Cryptojacking Detection with CPU Usage Metrics

Fábio Gomes[1,2] Miguel Correia[1]
[1]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa – Portugal
[2]INTEGRITY S.A., Lisboa – Portugal
fmjgomes@gmail.com miguel.p.correia@tecnico.ulisboa.pt

***Abstract*—Cryptojacking is currently being exploited 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 computer performance and potentially reduce the hardware lifetime. We introduce a new cryptojacking detection mechanism based on monitoring the CPU usage of the visited web pages. This may look like an unreliable way to detect mining malware since many web sites are heavy computationally and that malware often throttles CPU usage. 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.**

I. 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 web sites was provided by embedding advertisements (ads) on web pages and pop-ups that in some cases made web sites annoying and others even unusable, leading to the spread of ad blockers. Although ads remain dominant, administrators started looking for other ways to monetize their sites 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 circa 2008 [1], but today there are many others [2], [3]. Bitcoin, Monero, Ether, Litecoin, and many other cryptocurrencies are generated using a *mining* process. This process takes much time to generate a single coin and is expensive since the machine processing power is high. Energy tends to be expensive and high levels of processing power lead shorter hardware lifetime, 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.

This brings us to a recent cyber-crime trend: *cryptojacking* [4]–[6]. Cyber-criminals saw an opportunity here. If they could hack web pages and embed mining scripts without getting detected, they could capitalize on the millions of visits a day these pages get to obtain profits. Mining cryptocurrencies in web pages is not illegitimate per se. In fact, it is an alternative to ads. 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.

978-1-7281-8326-8/20/$31.00 ©2020 IEEE

We introduce a new *cryptojacking detection mechanism* based on the CPU usage of the visited web pages. Our detector monitors activity 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 web pages 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 periods, 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. We solve these two challenges 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 can pick a good combination of features to detect the phenomenon.

To configure the detector, we obtained an extensive collection of the CPU metrics measurements with mining algorithms running, with different loads and 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 malware cases that exploited browsers and used the GPU for mining, probably because the JavaScript APIs that allow it are recent and existing malware is often based on Coinhive [4], the pioneer on web mining as a substitute for ads.

In our study, we faced a third and unexpected challenge. We used a specific web search engine to find pages contaminated with cryptojacking scripts. However, some of these scripts never run, either because they only run in some browsers or because they have bugs. Either way, this made it harder to obtain ground truth to train and evaluate our detector.

Sections II and III explore the topics and research performed in the mentioned fields of study. Section IV presents our solution composition. Section V presents the experimental evaluation and Section VI concludes the paper.

II. Background

This section presents some background on our work.

A. Cryptocurrencies

Cryptocurrencies appeared intending to eliminate the need for third-party institutions on online money payments and transfers. As described by Nakamoto [1], the existing approaches are not a good fit for today's online conditions. Nakamoto's Bitcoin solved that problem but also paved the way for the appearance of many alternative solutions. Therefore, hundreds of slightly different new cryptocurrencies have been appearing[1], with the emphasis on different problems, from faster transactions to better anonymity or stability. This flourishing ecosystem has also lead to many research in different aspects of cryptocurrencies, from different networks [2], [3], [7]–[9] to attacks [10], [11], consensus algorithms [12], [13], and many other topics.

B. Machine Learning

Brownlee states that we can divide machine learning algorithms into two big groups: supervised and unsupervised [14]. Machine learning classifiers belong to the supervised learning category, where an input is given, and a specific output is expected. A classifier has several 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.

C. Intrusion Detection Systems

Wagner and Soto [15] mention that there are broadly speaking two kinds of intrusion detection systems: network intrusion detection systems and host-based detection systems. Our interest lies with host-based intrusion detection systems (HIDS) since they are the most adequate for what we aim with this research. HIDSs can be further divided into two categories: signature-based schemes and anomaly-based. Wagner and Soto claim that signature-based detection is easier to bypass than anomaly-based detection. However, web applications are very heterogeneous, so it does not seem practical to obtain a model of behavior that covers all of them. Therefore, we decided on a signature-based intrusion detection system, with signatures created automatically using machine learning.

