RAPID: Resource and API-Based Detection Against In-Browser Miners

Juan D. Parra Rodriguez
University of Passau
Passau, Germany
dp@sec.uni-passau.de

Joachim Posegga
University of Passau
Passau, Germany
jp@sec.uni-passau.de

ABSTRACT

Direct access to the system's resources such as the GPU, persistent storage and networking has enabled in-browser crypto-mining. Thus, there has been a massive response by rogue actors who abuse browsers for mining without the user's consent. This trend has grown steadily for the last months until this practice, i.e., CryptoJacking, has been acknowledged as the number one security threat by several antivirus companies.

Considering this, and the fact that these attacks do not behave as JavaScript malware or other Web attacks, we propose and evaluate several approaches to detect in-browser mining. To this end, we collect information from the top 330.500 Alexa sites. Mainly, we used real-life browsers to visit sites while monitoring resource-related API calls and the browser's resource consumption, e.g., CPU.

Our detection mechanisms are based on dynamic monitoring, so they are resistant to JavaScript obfuscation. Furthermore, our detection techniques can generalize well and classify previously unseen samples with up to 99.99% precision and recall for the benign class and up to 96% precision and recall for the mining class. These results demonstrate the applicability of detection mechanisms as a server-side approach, e.g., to support the enhancement of existing blacklists.

Last but not least, we evaluated the feasibility of deploying prototypical implementations of some detection mechanisms directly on the browser. Specifically, we measured the impact of in-browser API monitoring on page-loading time and performed micro-benchmarks for the execution of some classifiers directly within the browser. In this regard, we ascertain that, even though there are engineering challenges to overcome, it is feasible and beneficial for users to bring the mining detection to the browser.

CCS CONCEPTS

• **Security and privacy → Browser security**; *Web protocol security*; *Web application security*;

KEYWORDS

Web Security, CryptoJacking, HTML5, Browser Abuse

ACSAC '18, December 3–7, 2018, San Juan, PR, USA

ACM ISBN 978-1-4503-6569-7/18/12.
https://doi.org/10.1145/3274694.3274735

ACM Reference Format:
Juan D. Parra Rodriguez and Joachim Posegga. 2018. RAPID: Resource and API-Based Detection Against In-Browser Miners. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3274694.3274735

1 MOTIVATION

HTML5 technologies such as WebGL and WebSockets contributed to achieve something that was attempted but not successful so far: in-browser crypto-currency mining [1]. On the one hand, crypto-mining could be a business model where visitors use energy and computational resources in exchange for the content they get, instead of coping with advertisements and their privacy implications [14, 16, 29, 47]. However, system administrators, developers, and attackers have started embedding crypto-mining scripts through their own and third-party sites without the users' consent.

Abusing the browser's resources for crypto-mining without the user's knowledge, i.e., CryptoJacking, has been already recognized by the ESET antivirus company as the top cyber attack today [17]. Further, Symantec states that browser-based CryptoJacking has increased by 8.500% in 2017 in their technical report [43]. From a practical perspective, there is no evidence of a transparent environment where users provide consent before sites start using their resources for mining. As a result, we propose to provision the browser with tools to detect on its own, without relying on information provided by the site visited, whether the site being rendered is performing mining or not.

Our **contributions** can be summarized as follows: *1)* we propose several novel mining detection techniques based on the system's resource consumption and resource-related browser API usage. *2)* we evaluate and compare the performance of the different proposed techniques by collecting information about how sites use the browser most resource-related APIs and system's resources. For this, we used real-life browsers on 330.500 sites based on the Alexa top ranking. *3)* we analyze the feasibility of porting API-based mining detection to the browser. To this end, we perform a quantitative analysis of the overhead on the page loading time and execute micro-benchmarks for the detection algorithm.

This paper is structured as follows: Section 2 states our problem. Sections 3 and 4 describe our methodology and evaluation. Then, we cover related work in Section 5 and draw conclusions from our work and discuss feature steps in Section 6.

2 PROBLEM STATEMENT

CryptoJacking has some critical differences with malware and other Web attacks. First of all, JavaScript malware, as well as attacks against the users' sessions and data, are executed once.

Once the attacker has obtained the data, session or control over the machine, the attack is successful and complete. For mining, an attacker benefits over longer periods of time because his earnings are proportional to the resources he can use on the browser's side. Second, conventional Web security mechanisms, such as the Same Origin Policy or Content Security Policies (CSP) cannot protect users against mining because in many cases the code is delivered as part of the frame or site rendered by the browser. Thousands of sites have been used to deliver mining scripts in different ways without signs of cross-site scripting or other Web attacks. For instance, Youtube advertisements [25, 46], Content Management System widgets [26], or even wrong permissions assigned to Amazon S3 buckets [28] have been used by attackers to modify sites and inject mining scripts.

So far, two possible countermeasures against mining have been implemented or discussed: blacklisting domains or blocking sites based on their resource consumption. However, **existing countermeasures have two problems**. Blacklists are easy to circumvent by moving miners to new domains[1]; thus, they yield too many false negatives. Second, there is a bug description asking Chromium developers to block sites with CPU usage; however, most developers consider it to be an unreliable indicator for mining because it can block many benign sites and create many false positives [13].

We aim to close this gap by providing a better detection mechanism that can achieve two goals. On the one hand, the detection mechanism should label miners on sites that were not previously seen, i.e., reduce false negatives of blacklists. On the other hand, it should not label sites as miners unless they are performing mining, i.e., decrease false positives with respect to a decision only considering processing power. To this end, we performed a thorough evaluation between a detection mechanism based on network, processor, and memory consumption and a more advanced approach based on monitoring resource-related API calls inside the browser.

The next aspect related to CryptoJacking is to assess when to perform **online or offline detection**. We refer to *online detection* when detection is performed on the same browser where sites are being browsed by users; conversely, we term *offline detection* when malicious or benign sites are classified in a different system than the one employed by the user, e.g., in a cloud infrastructure and then delivering a blacklist to the user's browser. In the case of offline detection, servers are constantly crawling the internet and could come across different content than what is served to actual users for two reasons: 1) the server could get a customized version of the content without the mining, 2) pages serving malicious content could be behind log-in screens, etc. Despite these issues, offline detection has been incredibly useful to train and verify results from more lightweight (online) mechanisms in the past; for example, Cujo [38] and ZOZZLE [9] were trained and verified, respectively, using JSAND [8]: an offline malicious JavaScript detection mechanism. Another advantage of offline mechanisms is that they can have more computing power and more specialized setups which translates into better detection performance.

