically.

The algorithm first assigns the bag label to each instance in the bag as its initial class label. After that applying the algorithm SMO to compute the support vector machine (SVM) for all instances in positive bags, finally, reassign the class label of each instance in the positive bag according to the result and iterate them until the labels stop changing.

- **RandomSubSpace**, Random Subspace Method [Ho98], constructs a decision tree based on the classifier improving the generalization accuracy as it increases in complexity. The classifier consists of multiple trees constructed systematically by pseudorandomly selecting subsets of components of the feature vector, meaning constructing trees in randomly chosen subspaces.

- **SMO**, Sequential Minimal Optimization [Pla98], substitutes all the missing values in the dataset by binary values, normalizing them. Pairwise classification is used to solve multi-class problems. It is a very scalable algorithm since it breaks Quadratic problems in small ones solvable analytically.

The mentioned classifiers were used in each iteration as some of those having better results. A lot more classifiers were used in an effort to understand if they could be used as a good fit but none of them gave us good results.

There was no perfect way to validate the presence of webminers, a constant everywhere for this is the presence of a constant and usually big CPU usage. After this behavior gets spotted there is an extra need to check if there is any script being loaded that can be pointed as a miner.

To aid with the miner detection there is a phase of gathering all the URLs present in the pages. All the URLs are grouped by page and a URL blacklist runs through all the resources trying to find traces of cryptominers on them. The URL blacklist is constituted by multiple lists trying to cover as much ground as possible allowing for a truly decent term of comparison against the model that we are trying to construct on CPU monitoring. This step is done by hand, where a sub-sample of the crawled pages chosen and analyzed for the presence of such scripts as well as to evaluate the reason why the page is giving such a result. Since there is no good mechanism to prove this presence with no doubts besides going through the page by hand.

We understand that we can not analyze a big number of pages by hand in a reasonable amount of time and that by extrapolating the results of a small subset of pages to a result of hundreds may lead to errors and the result gotten from it may not be entirely accurate. But this was a needed evil.

This chapter presented the actual implementation choices, technologies used, and the reason why they were chosen instead of some other options described in previous chapters. It describes the problems and setbacks we found during the development of our solution. Some used tools were also further explained in this chapter.

Chapter 4

Experimental Evaluation

In this chapter, we explored our schema, monitoring different numbers of cores for different time frames to understand how to configure our detector. It is divided into 5 different parts that reflect 5 different experiences, adapted at each iteration with the results of the previous ones. Since we did not know what kind of results to expect, they fully result from what has been observed.

4.1 Experiment Considerations

It is important to mention that there was no way to obtain ground truth since there is no way to assert for sure if a webpage is running a cryptojacking miner or not at the moment of writing of this research. This is a problem we are trying to solve. Our values are extrapolated from a laboratory environment to the real world. Our precision rates, false positive and true positive values are evaluated against our training examples so there is the possibility of some deviations once we apply our training model to the real world testing set we chose to use.

Weka did have a good number of evaluation metrics we could use. We made the choice to keep them in the tables across Chapter 4 but we did not analyze some of them when choosing the algorithm that better fit our research.

There were three metrics we focused on, that had a higher degree of importance once evaluating the results, first of them the precision rate. The Precision outputs the accuracy of a class's correct positive prediction. Precision is the calculated value between the True Positive divided by the sum of the True Positives and False Positives, represented by the following equation:

$$Precision = \frac{TruePositives}{(TruePositives + FalsePositives)} \tag{4.1}$$

The other two crucial metrics are the following:

- False positive rate which translates the amount of pages that were flagged as being running malicious scripts when indeed they were not;
- False negative value, which illustrates the amount of pages that were not flagged as malicious when they indeed were malicious.

It matters to understand that the false negative rate is the same as having the false positive rate of "no miner running", which happens in some of our tables where there are the different values of false positives for both options (running cryptojacking or not).

Both of these metrics were important in their own way. The false negative rate (false positive of "no cryptojacking present") was more important to keep at a closer to zero value than the false positive rate (false positive rate of "crytojacking present") because our main concern was to make sure the algorithm was able to catch all the malicious pages to warn a user that something was happening, so they could close the page as soon as possible. Secondly, the false positive rate was also important but not as much as the previous one to make sure we did not mark pages that were not as malicious as such. Even if this behavior happened we were just warning the client wrongly, this means that the client was still "protected" but we were wrongly accusing the web page of something that was not happening and it could make the client not trust the web page anymore. Table 4.1 can help understand our previous statements, since false negatives of the generality are the same as false positives of the nonpresence of cryptojacking we decided to also include the true positive rate.

Class	TP (True Positive) Rate	FP(False Positive) Rate
0 No cryptojacking	Stating there is no miner running when in fact there is no miner present	Stating that there is no miner running but in reality there is a miner present
1 cryptojacking	Stating that there is a miner running when in fact there is a miner present	Stating that there is a miner running when in reality there is no miner present

Table 4.1: Table explaining the False positive rate applied to our use case.

The Recall value is one of the presented values that we did not look for when taking conclusions of our results. Recall outputs the amount of correct class predictions out of a given class existing examples. It is computed using the following equation:

$$Recall = \frac{TruePositives}{(TruePositives + FalseNegatives)} \tag{4.2}$$

Next F-Measure takes into consideration the values of Precision and Recall, and it outputs a single value that was created to have a metric based on both Precision and Recall. F-Measure

is calculated by the following equation:

$$F - Measure = \frac{2 * Precision * Recall}{(Precision + Recall)} \tag{4.3}$$

MCC which stands for Matthews correlation coefficient is mentioned in machine learning as a binary quality measure. The values lie between -1 and 1, where +1 is a perfect model and -1 is a bad model.

$$MCC = \frac{(TP * TN - FP * FN)}{[(TP + FP) * (FN + TN) * (FP + TN) * (TP + FN)](1/2)} \tag{4.4}$$

TN stands for True Negative, TP for True Positive, FP False Positive and FN False Negative.

Finally, we have ROC Area and PRC (Precision-Recall curves) Area which are diagnostic tools for binary models.

- ROC Area is usually used when they are close to the same amount of samples from both binary outcomes, a result between 0.8 and 0.9 is considered excellent while a result above it is considered outstanding.

- Precision-Recall Curves Area (PRC Area) is usually considered when the number of both samples is imbalanced. A PRC Area with a greater number will mean that there is a high precision and high recall value meaning there is a low false positive and a low false negative value respectively.

4.2 Training Dataset Composition

We started gathering the training metrics by dividing them into different groups where we would gather the metrics 60 times for the specified amount of time in each experiment, which varied between 15 and 60 seconds. The groups and sub-groups in which we divided our training dataset are the following:

- *Running a cryptominer while no one is working on the computer* – We did this at different CPU consumption rates (20%, 50%, 75%, and 100%) giving a total of 240 runs (60*4)

- *Running a cryptominer while someone is working on the computer* – We did this at different CPU consumption rates (20%, 50%, 75%, and 100%) giving a total of 240 runs (60*4)

- *No one is working on the computer and no miner is running* – This is control group which will give us the CPU normal usage

- *Someone is working on the computer but no miner is running* – The idea is to have entropy and understand the difference between mining and normal user usage. The usage here described was essentially browsing different webpages, running some java programs and some bash scripts. None of them are CPU heavy tasks.