D. CPU Monitoring

CPU monitoring is not new. Multiple CPU metrics can be provided to the user, either by the hardware firmware, by the operating system, or by APIs written for this purpose. In some cases the motherboard itself may provide more sensors than those in the CPU. 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 work on the highest number of systems possible without the need for modifications.

One of the most noticeable changes on 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 web site 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.

[1]https://coinmarketcap.com/

III. Related Work

Cryptojacking grew much in 2017, 8500% according to the 2018 Symantec security annual report [16], then went down and is now increasing again.

Rauchberger et al. [17] mention that the best ways to identify a browser mining a 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 legitimate web apps widely use it).

MiningHunter [17] is a framework to track mining scripts. It had an interesting concept in mind, where for every loaded web site, the metadata, all executed JavaScript, and raw Websockets traffic are 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 figure out which ones belong to the same mining campaigns.

Saad et al. [18] noticed that during cryptojacking script execution, the code would establish a WebSocket connection with a remote server and perform a bidirectional data transfer. 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.

Outguard [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 Web Workers' presence and the use of WebAssembly, but also events like *PostMessage Event Load* and *MessageLoop Event Load*. They also added the CPU usage 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.

Coinpolice [20] also uses multiple features to detect cryptojacking. 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 convolutional and recurrent neural networks), 4),5) two classifiers based on JavaScript/WebAssembly function

execution time series, and 6),7) two classifier based on the JavaScript/WebAssembly features and throttling-detection features. Expanding their CPU threshold to 30% allowed them to have a broader and more complete mining detection and introduce the possibility of a higher false positive rate if used on its own which was not the case.

A code analysis classifier that performed dynamic analysis of the opcodes [21] 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 basis that dynamic opcode tracing is extremely effective at detecting cryptojacking, as they obtained 99.9% accuracy.

SEISMIC detects cryptojacking based on semantic signatures of behavior [22]. Moreover, it uses the fact that its detection mechanism provides low false positives to dynamically block scripts flagged as executing cryptojacking.

Munoz et al. detect cryptojacking by inspecting network traffic [23]. The detection is based on monitoring NetFlow/IPFIX flows that summarize traffic. CryingJackpot is also based on traffic inspection but in combination with data from operating system logs, achieving a high F1-Score [24].

As Carreiro finds in his work [25], 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 higher temperatures on these components.

IV. The Intrusion Detection Mechanism

This section presents our intrusion detection mechanism, starting with an overview, then with details of the detector's modules.

A. Detector Overview

Our intrusion detection tool was designed to run inside virtual machines. For now, it is set up to check if a web page executes cryptojacking scripts. For that purpose, the detector is started by executing a Python script composed of four modules, represented in Figure 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 the Weka tool [26]. 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.

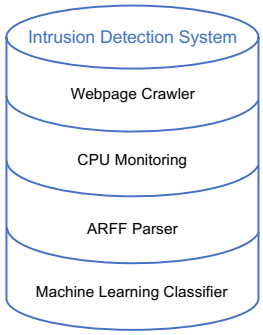

Fig. 1: Intrusion detector architecture

There were multiple steps to take into account to reach the previously mentioned solution, described next.

B. Webpage Crawler

The logical division we did was the following. First, there was the need to crawl the chosen pages; for each of the use cases (for training and testing), different lists would be supplied in both cases. In the first list (training set), each dataset's results would already be known, but in the second list (testing set), the results would not be known. The testing set would be a group of web pages accessible online. The crawling would be performed by a simple script that would go through each page individually and sequentially loading all the initial page resources. For the crawler, we chose to use a project publicly available on GitHub called Puppeteer-Cluster [27]. The project has a practical way of handling the top Alexa list [28], which is the list we used to perform the crawl, trying to find pages containing cryptojacking infected pages.