Every detection method we analyze can be deployed offline. In particular, the resource-based approach can only be performed offline because it executes a browser in an isolated environment during a single site visit. Contrarily, the resource-related in-browser API classifier has more potential to be deployed directly in the user's browser, i.e., for online detection. Thus, we measure the impact on page-loading time imposed by the API monitoring and the computation of the feature vector.

[1] [6] showed that CryptoJacking actors started hosting their own proxy servers to connect to mining pools [49] to avoid fees for Monero and to make detection harder to perform as domain blacklisting is not sufficient anymore.

3 METHODOLOGY

We started by performing a large-scale collection of how sites use system's resources particular API calls. Afterward, we labeled the data with two classes (mining, or benign sites). Then, we chose different sets of features as well as a learning model to train a classifier predicting whether a site is performing mining or not.

3.1 Data Collection

There have been several approaches to automate browsers for data collection, e.g., OpenWPM created by Englehardt and Narayanan [14], yet we decided to implement our own mechanisms for various reasons. We needed to control and instrument the browser remotely to receive the events created by our instrumentation through a highly efficient communication channel. Based on these requirements, we decided to use the `Chrome Debugging Protocol` with Chromium[2].

In addition to collecting which API calls have been performed by Websites, we faced the challenge of quantifying the resources used by each browser during the experiments. To tackle this, we used `docker containers` to isolate the browser processes and count the amount of memory, processor, and networking used. Additionally, through the docker API, we were able to create, stop and delete containers for each visit. This ensures that the initial state is always the same, i.e., no cookies, local storage or cached sites, and the reproducibility of the experiments.

Figure 1 shows the logical architecture of our crawling system. A Crawler Node selects a pending job (site visit) from the database, creates a new chromium container and starts listening for memory, CPU, and network consumption events through docker APIs. Then, each node instructs one browser to instrument all pages loaded and to visit a particular page. During the visit, which lasts 35 seconds, the node receives all events detected by the instrumented code from the browser through the Chromium debugging protocol. Finally, it stops the tracing and waits until all data has been received before stopping the resource monitoring for the container and then removes the container. Additionally, there is a shared MongoDB database holding all pending jobs and results obtained so far. The database was designed to ensure that several nodes can be executed in parallel without conflicting with each other. We used one Virtual Machine (VM) with 700 GB of disk to host the database and four VMs with 15 GB of disk for crawling. All VMS were executed inside a single physical host running a VMWare ESX Virtualization server and had eight virtual processors and 8 GB of RAM. Moreover, each crawling VM had seven nodes (each node controlling one browser). To run Chromium on headless mode, we used the X virtual frame buffer tool (Xvfb) to execute it without requiring a screen for the servers visiting the sites.

[2] A key advantage is that Chromium enables us to leverage the internal network and profiling protocol to send events from the browser to the monitoring program by calling `console.timeStamp()` within the instrumented code.

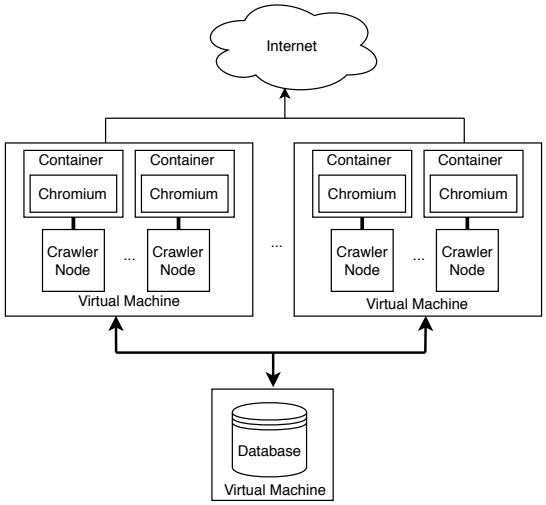

Figure 1: Crawler's Logical Architecture

Resource	Monitored API
CPU	`WebWorkers` (creation) `WebAssembly` (creation)
GPU	`WebGLRenderingContext` (all functions) `CanvasRenderingContext2D` (all functions)
Storage	`LocalStorage` (set, get and remove) `IDBObjectStore` (all functions)
Networking	`WebSocket` (creation, send and receive) `RTCDataChannel`-WebRTC (all functions)
Inter-Window	`Window` (postMessage and onMessage) `WebWorkers` (postMessage and onMessage) `SharedWorker` (postMessage and onMessage) `EventSource` (creation and receive)

Table 1: APIs monitored

Overall, we obtained 285.919 sites out of the top 330.500 Alexa sites. We tried to re-execute sites that failed due to network timeouts or operational errors after the first run was finished. The main operational error causing crashes during our experiments was non-deterministic. The problem was that Chromium stopped sometimes when Xvfb was used [33, 34]. From the 285.919 sites that were adequately visited, we found 656 sites performing mining using the labeling technique described in Section 3.2.

System's Resource Monitoring: To minimize measurement errors, we monitor all the resources used by the container running (only) the Chromium instance that opened a site through the docker stats API. In particular, our Crawler Node obtains events including memory, processor, and network consumption. Afterward, we aggregate the total number of ticks used by the processor, the total number of bytes used from main memory and the total number of bytes sent or received over the network separately.

JavaScript APIs: The computational resources accessible through the browser are CPU, GPU, storage, and networking. So, we override the browser's behaviour by instrumenting the APIs using those resources. In addition to APIs accessing the resources directly, we have also monitored other APIs enabling inter-window communication because they also reflect interactions between different domains and are commonly used by WebWorkers, etc. Nonetheless, we tried to instrument the least amount of APIs, yet still obtaining an overview of the resources being accessed through the browser (see Table 1).