We did supply a higher number of metrics flagged as having a miner running than those not having anything running. It was more important to our work, not to miss any page mining, so we tried to supply as many examples on the training set as possible. This premise was considered faulty in later runs, and we decided to add some more values of pages not running any miner script, to make sure the algorithms had a chance to establish a difference between the presence or not of a miner.

4.3 1 CPU core for 15 seconds

We started by configuring our detector to monitor 1 core and to obtain metrics for periods of 15 seconds.

After all the metrics gathered, parsed, and compiled into an Arff file divided into relational bags we ran all the algorithms in Weka that could be applied to our values. Looking at the initial results we ended up choosing the algorithm with better precision values among all the existing classifiers, which was the TLC classifier [WFP03]. Evaluating the results and keeping in mind it was our first experiment we got a better result than anticipated, with an average precision of 92.5%. While not being a good result for a final solution, it showed us that with some improvements it could be possible to have a working solution at the end of our trials. Also, we got only an average of 7.5% false positive rate.

We started by training our algorithm, using a 10-fold cross-validation. Table 4.2 shows the training results we extracted from Weka for the algorithm with the best values given our training set. Weka performs after the training step a validation with the classifier and the training data to verify if the newly trained classifier will be able to rightly classify the supplied values. The classifier was called (TLC). The first training values were gathered by loading the pages for 15 seconds on a 1 core machine.

Class	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area
0 No cryptojacking	0.940	0.090	0.913	0.940	0.926	0.850	0.966	0.943
1 cryptojacking	0.910	0.060	0.938	0.910	0.924	0.850	0.966	0.963
Weighted Avg.	0.925	0.075	0.925	0.925	0.925	0.850	0.966	0.953

Table 4.2: Training results using the metrics for 15 seconds and the TLC algorithm

We obtained a reasonable result for a first run. At this point, we started to adapt our

crawler to go through the chosen dataset, the top 12500 Alexa websites.[1] For this purpose, we initiated four EC2 instances all from a single image. After running the first sample for a while we realized that some websites were not responding or were dead. So some modifications were needed to make sure we got the amount or results we wanted. We ended up running the crawlers against 12500 web pages where only 11489 ended up being useful to our work. At this point we noticed that we could not use the metrics gotten by core since different systems would have a different amount of cores and so compare this would not be accurate, so we ended up using the average by second of all the cores. We parsed all the results and compiled them into two files: one containing all the relational bags of the web pages and the second with all the web pages URLs, so we could match each result to each web page. This was also the approach for the rest of the runs in our experiments, there were always two lists, one with the CPU results and another with all the URLs of the crawled pages. The results were unexpected given the training set. We found 1837 possible pages containing cryptojacking malware running on them, given the 11489 active crawled pages sample that is 15.99% of the active sample. We found this to be unrealistic since it was a much higher number than those observed in all the related work. Given nowadays conditions, and since the mining scenario has been going down because of the increasing difficulty of mining coins like Bitcoin (the difficulty changes with the amount of effort being put in mining [Cor19]), the decreasing coin value, and the main cryptojacking player (Coinhive) stopping its operation, gave us much less confidence in these results. Therefore we decided to analyze a small sample of the pages manually.

After checking the following sample by hand:

- http://tmall.com
- http://sohu.com
- http://taobao.com
- http://jd.com
- http://alipay.com

We noticed no miner was running and the pages did indeed yield a higher CPU usage for a while longer than the 15 seconds. This observation made us apply a change in the way we gathered the metrics. We decided to extend the metric gathering stage and instead of doing it for 15 seconds we gathered them for 60 seconds (next section). As mentioned before we would prefer to have a higher false positive rate on the presence of a miner than on the non presence of a

[1] https://www.alexa.com/topsites

miner, since it is a trade worth taking, marking some pages as malicious while not being, instead of not flagging those that must at all cost be flagged.

4.4 1 CPU core for 60 seconds

We did a second experiment changing the amount of time each page is visited, increasing it to 60 seconds so a new set of metrics was gathered.

After all the results were parsed and applied to the algorithm previously selected (TLC), we had an even better performance, which made sense. By increasing the amount of time we gather metrics there would a higher number of values that could be used to trace a pattern and then apply it to unknown pages. Table 4.3 shows the results we obtained.

Class	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area
0 No cryptojacking	0.988	0.021	0.976	0.988	0.982	0.967	0.992	0.980
1 cryptojacking	0.979	0.012	0.989	0.979	0.984	0.967	0.992	0.993
Weighted Avg.	0.983	0.016	0.983	0.983	0.983	0.967	0.992	0.987

Table 4.3: Training results using the metrics for 60 seconds and the TLC algorithm

Those were good results. We jumped from an average of 92.5% to 98.3%. With the increased precision we expected better results when evaluating the new 60 seconds sample of the 12500 pages against the newly created training model. If we compare it to the false positive rate we had a promising result of 1.6% on average. Which makes us believe the results should be greatly improved.

We ran our new solution against the sample of the 12500 pages and compare it to the previous one.

In the second run, of the 12500 web pages, only 11593 responded successfully, a few more than the last time. Of the total sample, we got 982 pages marked as being running mining scripts. Although this still looks like a big number it is close to a 50% reduction from the last run, associated with a low false positive rate made us believe our results may be accurate. We ended up analyzing some pages by hand to make sure they were running scripts or not, we chose a small sample of the first 10 web pages flagged as being mining, which was the following:

- http://friv.com
- http://vnexpress.net
- http://wiley.com
- http://orange.fr

- http://gamib.com
- http://tempo.co
- http://rockstargames.com
- http://news.com.au
- http://telekom.com
- http://filgoal.com

After a careful evaluation of the chosen pages sample we excluded 3 of them since they did not yield a high CPU usage for a long period of time, it was not understood why the model flagged these pages since even the first crawl results did not have a high CPU usage all through the time. If they had, then a miner running, there were a number of reasons why we would not find it at the time of result evaluation, the top two being:

- the miner was removed before the evaluation;
- the miner only loaded from time to time, to some users.

Page "tempo.co" was a dead-end, it indeed had a higher CPU usage, for a longer time frame but it dropped a few seconds after the metrics stopped being recorded. "news.com.au" had a higher CPU usage but it was due to the amount of resources loaded, which was also the same reason for "telekom.com" and "filgoal.com". The only 2 remaining to analyze were number 1 and number 5. This pages indeed had a high CPU usage, we started by loading the page and understanding what was it doing. The first page was an online game web page doing computation on the client side. It is normal for such web pages to have a high CPU usage and stable if not a lot of interaction is applied. To make sure nothing is running in the background unwanted, we carefully downloaded all the resources that made up the page and analyze them looking for scripts that could indicate a miner. Nothing was found we ended up running the page on a local web server to check if the behavior, but it was a dead end. There was nothing apparent that made us think there was a miner present. The model failed once again. Page 5 was also a disappointing realization, it showed indeed high CPU usage but it was just an image modeling changing as time passed and even simpler scripts, again not running what we thought it was.

4.5 Average of the cores for 60 seconds

At this stage, we realized that there should be some CPU metrics of webpages publicly available in our training dataset. These pages loaded a higher amount of resources making the CPU have a higher usage for longer than usually observed, while still not running any malicious scripts.