C. CPU Monitoring

Metrics are gathered by retrieving the CPU values for each web page 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 in 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). We ended up choosing the `mpstat` command line tool 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 helpful to some of our observations. The tool outputs several CPU metrics; it is customizable in terms of metrics

by core and by time interval at which the reading should be done.

`mpstat` allowed the following metrics to be extracted from the different CPU cores independently:

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

To make the metric gathering faster, we used Amazon EC2 instances, 4 to be precise. All only with one CPU core, with 512 MB RAM and 40 GB internal storage. `mpstat` had the options to read the CPU usage by core and give an average of all the cores; 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 overall average and not the average by CPU core.

Figure 2 shows the CPU average consumption while the page *sohu.com* is downloaded and running. That 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. CPU time shows multiple peaks and some reduction on average, except for peaks that continue to appear when a routine or script is executed. Figure 3 shows 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.

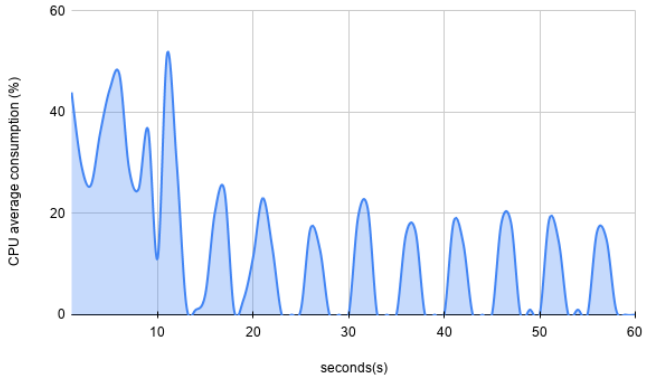

Fig. 2: CPU behavior while the sohu.com page is being loaded

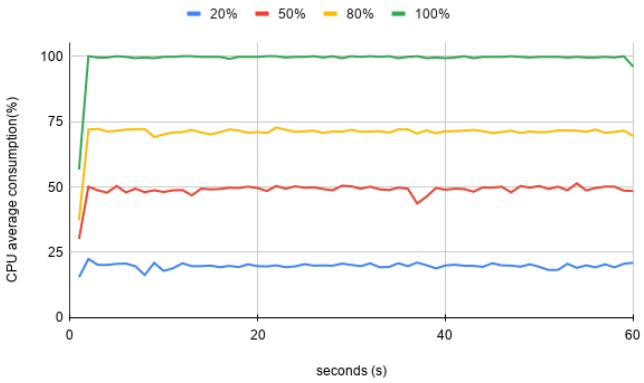

Fig. 3: CPU behavior while mining at different with different throttling

D. Arff Parser

The next step of the process was to develop a script that transforms the data stored on the individual files to 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. These files are divided into a header part that describes how the data is subdivided in 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; it 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 of strings instead of integers, but there was no advantage to this approach in our case.

E. Machine Learning Classifier

This is the last and one of the most critical stages. We used 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. 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.

We started by training our algorithm using k-fold cross-validation. The cross-validation method is better than others to our approach since it helps to have a better and more realistic

estimate on how the model is expected to behave against data not used in the training set.

During the experiments, we tested multiple classifiers to find those that provide 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 [29] uses a single decision tree to obtain propositional data. TLC represents regions with assigned attributes in their 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 to learn each region's influence on a bag's classification.
- *MISVM*, Multiple-Instance Support Vector Machine [30] 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 [31], 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 the feature vector components, meaning constructing trees in randomly chosen subspaces.
- *SMO*, Sequential Minimal Optimization [32], substitutes all the missing values in the dataset with binary values, normalizing them. It is a very scalable algorithm since it breaks Quadratic problems in small ones solvable analytically.