To monitor the APIs within the browser remotely, we used the Chrome Debugging Protocol and let the Crawler Node instrument the browser remotely using the `addScriptToEvaluateOnLoad` and `setAutoAttachToCreatedPages` calls from the `Page` API. This injects scripts when a new document is created. Despite what the term addScriptToEvaluateOnLoad suggests, this function injects the code before the site is loaded [21]. To receive events through the Chrome Debugging Protocol, we used the `Tracing.dataCollected` function to get events generated by Chromium's console whenever the function `console.timesTamp` was called from the instrumented code.

To instrument the browser we followed an approach attempting to introduce the least performance impact possible. That is to say, whenever it was possible, we added a listener to the event without redefining the API. For instance, this is possible when our script needs to get the postMessage events from the window by adding an EventListener to the window for the event "message". In other cases, when we only needed to monitor a few specific calls, we redefined the functions. Whenever, there were APIs intensively used by mining scripts, such as the GPU-related scripts, we opted for redefining all the functions and notifying the Crawler Node through the Chrome Debugging Protocol whenever any function was called on the object. To redefine methods we used the method `Object.defineProperty`, as it has been previously used by IceShield [19] or the OpenWPM Firefox extension instrumenting JavaScript [15].

As the database logged the time for each function call, the result can be seen, from the browser's point of view, as a list of events observed during the execution of a site.

3.2 Labeling

We performed the data collection from the top 330.500 Alexa sites between October the 2^{nd} and November the 8^{th} 2017; this time window was very convenient. On the one hand, this collection started shortly, i.e., less than 3 weeks, after Coinhive started operating (the number one threat currently according to ESET [17]). In particular, this time was enough to evidence the overwhelming adoption of Coinhive described in their first-week report [44]. On the other hand, data collection also took place before miners started using their own domains to avoid detection and avoid paying fees to Coinhive [6]. This evasion technique was rarely used since late November 2017 and only became a trend towards March 2018. The lack of evasion techniques allowed us to obtain high-quality data for labeling our learning dataset.

To label the mining and benign classes, we queried our database to find sites who had sent, received messages or opened WebSockets with any domain that had been blacklisted by the existing miner

blockers [22, 50]. A detailed list of the domains used is shown in Table 7 (Appendix B).

Querying the database after dynamic monitoring took place, i.e., WebSocket creations and sending or receiving events, gave us a significant advantage over static techniques. In particular, we obtained proportionally more miner samples by analyzing the site's behavior during runtime in comparison to other work analyzing the evolution of in-browser miners, which executed queries on the static content provided by Websites over time [18][3].

3.3 Feature Selection

At this point, we have described how we obtained two data sources: resources consumed (memory, network, and processor), and JavaScript API Events (a sequence of resource-related events that occurred for each visit). However, before we can describe which features will be used, we have to ensure fairness when making the comparison between both datasets. Although we collected 285.919, only 196.724 actually used resource-related APIs. Thus, to be fair, we used only the 196.724 visits that used resource-related APIs to obtain features from both datasets (resources and API calls). This cannot leave any mining site out of the dataset because we labeled mining sites based on the WebSocket messages they exchanged with a known mining server: a resource-related API call.

System's Resource Monitoring: System's resources could only be quantified as a total amount of processor ticks, bytes sent or received over the network or number of bytes of memory used along the execution, i.e., a vector in $\mathbb{N}^4$.

In the case of system's resources, not all features have the same scale; for instance, processor ticks is likely to have higher values than the number of bytes sent over the network. To improve performance of the learning algorithm we had to perform **scaling** of each vector component to a fixed range for all features. However, scaling can be harmful when outliers are an essential aspect of the problem (uncommon sites doing mining). So, we used a robust scaling preprocessing algorithm from Scikit-learn to map vectors to $\mathbb{R}^4$ while affecting outliers the least possible. We have witnessed an improved performance after scaling.

Nonetheless, the fact that we have to perform scaling of each feature in comparison to all other samples observed brings a particular downside. The scale for each feature, e.g., min and max number of processor ticks, is fixed after scaling; as a result, scales depend on each computer executing the prediction, e.g., big desktop PCs will observe different ranges than phones in practice. So, online deployment, as described in Section 2, is not an option. Instead, this classifier could be supported to enhance existing blacklists.

JavaScript APIs: This section describes how feature vectors are created step by step. First, we explain how we extract relevant values for each vector component. Then, we describe the vector's normalization process before they are used for training and classification. Lastly, we describe the technical aspects considered for efficiency during the learning and prediction process.

In the case of the resource-related API events, we face a challenge: mapping the sequence of events for every visit as a set of numerical features. We explore two approaches to map events to vectors: bag of words, and q-grams. Luckily, Rieck et al. already introduced notation to describe q-grams in Cujo [38]. The same q-gram representation and set of features were later used by EarlyBird by Schutt et al. [40]. Thus, we will use the same notation used by EarlyBird and Cujo to show the bag of words and q-grams approach.

Bag of words: let E be the set of all possible events and let ϕ^b map x to the vector space $\mathbb{N}^{|E|}$ representing the bag of words. In particular, we associate every API call $e \in E$ to one dimension in the vector. So, we can define ϕ^b using every API call to create the vector like this:

$$\phi^b : x \to (\phi^b_e(x))_{e \in E}$$

where the $\phi^b_e(x)$ counts the number of times that e appears in x. We observed 173 resource-related API calls being performed throughout our crawl, so $|E| = 173$.

Q-grams: let S be the set of all possible q-grams and let ϕ^q be a function mapping x to the vector space $\mathbb{B}^{|S|}$. Now, like Rieck et al. [38], we map every possible q-gram to a component of the vector and using the function:

$$\phi^q : x \to (\phi^q_s(x))_{s \in S}$$

$$\text{where } \phi^q_s(x) = \begin{cases} 1, & \text{if } x \text{ contains the q-gram } s \\ 0, & \text{otherwise.} \end{cases}$$

Vectors produced by ϕ^b and ϕ^q have the same scale for every feature, i.e., they are all counting events or q-grams; therefore, unlike like the resource-based approach, this approach does not need scaling. API-based approaches are likely to perform better than the resource-based model on systems with different computing power because they focus on particular patterns observed on the sequence of API calls, rather than using the actual amount of resources consumed.