- http://tmall.com
- http://sohu.com
- http://taobao.com
- http://jd.com
- http://alipay.com

These pages were those that got flagged in the first run against the Alexa top pages. We ended up choosing them as a good real-world indication of pages that run for longer on a higher CPU consumption, while not having any malicious cryptojacking present. So the new data was once again put to test with the set of existing algorithms. Figure 4.1 and Table 4.4 captions presented next are composed by the multiple algorithms that were put to test, trying to understand which one would have better results:

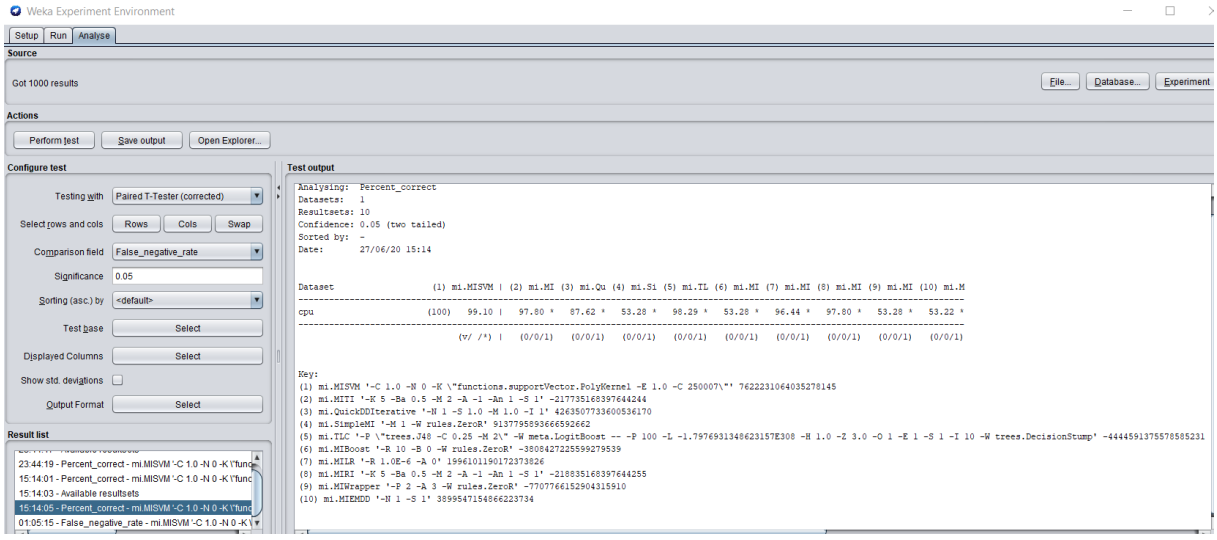

Figure 4.1: Weka precision percentage.

Dataset	(1)mi.MISVM	(2)mi.MI	(3)mi.Qu	(4)mi.Si	(5)mi.TL	(6)mi.MIBoost	(7) mi.MILR	(8)mi.MIRI	(9)mi.MIWrapper	(10)mi.MIEMDD
cpu(100)	99.10	97.80	87.62	53.28	98.29	53.28	96.44	97.80	53.28	53.28

Table 4.4: Training results for the average of the cores using the metrics for 60 seconds

Table 4.4 captions presented next are composed by the multiple algorithms that were put to test, trying to understand which one would have better results:

- **(1)** – mi.MISVM
- **(2)** – mi.MITI
- **(3)** – mi.QuickDDIterative
- **(4)** – mi.SimpleMI
- **(5)** – mi.TLC
- **(6)** – mi.MIBoost
- **(7)** – mi.MILR
- **(8)** – mi.MIRI
- **(9)** – mi.MIWrapper
- **(10)** – mi.MIEMDD

A new algorithm came with better performance: MISVM. It performed slightly better than TLC, although close in terms of false negatives. Table 4.5 shows an in-dept analysis of the MISVM algorithm with the training set provided to it.

Class	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area
0 No cryptojacking	1.000	0.017	0.981	1.000	0.991	0.982	0.992	0.981
1 cryptojacking	0.983	0.000	1.000	0.983	0.992	0.982	0.992	0.992
Weighted Avg.	0.991	0.008	0.991	0.991	0.991	0.982	0.992	0.987

Table 4.5: Training results for the average of the cores for 60 seconds (MISVM)

Something remaining to do was to test against the Alexa top pages. After evaluating the crawled web page values through the new algorithm we ended up with 851 pages with a positive result of the presence of a cryptocurrency miner. Where some of the previously flagged pages got dropped which was a better sign. We chose once again the 10 first pages flagged to get evaluated, the next list reflects the web pages evaluated by hand:

- http://friv.com
- http://repubblica.it
- http://battle.net
- http://gamib.com
- http://tempo.co

- http://falafelandcaviar.com
- http://gjirafa.com
- http://news.com.au
- http://rp5.ru
- http://masrawy.com

All the mentioned pages were once again free of any detectable cryptominer. They either loaded a higher number of resources ending up consuming more CPU, or had heavier scripts that maintained a high usage. Our choice to add some real-world examples paid off, we needed to feed the algorithm with some real world pages so it could find the difference between them and those running miners.

We did consider grabbing the obtained metrics from the 60 seconds crawl and adapt them to 30 seconds. If we had good results to make it usable since 60 seconds is a high time to wait for a miner to be flagged. Even 30 seconds is not a perfect value but since 15 was out of the question it would be the next best option. This step was cut short since the 60 seconds results did not have good values, although it looked like good values at first glance.

There was something extra noticeable. While choosing only to look for the average values of the CPU usage we made a mistake. With a closer look, we noticed although the CPU usage average was high in most of these cases not all the cores had the same value, while our training set of running miners maintained the same average value on all the cores. This discrepancy could still lead us to a working solution. We still had one last question to answer, was there any connection between the number of CPU cores used and the possibility of getting an algorithm that detects a malicious usage of our CPU power?

Also, there was a strange result. With the MISVM we had a 0% false positive rate given our input values. By observing the results we can determine that the given values do not represent the real world since we had some false positives when running against the Alexa top pages, the values we are computing do not allow us to have a precise algorithm. If they did we should not find a single page marked as running malicious scripts that were in fact not running them, which was the case for our sample. Although even if this was a final experience, we would have preferred to have a higher false positive rate of the presence of a mining algorithm.

4.6 2 CPU cores for 60 seconds

At this stage, we recompiled Arff files but instead of just looking for the average values of all the CPUs we also included the CPU values of 2 CPU cores independently.

This realization brought a new problem, a new set of training metrics and a new swipe thorough all the Alexa top pages needed to be done. Since most of the metrics were gathered using EC2 instances that had only 1 CPU core. There was a small sample of metrics that were gathered in a local virtual machine, which was a Linux machine (Kali-Linux Distribution) that had 2 CPU virtual cores supplied to it. The metrics gathered this way were those recorded while the researcher was working on the machine to introduce some entropy, so it would look like someone was using the computer in day to day activities.

When the first set of metrics was recorded we were only looking for the average values of the CPU, and looking at these values there was no difference between having a machine with 1, 2, or 4 cores, since the cryptominer only looked for the average value of the CPU to guide itself to the value preset by the person controlling the miner. It was required to re-record a big set of the metrics we had previously used in our experiments. More than half of these training values.