V. Experimental Evaluation

We explored our scheme by monitoring different numbers of cores for different time frames to understand how to configure our detector. We use the usual concepts of true positive (TP, cryptojacking well detected), false positive (FP, wrongly detected), true negative (TN, no cryptojacking not flagged), and false negative (FN, missed cryptojacking). We also use common metrics: TP rate, FP rate, precision, recall, f-measure, MCC (Matthews correlation coefficient), ROC area (or area under ROC curve), and PRC area (area under precision-recall curves).

A. Training Dataset Composition

We started gathering the training metrics by dividing them into different groups. 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 is 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 (60x4);
- *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 (60x4);
- *no one is working on the computer, and no miner is running* – This is the 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 described was essentially browsing different web pages, running some java programs, and some bash scripts.

In later runs, 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.

B. 1 CPU core for 15 seconds

We started by configuring our detector to monitor 1 core and 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. Evaluating the results, we obtained better result than anticipated for a first experiment, with an average precision of 92.5%. While not being a good result for a final solution, it suggested that it could be possible to have a working solution at the end of our trials with some improvements. 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 I 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 classify the supplied values correctly. The classifier was called (TLC). The first training values were gathered by loading the pages for 15 seconds on a 1 core machine.

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 web sites.[2] For this purpose, we initiated four EC2 instances, all from a single image. After running the first sample, we realized that some web sites were not responding or were dead. We ended up running the

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

TABLE I: Training results using the metrics for 15 seconds and the TLC algorithm

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

crawlers against 12500 web pages, where only 11489 ended up being useful to our work. 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 that we could match each result to each web page. This was also the approach for the rest of our experiments; there were two lists, one with CPU results and another with the crawled pages' URLs.

We found 1837 possible pages containing cryptojacking malware running on them, given the 11489 active crawled pages sample of 15.99% of the active sample. We found this unrealistic since it was a much higher number than those observed in all the related work. Therefore, we decided to analyze a small sample of the pages manually. 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 change the way we gathered the metrics. We decided to extend the metric gathering stage and to extend it for 60 seconds, instead of 15 seconds (next section).

C. 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, a higher number of values could be used to trace a pattern and then apply it to unknown pages. Table II shows the results we obtained.

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.

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 significant 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 these pages sample, we excluded 3 since they did not yield a high CPU usage; we did not understand 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 several reasons why we would not find it at the time of result evaluation, the top two being: (i) (i) the miner was removed before the evaluation; (ii) 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 number 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. These pages indeed had a high CPU usage. We started by loading the page and understanding what it was doing. The first page was an online game web page doing computation on the client-side. It is usual for such web pages to have high CPU usage and stable if not many interactions are 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 webserver 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 disappointing; it showed high CPU usage indeed, but it was just an image modeling changing as time passed and even simpler scripts, again not running what we thought it was.

D. Average of the cores for 60 seconds

At this stage, we realized that there should be some CPU metrics of web pages 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 the test with the set of existing algorithms. Table III shows which algorithms better fit our models. The names of the algorithms are abbreviated in the table. They are: *(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

TABLE II: Training results using the metrics for 60 seconds and the TLC algorithm

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 III: Training precision (%) results for the average of the cores using the metrics for 60 seconds

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

of false negatives. Table IV shows an in-depth analysis of the MISVM algorithm with the training set provided to it.

Something remaining to do was to test against the Alexa top pages. After evaluating the crawled web page values with the new algorithm, we ended up with 851 pages, with a positive result (presence of a cryptocurrency miner). Some of the previously flagged pages were dropped, which was a good sign. We chose once again the 10 first pages flagged to get evaluated.

All 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 to find the difference between them and those running miners.

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 malicious CPU power usage?

E. 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. A small sample of metrics was gathered in a local virtual machine, a Linux machine (Kali Linux distribution) with 2 CPU virtual cores. 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 rerecorded using more capable EC2 instances. Therefore, 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 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.

All the metrics were retrieved, parsed and compiled inside a single arff file, with relational bags. Table V shows the precision of the algorithm with our new training data. The number of false positives was once again calculated, so we could find out if this new model would be a good fit (see Table VI).