Notwithstanding, we cannot use vectors produced by ϕ^b and ϕ^q as the feature vectors yet. The main problem is that, so far, this feature vector would give more weight to event sequences x containing more q-grams. Thus, we **normalized** vectors produced by ϕ^b and ϕ^q using the ℓ^2 (or also known as Euclidean) norm. After normalization, the bag of word vectors are mapped from $\mathbb{N}^{|E|}$ to $\mathbb{R}^{|E|}$ and q-grams vectors are mapped from $\mathbb{B}^{|S|}$ to $\mathbb{R}^{|S|}$; furthermore, every vector has the same norm after normalization. As a result, the value represented in a single q-gram dimension will be diminished when sites contain many other q-grams. On the contrary, for an event sequence with very few q-grams, each one of them will have more weight. An important advantage of normalization over scaling is that normalization is applied only considering a single sample, i.e., no need to know the distribution of the data beforehand.

We must clarify that even though Cujo [38] and EarlyBird [40] use the same q-gram representation, our approach uses a different set of features oriented to resource-related APIs. More details on similarities and differences are discussed in Section 5.

The size of all possible q-grams grows exponentially with q; theoretically, $|S|$ has an upper bound of $|E|^q$. However, we found that in the case of APIs used by sites the combinations observed on the q-grams is considerably smaller as q grows, i.e., $|S|$ was 3.698, 17.604, 48.657, 103.034 for q values of 2, 3, 4 and 5 respectively.

[3] [18] reports circa 1.000 mining sites on the Alexa top million. We discovered 656 sites within the top 330.500 sites.

Still, due to the q-gram's high dimensionality and the fact that they would have many components of each vector with null values, we used a sparse "triplet" representation for the matrix continuing all the vectors in Scipy [30]. Thus, the k^{th} elements in the arrays i,j,v represent that $matrix[i_k, j_k] = v_k$.

3.4 Learning

Even though there are several classification algorithms such as linear discriminant analysis, logistic regression, etc., Support Vector Machines (SVM) are prevalent for tackling security problems because, unlike other classification methods that minimize the empirical loss, SVM attempts to minimize the generalization loss. To do this, the SVM draws a separator hyperplane in an n-dimensional space and tries to find the separator that is the farthest away from known samples [39]. This maximizes space around the separator for new samples that have not yet been observed during training.

Another vital point visible by looking at the results of the labeling process is that we are facing a problem with highly unbalanced classes (656 malicious sites vs. 285.858 benign sites). From this point of view, we weighted the vectors provided during training using the `balanced` class weight Scikit-learn [32]. Thus, every sample weight is inversely proportional to the class frequency, i.e., mining samples have more weight as their class is considerably smaller.

Lastly, we use a linear kernel on the SVM. This helps in two aspects: runtime performance and the kernel size is bound by the number of dimensions of the vectors. Some other kernels, e.g., radial base function, grow with the number of samples used for training. The bigger the kernel is, the less feasible will be to execute the SVM on the browser after training it with tens of thousands of sites.

4 EVALUATION

We perform two kinds of quantitative evaluation on the proposed detection approaches. First of all, we analyze the performance of the detection algorithm by measuring its precision and recall. Then, we assess the impact on page-loading time imposed by the API monitoring and the effort required to calculate q-grams while Websites are loading. Last but not least, we measure the average time required by an SVM to classify a website within the browser.

4.1 Detection

A common way to chose parameters for a particular machine learning model is to use cross-validation. Although this is fine to tune arguments and to train the model, performance metrics calculated during cross-validation, e.g., precision, should not be used to evaluate the model. When the metrics calculated by cross-validation are used, the model is evaluated with data already used to tune the parameters; as a result, some information from the training data can be implicitly included in the model which results in over-fitting and a false increase on the metric [2].

To avoid this, we do not use the metrics calculated during cross-validation. As Figure 2 shows, we do a random split on the dataset between a training and an evaluation partition. Then, we perform cross-validation with K-fold ($k = 4$), again splitting the set randomly, to train the model and find the best classifier. Once the best classifier is obtained by cross-validation, we apply it on the evaluation dataset.

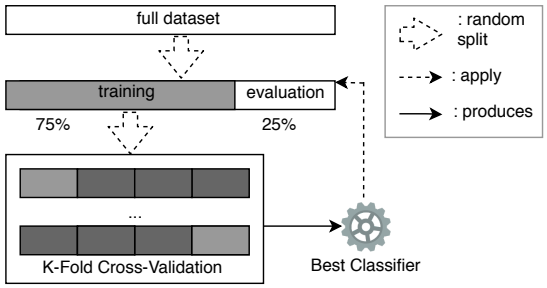

Figure 2: One Iteration Evaluating the Detection

Following the process described in Figure 2 guarantees us to obtain real performance, i.e., precision and recall, because they are calculated by applying the best found classifier on data which has never been used in the process so far. Also, this ensures that the classifier can generalize well when classifying samples it has never seen. In other words, **during the evaluation process the classifier judges sites it has never seen before, not even during the training phase**. Further, to have certainty that high performance is not due to a "lucky" random split, we execute the procedure shown in Figure 2 five times, for each set of features (resources, bag of words, and all q-grams), and calculate the mean and standard deviation.

Metrics: Machine learning algorithms are sometimes evaluated based on their accuracy, i.e., right predictions/total predictions; however, this metric does not consider for which class the prediction took place. Thus, wrong classifiers can still have high accuracy on unbalanced datasets; for instance, a wrong classifier predicting every site as benign would have high accuracy because most of the sites in the dataset belong to the benign class in the first place. To avoid this, we apply the classifier to the full evaluation set and calculate precision and recall[4] on a per-class basis.

By using these two metrics on a per-class basis, we obtain complementary views with respect to how elements are labeled for a specific class. **Recall** for a single class shows whether the classifier can detect all relevant samples. **Precision** ensures that the classifier does not label elements that do not belong to the class. In simpler terms, only an optimal classifier can have high precision and recall. A classifier detecting most of the elements for the class obtains high recall; nonetheless, if the classifier labels too many elements, its precision will drop, e.g., blocking sites only due to high processor consumption. Conversely, a classifier only labeling few elements in the class, to ensure that they are indeed there, has high precision; however, it is more likely that several elements in the class remain undetected, so its recall will drop, e.g., blocking sites using blacklists. Last but not least, we also calculate the f1 score[5] to have a single metric balancing precision and recall at the same time.