Besides the training values, all the top 12.5k pages needed to be re-recorded using more capable EC2 instances, instead of using t2.nano instances we ended up using 4 t2.medium instances which had a price close to 9 times higher than the nano instances previously rented. The new rented instances had as mentioned 2 CPU cores and 4 GB of RAM, which we did not need for our experience but it was not possible to downgrade. These instances were left running for a few days while all the new pages were being crawled.

After all the metrics got retrieved, parsed and compiled inside a single Arff file with relational bags while keeping all the MPSTAT output metrics, we got the results present in the Table 4.6 that represents the percentage of precision the algorithm had when tested with our new supplied training data:

Dataset	(1)mi.MISVM	(2)mi.MI	(3)mi.Qu	(4)mi.Si	(5)mi.TL	(6)mi.MIBoost	(7) mi.MILR	(8)mi.MIRI	(9)mi.MIWrapper	(10)mi.MIEMDD
cpu(100)	98.82	98.12	50.77	53.23	97.71	53.23	92.90	98.13	53.23	53.59

Table 4.6: Training results for the precision (%) collecting the metrics for 60 seconds and 2 CPU cores

The number of false positives was once again calculated to have a more complete overview of the results, so we could find out if this new model would be a good fit, the Table 4.7 shows after multiplying by 100, the percentage (this percentage is the average between the false positive rate of the 1 cryptojacking being present and the false positive rate of the 0, there is no cryptojacking present):

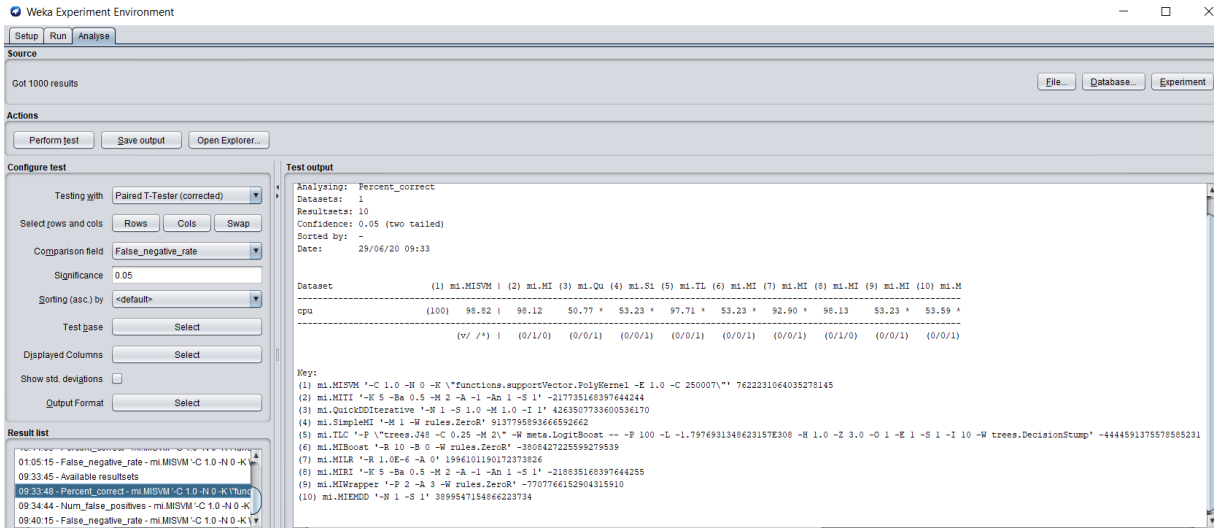

Figure 4.2: Weka precision percentage.

Dataset	(1)mi.MISVM	(2)mi.MI	(3)mi.Qu	(4)mi.Si	(5)mi.TL	(6)mi.MIBoost	(7) mi.MILR	(8)mi.MIRI	(9)mi.MIWrapper	(10)mi.MIEMDD
cpu(100)	0.02	0.02	0.92	0.00	0.02	0.00	0.08	0.02	0.00	0.87

Table 4.7: Training results of classifiers for the false positives number collecting the metrics for 60 seconds and 2 CPU cores

Table 4.6 and 4.7 captions:

- **(1)** – mi.MISVM
- **(2)** – mi.MITI
- **(3)** – mi.QuickDDIterative
- **(4)** – mi.SimpleMI
- **(5)** – mi.TLC
- **(6)** – mi.MIBoost
- **(7)** – mi.MILR
- **(8)** – mi.MIRI
- **(9)** – mi.MIWrapper
- **(10)** – mi.MIEMDD

After observing the previously mentioned table with the same set of algorithms we used in the previous runs we had the first one (MISVM) as the best choice for the mining detector, but evaluating the percentage of correct results 98.82% and the average percentage of false positives

of 2% is still a good value, although not being the best one we got until now. This value, although not critical for our evaluation, represents the number of pages falsely set as running malware, which could induce the user in error.

The new training metrics ended up having worse performance than the previous runs, which was also a dead-end for a good way to predict the presence of a miner.

4.7 4 CPU cores 60 seconds

After a careful evaluation of the results and metrics gathered across all experiments, we ended up finding a link among all the results that could be tied to a miner. A computer with a higher number of CPU cores (let us say 4) usually even with pages that take longer to load or consume more resources to keep running does not use all the cores at full capacity or even all at the same level. But once they start running miners the values across all the cores are a lot closer. Even if one of them goes down for a single second another one would take its place, and the average of all the processors would be almost identical. Which is something that does not happen in a usual page without any miner. In the usual web pages (not mining) although having a higher CPU consumption at first, the percentage of each core being used is not constant. But the CPU consumption value by core of mining pages only varies by 1 or 2 percentage among them. With this in mind we ended up taking a last set of metrics using a local virtual machine with 4 CPU cores, to validate the observations, which are represented in the following figure:

To get this last set of metrics a local virtual machine was used running a Linux distribution (Kali Linux), giving it the above-mentioned number of virtual cores. Here we gathered the training metrics using the same methods mentioned earlier also maintaining the different loads of CPU mining as well as gathering some metrics working on the machine, so we could keep the values as close to reality as possible. The same set of training web pages was crawled to while there was no one working on the machine doing another CPU intensive tasks.

At this point and as a final test two different solutions were developed, one with a legacy Arff file where we did not use the relational bags, and we carefully choose the values that matter the most and another one not using any machine learning algorithm but looking at the averages and deviations of these values to try to make sure they did not have a huge deviation among them.