After observing the previously mentioned tables with the same set of algorithms we used in the previous runs, we had MISVM as the best choice for the mining detector. By evaluating the percentage of correct results, 98.82%, and the average percentage of false positives of 2%, we realized it is still a good result. Although not critical for our evaluation, the false positive rate 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.

F. 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 (say 4) does not use all the cores at full capacity or at the same level. Even when running pages that load a higher number of scripts, or scripts that are more CPU intensive. However, once the CPU starts running a miner, *the values across all the cores become closer*. Even if one of the cores goes down for a second, another core takes its place. *The average of the processors is approximately constant, something that does not happen on a page without*

TABLE IV: Training results for the average of the cores for 60 seconds

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

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 VI: Training results for the false positives number collecting the metrics for 60 seconds and 2 CPU cores

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

a miner. In the web pages without miner, even when CPU consumption is high, the percentage of each core being used is not constant. The CPU consumption value by core of mining pages only varies by 1 or 2 percent among them. With this in mind, we ended up taking the last set of metrics using a local virtual machine with 4 CPU cores, to validate these observations.

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 while also maintaining the different CPU mining loads. There was also an error that led to a decrease in the amount of training data supplied to the classifier, reducing them by roughly half. Values with entropy added were removed, and the training data present was composed of a higher number of examples with cryptojacking present.

At this point two different solutions were considered. The first solution involved a legacy arff file without relational bags but with carefully chosen values that matter the most. The second did not use any machine learning algorithm, but involved looking at the averages and deviations of the values to try to assess if there were high deviations 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 we are not running anything else on the computer that uses the CPU heavily, actually, this condition was always a problem since it would possibly produce erroneous results in all the solutions we developed until now.

Our custom-made solution developed 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 cryptominers at a 100% load, which are those that should never be missed. 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 had many close results with little to no difference, but the one that got the better result was meta.RandomSubSpace. Table VII shows the values we got out of the training, to this specific algorithm.

Analyzing the values, we got 99.7% 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 are expensive) 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 there was anything 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 ruled out since the CPU results were not close enough to signal a potential miner. We did not understand 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*. The value was still under 20% and was not very steady, indicating that the page might just be loading heavier scripts. Or possibly the mining script (if present) was not loaded while performing the manual analysis.

The 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

TABLE VII: Training results using the metrics for 60 seconds and 4 cores

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

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 analyzing the values we got from the training results table, we decided to use the second best algorithm of the previously mentioned selection, the algorithm SMO.

The results of SMO are slightly worse than those of RandomSubSpace. However, the difference is barely noticeable. 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 may be possible some pages 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.

VI. Discussion and Conclusion

We have shown that by combining a set of CPU metrics it is possible to detect browser-based cryptojacking with high precision. However, it is important to mention that these results were extrapolated from a classifier evaluated in a laboratory environment, which may influence the values when evaluated in a real-world scenario. Some other conclusions were:

- Running cryptojacking shows a discernible pattern on CPUs with multiple cores.
- A set of machine learning classifiers can detect the mentioned pattern with high precision.
- On machines running only a detector, if the page was running a miner we could see the average values of the CPU to stabilize and showing homogeneous values across CPU cores.
- Detection performs worse on a machine with multiple other programs consuming CPU.

Our research may have a broader application than browser-based miners. We base our work on CPU metrics and patterns, which are equally influenced by miners not running in the browser. However, we leave the evaluation of our approach with non-browser based cryptojacking malware as future work.

Acknowledgements

This research was supported by national funds through Fundação para a Ciência e Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID).