[4] $precision = \frac{|TP|}{|TP| + |FP|}$ and $recall = \frac{|TP|}{|TP| + |FN|}$ where TP is a set containing all true positives, FN contains all false negatives, and FP contains all false negatives.

[5] $f1 = 2 * \frac{precision * recall}{precision + recall}$

Classif.	Precision	Recall	F1
resources	99.98 (0.00)	97.97 (0.12)	98.97 (0.06)
b. words html5	99.98 (0.00)	98.16 (0.46)	99.06 (0.23)
2grams html5	99.99 (0.00)	99.99 (0.00)	99.99 (0.00)
3grams html5	99.98 (0.01)	99.99 (0.00)	99.99 (0.00)
4grams html5	99.99 (0.00)	99.99 (0.00)	99.99 (0.00)
5grams html5	99.99 (0.00)	99.99 (0.00)	99.99 (0.00)

Table 2: Mean and Std. Dev. for Metrics (Benign)

Classif.	Precision	Recall	F1
resources	13.32 (0.98)	93.89 (1.34)	23.31 (1.54)
b. words html5	15.20 (3.74)	93.52 (1.62)	25.95 (5.48)
2grams html5	96.57 (1.06)	96.98 (0.96)	96.77 (0.70)
3grams html5	96.79 (1.34)	95.33 (1.91)	96.04 (0.86)
4grams html5	96.47 (1.07)	97.84 (1.29)	97.15 (1.04)
5grams html5	96.54 (1.15)	95.48 (1.38)	96.00 (0.76)

Table 3: Mean and Std. Dev. for Metrics (Mining)

Benign Class: Table 2 shows the mean and standard deviations for precision, recall and the f1 score for each classifier. From 10.000 benign sites, all q-gram classifiers detect 99.99 of them, i.e., 99.99% recall. Q-grams obtained a similar result regarding precision; specifically, 2-, 4- and 5-grams produce one false positive for every 10.000 sites predicted as benign (99.99% precision). Also, the minimal standard deviation, below 0.01%, shows that we can be very confident of these results. Moreover, in the case of 3-grams, which seem to have a lower precision than the others, they have a higher standard deviation. This indicates that they could either have a value between 99.97% and 99.99% as precision. For the bag of words and resource-based approaches, the classifiers still obtain very high precision and accuracy, i.e., they only differ by 0.01% and 1.83% in comparison to q-grams. Considering that the benign class is the biggest class, results presented in Table 2 show that most of the samples are correctly classified.

Mining Class: Figure 3 shows the mean and standard deviation for precision and recall; further, Table 3 contains the same metrics along with the f1 score. From the recall point of view, all classifiers perform over 93%. In other words, *regardless of the classifier type, all the proposed detection mechanisms can detect at least 93% of all mining sites.* All classifiers based on q-gram have recall between 95% and 97%. From the precision point of view, q-grams perform really well, e.g., *from 100 sites labeled as mining by a q-gram classifier there are between three and four false positives.* Now, we address the relatively high deviation of all the precision, and recall values for the mining class, in comparison with the low deviation shown for the benign class. Further, we also address why there is a gap between the group containing the resources- and bag of words-based classifiers in comparison to all q-grams. Especially, because the high precision drop for the mining class was not observed when analyzing precision and recall values for the complementary class, i.e., benign sites, presented before.

In the presence of two complementary and significantly unbalanced classes, false positives and false negatives have a higher impact on precision and recall for the smaller class. For the sake of the argument let us assume we have a mining class with ten sites and a benign class with 10 million sites. Let us further assume we have a classifier detecting all ten mining sites, but also producing ten false positives. In such a scenario, the classifier would only have 50% precision for the mining class and 100% recall. Apparently, as we only looked at the mining class, with ten sites, the previously mentioned classifier has very poor precision. However, we should keep in mind that such classifier is producing ten false positives for

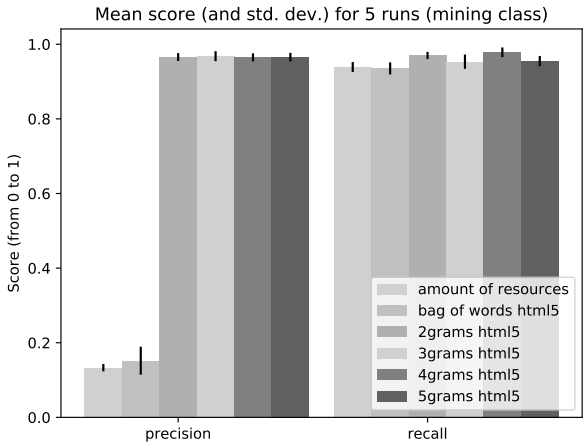

Figure 3: Detection Performance for Mining Class

10 million sites, so it will generate one false positive the 0.0001% of the times it is used.

The benign and mining classes whose results are presented in Table 2 and Table 3 have unbalanced sizes, i.e., 285.858 and 656 sites, respectively. So, although the real classes are not as unbalanced as the previous example, a small amount of false positives or false negatives has a considerably higher impact on precision and recall for the mining class than for the benign class. Further, due to these high variations, introduced by a few false positives, the smaller class has higher deviations than the benign class.

Appendix A formalizes the intuition behind class sizes and how this affects the precision and recall metrics for the smaller class. This formulation is used to calculate a theoretical upper bound of false positives and false negatives with respect to the whole dataset, i.e., 285.919 sites. This reveals that *although q-grams perform better than the bag of words- or the resource-based approach, i.e., 0.008% of false positives with respect to the whole dataset, the latter approaches are also working well in practice.* Moreover, we show that the resource-based and the bag of words approaches produce 1.5% and 1.3% of false positives (compared to the whole dataset) while producing around 0.015% of false negatives.

Another takeaway from Table 2 and 3 is that q-grams perform similarly. The f1 score, precision and recall show that even though there seems to be a slight trend to increase performance towards 4-grams and then decrease for 5-grams, the standard deviation values

do not let us have certainty that this is the case. So, depending on performance overhead introduced by the q-gram calculation, a reasonable trade-off could be to use 2-grams instead of 5-grams: more on this is explained in Section 4.2.1.