To find if a miner is running with MPSTAT and using Chromium, we just need to look at the value of *%usr* across all the cores and make sure the values stay high and with a value close to each other with a minimal deviation. Of course, this observation is only viable if you are not running anything else on the computer that uses the CPU heavily, actually, this condition

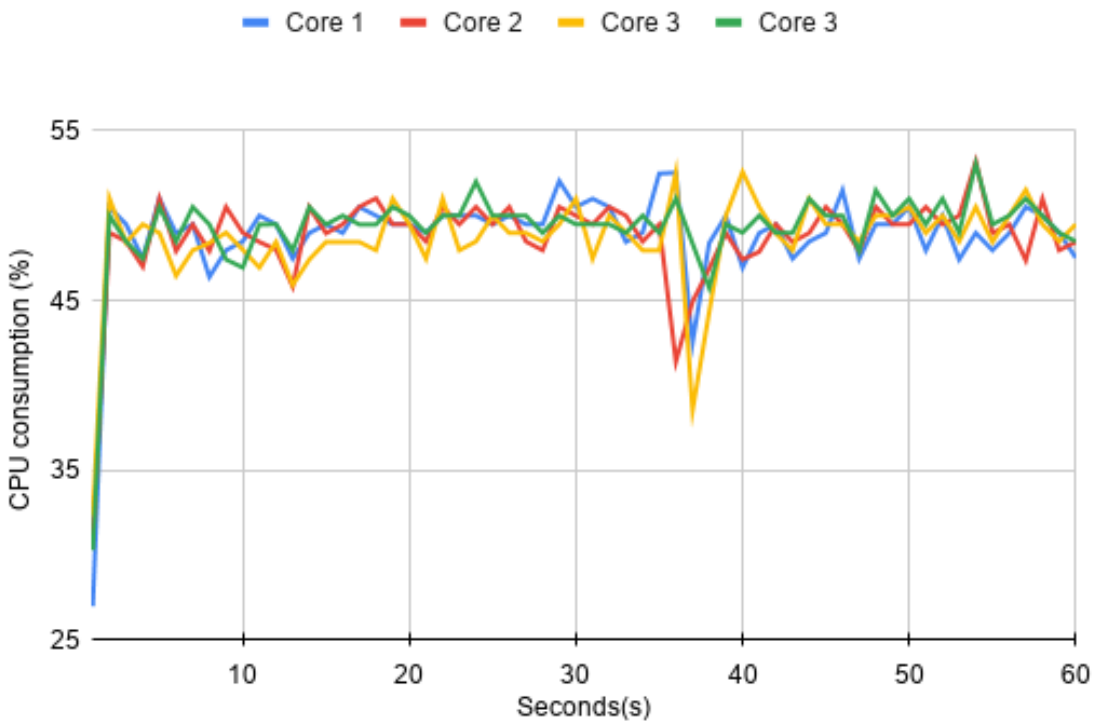

Figure 4.3: Behavior of the 4 cores while mining at 50%

was always a problem since it would possibly produce erroneous results in all the solutions we developed until now.

Our custom-made solution developed by hand without using machine learning gave a lower false positive rate of our training set but it had a big downside, it failed to find dangerous pages we feed that were running crypto miners at a 100% load, which are those that should never be missed since by our observations no pages in the wild require 100% of the CPU for long periods of time.

For this iteration, all the values of MPSTAT were stripped and only the values of each core for %usr were kept in a list, without relational bags. The new data sheet was compiled on a new Arff file and a new algorithm chosen. We ran our training set with all the algorithms present on Weka that could be used on our dataset and after some time the results are the following:

- **95.53** – (1) meta.MultiClassClassifier
- **99.44** – (2) meta.MultiClassClassifierUpdateable
- **66.67** – (3) meta.MultiScheme
- **99.42** – (4) meta.RandomCommittee
- **98.97** – (5) meta.RandomizableFilteredClassifier

- **99.67** – (6) meta.RandomSubSpace
- **66.67** – (7) meta.Stacking
- **66.67** – (8) meta.Vote
- **66.67** – (9) meta.WeightedInstancesHandlerWrapper
- **66.67** – (10) misc.InputMappedClassifier
- **66.67** – 11) rules.ZeroR
- **95.53** – (12) functions.Logistic
- **99.44** – (13) functions.SGD
- **66.67** – (14) functions.SGDText
- **99.08** – (15) functions.SimpleLogistic
- **99.44** – (16) functions.SMO
- **82.19** – (17) functions.VotedPerceptron
- **98.92** – (18) lazy.IBk
- **77.83** – (19) lazy.KStar
- **98.86** – (20) lazy.LWL
- **99.22** – (21) meta.AdaBoostM1
- **98.97** – (22) meta.AttributeSelectedClassifier
- **98.89** – (23) meta.Bagging
- **77.19** – (24) meta.ClassificationViaClustering
- **99.17** –(25) meta.ClassificationViaRegression
- **66.67** – (26) meta.CVParameterSelection
- **99.17** – (27) meta.FilteredClassifier
- **99.28** – (28) meta.IterativeClassifierOptimizer
- **99.31** – (29) meta.LogitBoost

We had a lot of close results with little to no difference, but the one that got the better result was the number 6 (meta.RandomSubSpace) The Table 4.8 shows the values we got out of the training to this specific algorithm:

Class	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area
0 No cryptojacking	1.000	0.004	0.992	1.000	0.996	0.994	0.996	0.976
1 cryptojacking	0.996	0.000	1.000	0.996	0.998	0.994	0.996	0.999
Weighted Avg.	0.997	0.001	0.997	0.997	0.997	0.994	0.996	0.991

Table 4.8: Training results using the metrics for 60 seconds and 4 cores (RandomSubSpace)

Our training values were in part composed of 300 different crawls in which the top 5 pages found on the top Alexa were fed to it as non-mining pages and each page was ran 60 times. Looking at the value it got us a 99.7% of precision which was a great result the best one so far. Which meant we were in the right direction. Also, we can see we were getting a false positive rate of 0% on the pages that were running cryptojacking, having a 100% success rate on them, not missing a single one, which is a very good result. We have a perfect false positive rating in one of the values we defined as the more important, although it would be better to have the inverse behavior and have a 0% false positive rate at the "No running cryptojacking"!

Due to some time constraints and some economic restraints (instances with 4 cores had a big cost) we decided to keep the sample smaller, and we crawled only the top 2500 Alexa pages with our previously mentioned crawler. When all the metrics were gathered the new results were parser with the same method as the training dataset the only difference being the last character that instead of indicating a presence or not of a miner, we used the character "?" to indicate we did not know if anything was running or not.

Using this algorithm against the top 2500 Alexa pages (from which only 2341 did respond) it flagged 83 pages as potentially be running cryptominers. All the 83 pages were analyzed by hand the following 10 are the first pages that got flagged on the Alexa top 2500:

- http://amazonaws.com
- http://liputan6.com
- http://kompas.com
- http://speedtest.net
- http://chaturbate.com
- http://google.co.id
- http://6.cn

- http://crptgate.com
- http://homedepot.com
- http://nianhuo.tmall.com

All the 83 pages were ruled out since the CPU results were not close enough to signal a potential miner, we did not understand the reason why these pages were being flagged since there was no apparent reason for this behavior. The only page having a higher CPU consumption was *feedly.com* these value was still under 20% and was not very steady, indicating that it was possibly the page is just loading heavier scripts. Or possibly the mining script (if present) was not loaded to the researcher while performing the analysis by hand.

The mentioned pages were mostly having a low CPU usage (around 5% or lower). The results gotten were not ideal. We still flagged 83 pages wrongly, 83 out of a sample of 2500 pages (2341 of those responded). It is a failure rate of close to 3.4% on the "No Cryptojacking", which is way off the values indicated in the previous table by Weka.