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.
[2] F. Tschorsch and B. Scheuermann, "Bitcoin and beyond: A technical survey on decentralized digital currencies," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 2084–2123, 2016.
[3] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, "SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 104–121.
[4] S. Eskandari, A. Leoutsarakos, T. Mursch, and J. Clark, "A first look at browser-based cryptojacking," in *Proceedings of the 3rd IEEE European Symposium on Security and Privacy Workshops*, Jul. 2018, pp. 58–66.
[5] H. L. J. Bijmans, T. M. Booij, and C. Doerr, "Inadvertently making cyber criminals rich: A comprehensive study of cryptojacking campaigns at internet scale," *Proceedings of the 28th USENIX Security Symposium*, pp. 1627–1644, 2019.
[6] S. Pastrana and G. Suarez-Tangil, "A first look at the crypto-mining malware ecosystem: A decade of unrestricted wealth," *arXiv preprint arXiv:1901.00846*, 2019.
[7] S. King, "Primecoin: Cryptocurrency with prime number proof-of-work," 2013.
[8] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine agreements for cryptocurrencies," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, 2017, pp. 51–68.
[9] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger Fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the 13th ACM EuroSys Conference*, 2018.
[10] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," in *International Conference on Financial Cryptography and Data Security*, 2014, pp. 436–454.
[11] M. Apostolaki, A. Zohar, and L. Vanbever, "Hijacking Bitcoin: Routing attacks on cryptocurrencies," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, 2017, pp. 375–392.
[12] T. Crain, V. Gramoli, M. Larrea, and M. Raynal, "Blockchain consensus," in *ALGOTEL 2017-19èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, 2017.
[13] M. Correia, "From byzantine consensus to blockchain consensus," in *Essentials of Blockchain Technology*. CRC Press, 2019, ch. 3.

TABLE VIII: Training results using the metrics for 60 seconds and 4 cores

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

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

[15] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002, pp. 255–264.

[16] Symantec, "Symantec cryptojacking growth 2018 annual security report," https://resource.elq.symantec.com/LP=5840?cid=70138000000rm1eAAA, 2018.

[17] J. Rauchberger, S. Schrittwieser, T. Dam, R. Luh, D. Buhov, G. Pötzelsberger, and H. 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.

[18] M. Saad, A. Khormali, and A. Mohaisen, "End-to-end analysis of in-browser cryptojacking," *arXiv preprint arXiv:1809.02152*, 2018.

[19] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, "Outguard: Detecting in-browser covert cryptocurrency mining in the wild," in *The World Wide Web Conference*, 2019, pp. 840–852.

[20] I. Petrov, L. Invernizzi, and E. Bursztein, "CoinPolice: Detecting hidden cryptojacking attacks with neural networks," *arXiv preprint arXiv:2006.10861*, 2020.

[21] D. Carlin, P. O'kane, S. Sezer, and J. Burgess, "Detecting cryptomining using dynamic analysis," in *16th IEEE Annual Conference on Privacy, Security and Trust*, 2018, pp. 1–6.

[22] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "Seismic: Secure in-lined script monitors for interrupting cryptojacks," in *European Symposium on Research in Computer Security*, 2018, pp. 122–142.

[23] J. Z. i. Munoz, J. Suarez-Varela, and P. Barlet-Ros, "Detecting cryptocurrency miners with NetFlow/IPFIX network measurements," in *IEEE International Symposium on Measurements & Networking*, Jul. 2019, pp. 1–6.

[24] G. Gomes, L. Dias, and M. Correia, "CryingJackpot: Network flows and performance counters against cryptojacking," in *Proceedings of the 19th IEEE International Symposium on Network Computing and Applications*, 2020.

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

[26] 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*, vol. 11, no. 1, pp. 10–18, 2009.

[27] T. Dondorf, "puppeteer-cluster," https://github.com/thomasdondorf/puppeteer-cluster.git, 2020.

[28] "Alexa top sites," https://www.alexa.com/topsites, accessed: 2020-08-06.

[29] N. Weidmann, E. Frank, and B. Pfahringe, "A two-level learning method for generalized multi-instance problems," *14th European Conference on Machine Learning*, 2003.

[30] S. Andrews, I. Tsochantaridis, and T. Hofmann, "Support vector machines for multiple-instance learning," in *Advances in neural information processing systems*, 2003, pp. 577–584.

[31] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, pp. 832–844, 1998.

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