During all the executions on the evaluation set, all q-gram approaches only flagged 22 sites as "false positives". After further investigation, we found that the classifier was **detecting new mining proxies** that were not part of the training set. For example, moneone.ga was reported to blockers only after we labeled our training set. Further, we found several domains with the same Monero URL structure, i.e., ending in /proxy or starting with ws01. In some cases, we came across proxies that were not part of the blacklists but are clearly associated with miners, i.e., wp-monero-miner.de. **This implies that the performance of the classifiers is even higher than the reported values**. For completeness, the sites along with the proxies as listed in Table 6 (Appendix B).

4.2 Impact on the Page Loading Time

We have produced a proof-of-concept implementation using Chrome Debugging Tools to inject the necessary scripts to monitor the resource-related APIs and apply the bag of words classifier predicting whether a site is mining or not. However, to quantify the overhead on the client side, we split our analysis on the runtime performance on the client side into two parts. First, we measure the overhead induced on page-loading time when APIs are monitored and bag-of-words or q-grams are being calculated simultaneously. Then, we evaluate in which cases can a classifier be executed within the browser and its execution time. We split the analysis into two parts because the classification takes place after the full site has been loaded; thus, it is not useful to attempt to measure the impact of the classifier on the page-loading time if it runs seconds after the page is rendered. On the other hand, the constant monitoring and calculation of q-grams can impact the page loading time; thus, it is critical to measure its impact during the initial stages of rendering.

Further, our measurements are a worst-case scenario from the runtime performance point of view because we are instrumenting the browser APIs with JavaScript from a remote program (Crawler Node). Therefore, an actual implementation of this mechanism within the browser can only improve the user's experience or deploy more powerful mechanisms than our prototypical version.

4.2.1 Speed Index. Analyzing the performance impact on page loading time for real Websites is a technically challenging problem. Previously, there have been a number of attempts to measure a particular event, e.g., window load, reflecting how favorable the user's experience is on a Website; however, these metrics were not successful because a single event is affected by a number of factors and not necessarily correlates with the moment when content is shown [11]. To overcome this, the speed index calculates a curve of visual completeness (Figure 4) using video captures. In particular, the index is calculated based on the moment when a page has finished loading. To have a bounded value evidencing the number of pixels that took certain time to be rendered, the speed index calculates the area over the curve (up to 100% of visual completeness), i.e., the background of the image that is not covered by histogram bars. Thus, a *high speed index reflects a site that is taking long to render content.* The main drawback of this index is that developers

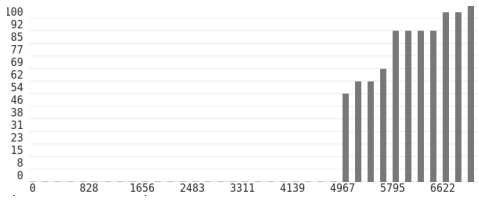

Figure 4: Facebook's SpeedIndex Visual Completeness Rendered by Speedline [20]

may need to remove visual content that is modified periodically, such as ads or banners, as the speed index would classify this as content which has not yet been loaded (as it keeps changing).

The speed index is the better-known way to analyze how fast sites are loading; consequently, we used it to analyze the impact of the API instrumentation and calculation of the bag of words or q-grams array. More to the point, we measured the speed index using the same setup described in Section 3.1, with some differences. First, only one Crawling Node was executed per VM. Further, we modified the browser instrumentation to calculate q-grams directly in the browser instead of streaming the HTML events to the Crawler Node. Also, instead of visiting a site only once, the speed index evaluation executed 20 consecutive visits (each one with a new container). For Each visit, we performed remote browser profiling and then used the Speedline library [20] in the Crawler node to calculate the speed index over the profiling data and store it in the database.

As the speedindex can severely vary due to a number of factors, we are mostly interested in the overhead induced by each API-based approach instead of the actual speed index for each site. Therefore, we executed five kinds of automated visits: baseline, bag of words, 2-, 3- and 4-grams. The **baseline** experiment opened a Website and then closed it without performing any API instrumentation. The **bag of words** experiment instrumented all APIs and monitored the number times each API call was performed. The **2-, 3-, and 4-grams** experiments also instrumented the resource-related APIs but additionally calculated the count for each q-gram observed during the execution. Moreover, to have real comparable results, we executed all the speed index calculations simultaneously (baseline, bag of words and q-grams) between June the 8^{th} and June the 10^{th} 2018. This avoids any network delays or content updates on the Websites that could affect our measurements if they were executed at different times.

As discussed in Section 4.1, we need to evaluate the overhead induced by 2-, 3, and 4-grams. Depending on the performance overhead, a trade-off may be required to save page-rendering time. Notwithstanding, it is clearly not a good idea to use 5-grams because we already have an indicator that its detection performance may drop, and in any case, 5-grams require more resources than the smaller q-grams.

When performing the speed index measurements, we encountered a problem with specific sites whose speed index varied too much across the 20 visits. We already suspected this could relate to dynamic content; withal, we decided to investigate further. Figure 5 shows a histogram for the standard deviations (calculated based on 20 subsequent visits per site) for the Alexa top 100. This

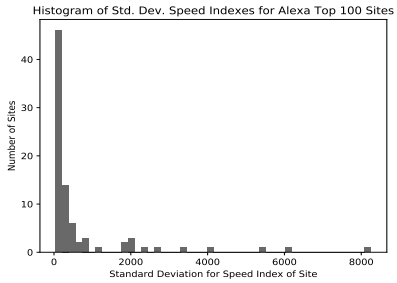

Figure 5: Histogram of Std. Dev. Speed Indexes for Alexa Top 100 Sites

gave us a clear experimental threshold (deviation of 1000) to keep most of the sites who had a deviation within the most significant part of the population. From 88 sites that we were able to visit 20 times successfully from the top 100[6], 61 sites had a smaller standard deviation than the threshold while 21 sites were above.

Interestingly, we found that although in several cases the speed index deviation happened due to visual elements that kept changing, e.g., banners or auto-playing videos, there seemed to be more sites from Chinese domains or companies in the list. According to an online forum discussing the topic, there seem to be several factors influencing network conditions when Chinese sites are accessed from outside of China. Apparently, China uses three locations for the Great Chinese Firewall, charges significant amounts of money for abroad connections and does not have enough submarine cable connections in proportion to their population [37, 45]. If these arguments are indeed true, this would explain the variations on the speed index due to significant latencies and unpredictable network conditions. For the full list of sites see Table 9 in Appendix B.