Since these results were not as close as expected, by analysing the values we got from the training results table, we decided to use the second best algorithm of the previously mentioned selection, the algorithm number 16 (SMO):

Class	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area
0 No cryptojacking	1.000	0.008	0.984	1.000	0.992	0.988	0.996	0.984
1 cryptojacking	0.992	0.000	1.000	0.992	0.996	0.988	0.996	0.997
Weighted Avg.	0.994	0.003	0.995	0.994	0.994	0.988	0.996	0.993

Table 4.9: Training results using the metrics for 60 seconds and 4 cores (SMO)

By analyzing the values the results of the SMO are slightly worse than RandomSubSpace. But the difference is almost unnoticeable. We decided to run it against the same Alexa top 2500 sample, and we only got 25 pages flagged as having a miner present. Of those 25, 21 were ruled out since the CPU results were not close enough to signal a potential miner, we did not understand the reason why these pages were being flagged there was no apparent reason for this behavior. The remaining 4 pages were further analyzed, being the following:

- http://lanacion.com.ar
- http://windy.com
- http://bd-pratidin.com
- http://friv.com

The mentioned pages were mostly having a low CPU usage (around 10%) but steady values, which only indicates that the page runs some scripts a little heavier than most pages, all except the *friv.com* which had a value of 20%. The mentioned page was already being flagged previously by another of our solutions but was once again evaluated carefully and the CPU values were taken for a longer time frame (240 seconds) which ruled it out as a potential page to be mining. Although we still flagged 25 pages wrongly, we managed to keep the value of failure really low, 25 out of a sample of 2500 pages (2341 of those responded), is a failure rate close to 1% on the "No Cryptojacking" corresponding to a 99.2% of true positive rate on the cyptominer being present as the initial test metrics indicated. It is possible that there are some pages that were not flagged as having a cryptominer present, since we did not have a 0% false negative rate when it comes to the "No cryptominer" metric. But the described behavior is not possible to know for sure since we did not know the prevalence of miners in the tested pages before evaluating them.

4.8 Blacklisting

After our solution development, we decided to cross-check our results running a blacklist against the top 12500 Alexa pages trying to understand if there was any call to resources that usually supply miners code. We started going through our sample (the top 12500 pages from Alexa) scrapping all the resources that are referred in the homepage (as soon as we perform the first GET request) and saving the URLs of all the individually requested URLs by each page and saving them in individual files. From the 12500 requested pages 11592 responded, and we were able to store a lot of individual resources from each page. Some of them loading upwards of 230.000 different resources. We had a realization that even the blacklists results needed to be parsed. Some flagged resources belong to pages that although being in a miner URL wordlist, fit better in the Adware category, they could be on this list due to support of some kind that made it at some point in time to serve cryptominers but this was not the case at the moment.

Despite that, there were multiple pages still loading Coinhive resources that have been decommissioned and although the URL was still present on the pages, there were no mining activities going on. There were also other pages that made reference to the alternative coins webpages that explored this kind of mining (CPU intensive via Web-browser) but not specifically to the scripts that server the cryptominer. We ended up understanding that references to the pages do not necessarily mean there is a malicious script running. Besides the described behavior even with URLs present there is a chance they are not running due to deprecated or dead links, or simply the scripts are disabled to a specific user.

This chapter presented the results we got from the different experiences and their analysis that lead to each subsequent experience. Although we did not find a perfect algorithm we believe the last experiment had consistent enough results that could lead to a successful solution, further discussion of our results will be presented in our last chapter.

Chapter 5

Conclusions

We have shown that by combining a set of CPU metrics it is possible to detect browser-based cryptojacking with high precision. In a perfect world, we would have a 100% precision rate result but at this stage, we had good results given the context.

We can inform a user that a specific page is having a behavior out of the ordinary and is consuming too many resources. This would leave the choice to the user to assume the risk or to stay at the page. The algorithms looked like they were getting good results understanding the difference between a heavy page and a mining page but a mechanism reliant on a single validation method would not be reliable enough.

These values were retrieved in a contained and controlled environment although supplying some values with entropy. In our opinion, we can not flag pages with a 100% degree of certainty running in a non-controlled environment. So, the solution would not be used by all the users in a distributed way, as we established initially. The solution would probably only be used by a company in a centralized way to perform recurrent sweeps through all the pages and doing a new sweep each time a new page was asked by a user that was not part of the database. Finally, apply some kind of rating to the likelihood of each page having a cryptominer running. The information gathered would be feed in real-time to the users via a browser extension, and each user would be responsible to analyze the risk supplied, accepting it, or not. But for this solution to even be viable only 1 crawl for page would not be enough. Multiple crawls should be done with different User-Agents headers, in different moments in time, with different IPs. And even then a good attacker could adapt over time and lead this method to miss classify some pages.

The method of flagging here described would be prone to some false positives as well as the Blacklist method as observed earlier. Both solutions would not be able to complete each other.

Our model would always need to alert a user at any flagged website, and some times wrongly since there are pages running heavy Javascripts. Our solution could be integrated with some

other solutions mentioned in the related work that used the CPU load percentage as a flagging system as a cross-check with other solutions, and instead of having this rudimentary check it would have our last experiment solution increasing this way the selection precision percentage.

We chose to use the machine learning algorithm that had the best results to our training dataset, at this point in time there may not be a perfect algorithm to do such task, but we state this without an absolute certainty, since we did not try it against all the existing algorithms. We did it only against the algorithms found to be a good option using the software we were using (Weka). The best option we found to detect a miner using machine learning was to use a SMO algorithm for 60 seconds on a machine with 4 CPU cores that although having a slightly lower precision rate than RandomSubSpace it gives us a higher degree of confidence not flagging as much pages wrongly.

By a lot of careful evaluation and observation, we came to realize that on machines only running a crawler if the page was running a miner we could see the average values of the CPU to stabilize and having values with a minimal deviation across CPU cores. But this observation could also be prone to error by the same reasons supplied at the beginning of this work and that is also stated next. A mining page that would load the script some seconds after the initial visit, if the attackers were to vary the amount of CPU used by reloading the script each 10 seconds with different values, if the page had other scripts that would make an extra CPU load without being stable or even if the user were using the computer for anything else than searching for a single page at a time the miner flagging would fail and throw to many false results to make it viable.

Additionally, the mining environment is at the time of writing slowing down, a lot fewer people are doing it actively, and even tho there are some news or breaches to make the systems mine for the intruders this is not as usual or as profitable as it used to be. Still defend it could be a good way to substitute the advertisements but only if left to take a small percentage of the users CPU, around 20, 30 %. Which is not viable and not profitable now.

5.1 Achievements

The developed work did manage to tie the values of a multiple-core machine to a mining script. We were able to see a pattern in the mining pages. Given a long enough time frame analyzing the values of all the CPU cores, it would be possible to tie these values to a page running malicious scripts. We are also able to say that this is not a good way to detect these scripts using the day-to-day user machine, since there are too many variables that would influence the detection mechanism that relies on the CPU. It is with no doubt, good to use this as a cross-

check validation, using other detection mechanisms. But this mechanism is very easily fooled by an attacker that aims to do so, since all he has to do is to load the script after some seconds, and the results will no longer be reliable.

5.2 Future Work

As a future work it may be interesting to keep testing algorithms that keep appearing not only available on Weka but in other platforms too. The Artificial Intelligence field is a fast growing one, and we believe this may be a viable option.

This research could and should be integrated with others that also try to solve this problem in an attempt to have as many flagging mechanisms as possible and develop a final solution as precise as possible while having a low rate of False Positives and False Negatives.

The CPU although not the best indicator if there is a running miner on a day-to-day user, this approach could be extrapolated for a solution that would be used on containers only used to detect pages running mining scripts and destroyed after each page load. This solution in group with other monitoring strategies mentioned above could be used to crawl web pages and provide a service that would crawl a page requested by a user or company and rate it, for the probability of it being using a cryptominer.