Due to space reasons, we report the mean speed index (based on 20 visits for each site) for the top 20 Alexa sites in Figure 6, but we still present the full Alexa top 100 list in Figure 7, as well as the behavior of sites excluded from the calculations in Figure 8.

Figure 6 shows the mean speed index and their standard deviations for the top 20 sites within the deviation threshold described above. Overall, visits yield a higher index as the computation increases, e.g., baseline requires less computation than to bag of words and therefore yields a lower speed index. Notwithstanding, although there is a clear trend for more complex monitoring schemes to increase the speed index, the overhead induced is not significant enough to guarantee that all sites have higher indexes for all comparisons. For example, Instagram's speed index has a lower value for 4-grams than for 3-grams, although it exposes an increasing behavior between the baseline, bag of words, 2- and 3-grams. This can be related to some dynamic features of the sites, but it could also happen because values for the mean of the speed index lie within an error margin (related to the standard deviation). So, we can only know that they are "close" the value shown.

To provide an even more intuitive view on the results from the Alexa top 100 speed index, Table 4 shows the mean and standard deviation of the overhead of the different instrumentation and

[6]Remember the non-deterministic bug described in Section 3.1

Approach	Overhead Mean	Overhead Std. Dev
bag of words	9.258832 %	10.167985 %
2-grams	24.943266 %	15.672606 %
3-grams	38.311742 %	26.489506 %
4-grams	39.164961 %	24.461527 %

Table 4: Overall Overhead in Comparison with the Baseline

vector calculation approaches using the baseline as reference for the different instrumentation approaches. It is clear that q-grams can have an overhead ranging from 20 to 40% on the speed index.

4.2.2 Classifier Execution. To evaluate to which extent can be classification mechanisms based on JavaScript features be executed directly in the browser today, we used a libSVM port to WebAssembly [24] to attempt to execute an SVM. Unfortunately, we found out that only the bag of words SVM could be executed in the browser. The primary constraint for the q-grams was that WebAssembly runs out of memory when trying to load the SVM. The main difference between the implementation of the libSVM WebAssembly library and Scikit learn is that Scikit lean provides the "triplet" representation for sparse data, and the WebAssembly wrapper requires a regular matrix with the samples.

Withal, we executed a micro benchmark in JavaScript measuring the time in milliseconds required to do 100.000 classifications individually in a loop using a Lenovo TS470S with 16 GB of RAM and an Intel Core i7-7500U CPU @ 2.70GHz. We decided to perform many classifications between the time measurements to decrease possible measurement error and because we realized that sometimes classifications took less than one millisecond. After executing the micro-benchmark described above ten times, we obtained a mean of 181 milliseconds, for the 100.000 predictions and a standard deviation of 6.54 milliseconds. So, we can say that in average each prediction takes 0.01 milliseconds.

5 RELATED WORK

Although there has been an initial study of the characteristics, e.g., timeline of events and adoption of particular miners, of in-browser crypto-mining [18], we are the first describing a detection mechanism that does not rely on blacklists and therefore achieves better detection performance. Also, this is the first quantitative analysis comparing resource and API-based detection of crypto-mining. However, we are not the first studying malicious JavaScript code exploiting the browser. In fact, we have leveraged similar concepts than those described by IceShield [19], Cujo [38] and HoneyMonkey [48]. In spite of extensive research efforts to analyze JavaScript malware using honeypots, e.g., PhoneyC [27] or WebPatrol [4], we focus on systems using real browsers for their measurements.

Provos et al. [35] demonstrated in 2008 that 1.3% of Google searches had a drive-by download attack. This created a wave of offline detection (according to the term introduced in Section 2) approaches. However, in 2005 malicious JavaScript code was already being automatically analyzed by Wang et al. when they introduced HoneyMonkey [48]: an automated crawling framework to detect malware downloads through the browser. HoneyMonkey simulated human behavior during visits to malicious sites, under Windows

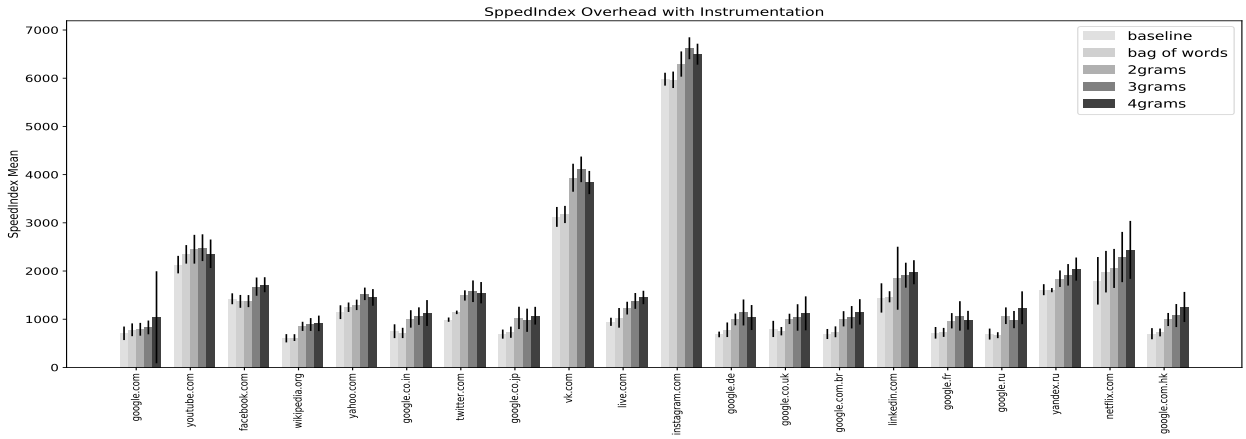

Figure 6: Speed Index comparison vs Instrumentation Method (top Alexa 20 sites)

XP, while monitoring operations to the registry performed by the machine. As the program simulating the user was not instructed to accept to download or install malicious software, activities observed in the registry after the end of the visit flagged exploit sites. Capture-HPC uses several HTTP clients, such as Firefox or Internet Explorer to orchestrate them and monitor the file system, registry, or processes a kernel level. As a result, it can flag malicious behavior whenever an unexpected change takes place [41].