There is the possibility of developing a solution that would be recompiled at each new test adding the new results as a training metric, to have a model that would learn with time by feeding it new results and hopefully increase its precision output. But this model could possibly be tampered in such a way that it will no longer detect any of the mining scripts. The problem may be avoided or at least mitigated with some tricks. By adding some conditions, defining when the algorithm is allowed to use the metrics to train itself or by running a small test with pre-populated values, after each real-time training, evaluating if it is still working and reversing its state if the solution is malfunctioning, among other.

Although we did had the objective to include the possibility of self-improvement in real time this is not the focus of our research, and we prefer to leave it for future iterations of this research.

References

[ABB+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the 13th ACM EuroSys Conference*, 2018.

[Ale] Alexa top sites. `https://www.alexa.com/topsites`. Accessed: 2020-08-06.

[ATH03] Stuart Andrews, Ioannis Tsochantaridis, and Thomas Hofmann. Support vector machines for multiple-instance learning. In *Advances in neural information processing systems*, pages 577–584, 2003.

[AZV17] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking Bitcoin: Routing attacks on cryptocurrencies. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, pages 375–392, 2017.

[BBD19] Hugo L J Bijmans, Tim M Booij, and Christian Doerr. Inadvertently making cyber criminals rich: A comprehensive study of cryptojacking campaigns at internet scale. *Proceedings of the 28th USENIX Security Symposium*, pages 1627–1644, 2019.

[BG16] Anna L Buczak and Erhan Guven. A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications Surveys & Tutorials*, 18(2):1153–1176, 2016.

[BMC+15a] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. Research perspectives and challenges for bitcoin and cryptocurrencies. *Princeton University, Stanford University, Electronic Frontier Foundation, University of Maryland, Concordia University*, 2015.

[BMC+15b] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. SoK: Research perspectives and challenges for Bitcoin and

cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, pages 104–121, 2015.

[Bro16] Jason Brownlee. *Master Machine Learning Algorithms: discover how they work and implement them from scratch.* Machine Learning Mastery, 2016.

[Bro09] Jason Brownlee. k-fold cross-validation. `https://machinelearningmastery.com/k-fold-cross-validation/`, Accessed: 2020-08-09.

[Car19] João Carreiro. Identification and analysis of cryptojacking: Performance effects. Master's thesis, Instituto Superior Técnico, Universidade de– Lisboa, 2019.

[CC03] Suresh N Chari and Pau-Chen Cheng. Bluebox: A policy-driven, host-based intrusion detection system. *ACM Transactions on Information and System Security*, 6(2):173–200, 2003.

[CGLR17] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. Blockchain consensus. In *ALGOTEL 2017-19èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, 2017.

[coia] Coinhive, browser mining. `https://coinhive.com/`. Accessed: 2018-12-15.

[coib] Coinimp, browser mining. `https://www.coinimp.com/`. Accessed: 2020-08-06.

[coi15] coinmarketcap. Cryptomarket informations. `https://coinmarketcap.com/all/views/all/`, Accessed: 2018-12-15.

[Cor19] Miguel Correia. From byzantine consensus to blockchain consensus. *Essentials of Blockchain Technology*, page 41, 2019.

[COSB18] Domhnall Carlin, Philip O'kane, Sakir Sezer, and Jonah Burgess. Detecting cryptomining using dynamic analysis. In *16th IEEE Annual Conference on Privacy, Security and Trust*, pages 1–6, 2018.

[DDW00] Hervé Debar, Marc Dacier, and Andreas Wespi. A revised taxonomy for intrusion-detection systems. In *Annales des Télécommunications*, volume 55, pages 361–378. Springer, 2000.

[Den87] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, (2):222–232, 1987.

[ELMC18] Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. A first look at browser-based cryptojacking. *arXiv preprint arXiv:1803.02887*, 2018.

[ES14] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International Conference on Financial Cryptography and Data Security*, pages 436–454, 2014.

[GHM+17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pages 51–68, 2017.

[HFH+09] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The Weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[HG02] Alex Hern and Samuel Gibbs. What is wannacry ransomware and why is it attacking global computers? `https://www.theguardian.com/technology/2017/may/12/nhs-ransomware-cyber-attack-what-is-wanacrypt0r-20`, Accessed: 2020-08-02.

[Ho98] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998.

[hos20] hoshsadiq. adblock-nocoin-list. `https://github.com/hoshsadiq/adblock-nocoin-list`, 2020.

[jp20] jaeles project. gospider. `https://github.com/jaeles-project/gospider`, 2020.

[Kin13] Sunny King. Primecoin: Cryptocurrency with prime number proof-of-work. 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*, pages 840–852, 2019.

[LLR02] Yiu-Chung Lau Laurie, Y.C. Chang Lennon, and Sarre Rick. Responding to cybercrime: current trends. `https://www.tandfonline.com/doi/full/10.1080/15614263.2018.1507888`, Accessed: 2020-08-02.

[lp01] Community library project. SciPy. `https://www.scipy.org/`, 2001.

[lp05] Community library project. NumPy is the fundamental package for scientific computing with Python. `http://www.numpy.org/index.html`, 2005.

[Min] Mintme.com coin. `https://www.mintme.com/coin/`. Accessed: 2020-08-06.

[MSVBR19] Jordi Zayuelas i Munoz, Jose Suarez-Varela, and Pere Barlet-Ros. Detecting cryptocurrency miners with NetFlow/IPFIX network measurements. In *IEEE International Symposium on Measurements & Networking*, pages 1–6, July 2019.

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

[NA08] Thuy TT Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, 10(4):56–76, 2008.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[opc15] opcm. Pcm. `https://github.com/opcm/pcm`, 2015.

[PIB20] Ivan Petrov, Luca Invernizzi, and Elie Bursztein. CoinPolice: Detecting hidden cryptojacking attacks with neural networks. *arXiv preprint arXiv:2006.10861*, 2020.

[Pla98] John C. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. *Microsoft Technical Report MSR-TR-98-14*, 1998.

[PLZ18] James Phung, Young Choon Lee, and Albert Y Zomaya. Modeling system-level power consumption profiles using rapl. In *2018 IEEE 17th International Symposium on Network Computing and Applications*, pages 1–4, 2018.

[PMB09] Francisco Pereira, Tom Mitchell, and Matthew Botvinick. Machine learning classifiers and fmri: a tutorial overview. *Neuroimage*, 45(1):S199–S209, 2009.

[PST19] Sergio Pastrana and Guillermo Suarez-Tangil. A first look at the crypto-mining malware ecosystem: A decade of unrestricted wealth. *arXiv preprint arXiv:1901.00846*, 2019.

[Ric13] Willi Richert. *Building machine learning systems with Python*. Packt Publishing Ltd, 2013.

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

[RSD+18] Julian Rauchberger, Sebastian Schrittwieser, Tobias Dam, Robert Luh, Damjan Buhov, Gerhard Pötzelsberger, and Hyoungshick Kim. The other side of the coin:

A framework for detecting and analyzing web-based cryptocurrency mining campaigns. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, 2018.

[SALY17] Shi-Feng Sun, Man Ho Au, Joseph K Liu, and Tsz Hon Yuen. Ringct 2.0: a compact accumulator-based (linkable ring signature) protocol for blockchain cryptocurrency monero. In *European Symposium on Research in Computer Security*, pages 456–474. Springer, 2017.

[SJN+13] M. Seigen, T. Jameson, Nieminen, A.M. Neocortex, and Juarez. Cryptonight hash function, 2013.

[SKM18] Muhammad Saad, Aminollah Khormali, and Aziz Mohaisen. End-to-end analysis of in-browser cryptojacking. *arXiv preprint arXiv:1809.02152*, 2018.

[Smi02] Noah Smith. Yep, bitcoin was a bubble. and it popped. `https://www.bloomberg.com/opinion/articles/2018-12-11/yep-bitcoin-was-a-bubble-and-it-popped`, Accessed: 2020-08-02.

[Sym02] Symantec. Symantec cryptojacking growth 2018 annual security report. `https://resource.elq.symantec.com/LP=5840?cid=70138000000rm1eAAA`, Accessed: 2020-08-02.

[tho20] thomasdondorf. puppeteer-cluster. `https://github.com/thomasdondorf/puppeteer-cluster.git`, 2020.

[TJYD10] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[TS16] Florian Tschorsch and Björn Scheuermann. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IEEE Communications Surveys & Tutorials*, 18(3):2084–2123, 2016.

[wek08] Data Mining Software in Java. `https://www.cs.waikato.ac.nz/ml/index.html`, 2008.

[WFP03] Nils Weidmann, Eibe Frank, and Bernhard Pfahringe. A two-level learning method for generalized multi-instance problems. *14th European Conference on Machine Learning*, 2003.

[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, 2018.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[WS02] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264, 2002.

Appendix A

Code Resources

In this chapter we have some code examples that resemble the ones used to get and parse the mentioned CPU metrics.

A.1 Crawler code

The following code is a JavaScript code that will visit a designated page. When we first GET the page, we already started, in the same script the MPSTAT command for 60 seconds. The same amount of time we waited for each page before halting them. The code will need the correct node modules to run and will need to have the absolute paths changed.

```
const { Cluster } = require('../dist');
const exec = require("child_process").exec
const fs = require('fs').promises;
let i=0;

(async () => {
    const cluster = await Cluster.launch({
        puppeteerOptions: { args: ['--no-sandbox'] },
        concurrency: Cluster.CONCURRENCY_CONTEXT,
        maxConcurrency: 1,
        monitor: true,
        timeout: 65000,
```

```
});

// Extracts document.title of the crawled pages
await cluster.task(async ({ page, data: url }) => {
    await page.goto(url, { waitUntil: 'domcontentloaded' });
    const command2 ="echo "+url+">/media/sf_kali/Tese/last_split_cpu_cores/r
    console.log(command2);
    exec(command2,(error,stdout,stderr)=>{})
    const command ="mpstat -P ALL 1 60 >>/media/sf_kali/Tese/last_split_cpu_c
    console.log(command);
    exec(command,(error,stdout,stderr)=>{})
    console.log("here");
    i++;
    console.log(i);
    const pageTitle = await page.evaluate(() => document.title);
    //const pagebody = await page.content()

    console.log(`Page title of ${url} is ${pageTitle}`);
    await page.waitFor(60000)
    //console.log(`${pagebody}`);
});

// In case of problems, log them
cluster.on('taskerror', (err, data) => {
    console.log(`  Error crawling ${data}: ${err.message}`);
});

// Read the top-1m.csv file from the current directory
const csvFile = await fs.readFile(__dirname + '/own.txt', 'utf8');
const lines = csvFile.split('\n');
for (let i = 0; i < lines.length; i++) {
    const line = lines[i];
    const splitterIndex = line.indexOf(',');
    if (splitterIndex !== -1) {
```

```
            const domain = line.substr(splitterIndex + 1);
            // queue the domain
            cluster.queue('http://' + domain);
        }
    }

    await cluster.idle();
    await cluster.close();
})();
```

A.2 Arff File creator

The following python code is one used to compile the CPU metrics into ARFF file that would latter be feed to the Weka. It is a very simple code that goes through the MPSTAT file and while it does not reach the end of it gets all the lines extracting the chosen metrics and putting them separate by commas.

```
final_arff.py

import sys
import os
import numpy

final_list = []


def write_arf(name):
        counter=0;
        fi= open("mine.arff", "w")

        number=0
        fi.write('@relation cpu\n')
        while number<240:
                fi.write('@attribute feature_'+str(number)+'numeric real'+'\n')
                number=number+1
        fi.write('@attribute is_cript {0,1}'+'\n'+'\n')
        fi.write('@data'+'\n')



        final_list = []
        f = open(name, 'rt')
        a = [[token for token in line.split()] for line in f.readlines()]
```

```
temp2=""
list_counter=0
for b in a:
        x=0
        temp=""
        temp_list=[]

        while x<len(b):
                #print(b)

                if b[x][0]=='%' or b[x][0]=='(' or b[0]=='Average:':
                        break;
                if x==1 and b[x]=='all':
                        list_counter=0
                        break;
                if x==2:
                        final_list.append(b[x])
                if x==3:
                        list_counter=list_counter+1
                        #print(list_counter,temp_list)
                        break;
                x=x+1

for a in final_list:
        fi.write(a+",")
fi.write('?'+'\n')

fi.close()
```

A.3 Weka Classifier Code

The code represented next is the one used to load the training metrics and later load the testing Arff file to classiify it. It relies on the pre compiled Arff file to the training step.

classify.py

```
import weka.core.jvm as jvm
from weka.classifiers import Classifier, Evaluation
from weka.classifiers import PredictionOutput, KernelClassifier, Kernel
from weka.core.classes import Random
from weka.core.converters import Loader, Saver


def predict():
        jvm.start(packages=True)

        loader = Loader(classname="weka.core.converters.ArffLoader")
        data = loader.load_file("./try_alt.arff")
        data.class_is_last()    # set class attribute



        classifier = Classifier(classname="weka.classifiers.meta.RandomSubSpace"
        evaluation = Evaluation(data)
        evaluation.crossvalidate_model(classifier, data, 10, Random(42))
        classifier.build_classifier(data)









        print(evaluation.summary())
        print("pctCorrect: " + str(evaluation.percent_correct))
        print("incorrect: " + str(evaluation.incorrect))
```

```
loader = Loader(classname="weka.core.converters.ArffLoader")
test = loader.load_file("./mine.arff")
test.class_is_last()    # set class attribute


for index, inst in enumerate(test):
    pred = classifier.classify_instance(inst)
    dist = classifier.distribution_for_instance(inst)
    if inst.class_attribute.value(int(pred))=="1":
        print("MINER_ALERT!!!")
    else:
        print("Clean")


jvm.stop()
```

A.4 Classifier main

Simple code where a URL is provided by running "solution.py URL". This main will run all the previous code samples in the defined order. The absolute system paths will need to be changed accordingly for it to work.

solution.py

```
import os, sys
from final_arf import write_arf
from classify import predict

def main(args):
        print("here")
        file= open("/media/sf_kali/Tese/puppeteer-cluster/examples/own.txt", "w"
        file.write("1,"+args[1])
        file.close()
        returned_value = os.system('node /media/sf_kali/Tese/puppeteer-cluster/e
# returns the exit code in unix
        print('returned value:', returned_value)

        write_arf("resulta0")
        predict()

if __name__ == "__main__":
        main(sys.argv)
```