We are following a similar approach as HoneyMonkey and Capture-HPC in the sense that we also use real browsers to visit sites but at the same time perform data collection in lower layers, i.e., docker would be analogue to the OS.

Heidrich et al. created an in-browser solution to detect and mitigate malicious Websites called IceShield [19]. IceShield has been implemented as a JavaScript Internet Explorer or Firefox extension that redefines browser APIs to monitor JavaScript code, extracts features and then uses Linear Discriminant Analysis (LDA) to detect malicious Websites. As the most significant contribution from IceShield is to provide a safe and lightweight manner to rewrite the browser APIs, the evaluation of their LDA is not as extensive as others found in the literature. Our approach uses similar techniques as IceShield to redefine the resource-related APIs and monitor them, yet we do this by instrumenting the browser through the Chrome Debugging Protocol API in a research context instead of using extensions for this.

Cova et al. used HtmlUnit, a headless browser based on the Rhino JavaScript interpreter, to create JSAND [8]. JSAND collected particular features related to four phases observed in drive-by downloads: redirection and cloaking, de-obfuscation, environment preparation, and exploitation. Also, JSAND was publicly available for some time as Wepawet, and the authors validated around 140.000 pages by 2010. Song et al. introduced an approach to prevent drive-by downloads via inter-module communication monitoring [42]. Song et al. created a browser module to monitor when COM modules were created, invoked or freed. Also, to increase the number of malicious samples that would attack the browser, the implemented an ActiveX emulator, to simulate several vulnerabilities. Song's approach is based on a Deterministic Finite Automaton (DFA) created based on 37 known exploits manually. Thus, whenever the DFA for a website reached a dangerous state, an exploit alert was generated.

Egele et al. [12] performed heapspray shellcode detection through the execution of a modified Firefox, with a modified SpiderMonkey instance monitoring all string concatenation and construction operations and emulating the bytes stored therein through a library an x86 code emulation library.

Ratanaworabhan et al. proposed NOZZLE [36]: a mechanism against heapspray attacks by using a Windows binary instrumentation framework to modify Firefox's routines to validate objects in the heap during garbage collection. Unlike Egele et al. [12] who only checked strings, NOZZLE inspects all objects allocated the heap. NOZZLE disassembles the code creating a control flow graph and analyzing data flows to find valid x86 sequences and. Then, it defines a heap attack surface area based on the previous analysis.

Curtsinger et al. created ZOZZLE [9] to achieve better performance than NOZZLE. To this end, ZOZZLE uses the Detours instrumentation library to perform JavaScript code deobfuscation by hooking calls to the eval function on the JavaScript engine of Internet Explorer. Once the deobfuscated code is available, ZOZZLE parses the JavaScript and obtains the Abstract Syntax Tree (AST).

Our approach relates to JSAND [8], Egele et al. [12], Song's inter-module communication monitoring [42], NOZZLE [36] and ZOZZLE [9] as they all collect features based on observations for a particular security problem.

Rieck et al. created Cujo: a Web proxy performing static and dynamic analysis of JavaScript code to detect drive-by downloads [38]. Specifically, Cujo uses a custom YACC grammar to extract tokens through lexical analysis. The result of the lexer shows a simplified representation of the code replacing variable and function names, string values and their length and numeric values by different tokens, e.g., ID = ID + STR reflects an instruction where the concatenation of a variable and a string is assigned to a variable. Then, Cujo uses ADSandbox [10], a modification of SpiderMonkey, to execute the JavaScript code to extract abstract operations. To relate the static and dynamic analysis with one another, Rieck et al. create q-grams from the reports generated by the static analysis and ADSandbox. EarlyBird [40] is an optimization of Cujo. EarlyBird is

a modified linear SVM with the same features as Cujo, yet giving higher weights to malicious events occurring in the early stages of an attack. On the one hand, EarlyBird gives higher weights to events commonly appearing early in malicious scripts in the training set, while leaving the weight for events mostly seen in benign sites constant. Also, according to Schutt et al., the authors of EarlyBird, this approach would produce a regular linear SVM whenever the data available for training is not useful to identify which events are present early in malicious Websites.

Clearly, the closest related work for our approach is Cujo and EarlyBird because they both use q-grams and a SVM. However, Rieck et al. focus on more generic features using dynamic and static analysis, even including variable assignments; on the contrary, we focus on particular resource-related API calls and do not need to perform static analysis. The main reason for this is explained in Section 2: attackers need to sustain the resource abuse to make mining profitable; thus, a pure dynamic analysis suffices because there is no way an attacker can perform mining without repeatedly executing resource-intensive calls. Also, EarlyBird and Cujo are deployed on a Proxy server; instead, we have studied, a worst-case scenario, in which the API-based classifier is deployed directly in the browser. We consider this effort to be valuable because the increasing adoption of TLS (after Cujo was published) makes harder the deployment of secure proxies. Especially, considering that proxying TLS connections require a man-in-the-middle attack: known to decrease security and privacy for users [31].

6 CONCLUSIONS AND FUTURE WORK

To the best of our knowledge, this is the first work proposing several learning-based approaches to detect in-browser mining and evaluate them thoroughly. We compare six approaches using two data sources: system's resources consumed by the browser, and APIs used by Websites. To assess the detection performance of each one of the feature sets extracted from the datasets above, we trained and evaluated the classifiers with 330.500 sites from the Alexa top ranking.

We conclude that our classifiers close the gap between false negatives introduced by blacklists [6] and false positives resulting from CPU-based site classification [13]. More to the point, using our feature sets and our labeling approach, we found very good classifiers.

The best classifier we found has 97.84% recall, i.e., detects 97.84% of all mining sites in an entirely new dataset. Moreover, the best classifier also attains 99.7% precision; that is to say, from 100 sites predicted to perform mining, there are 99.7 on average who are indeed abusing the browser. Also, all detection mechanisms presented by us are resilient to obfuscation techniques because they only rely on dynamic information. Nonetheless, our API-based detection mechanisms can misclassify sites whose APIs usage pattern changed, e.g., we have observed differences in versions of Coinhive rendering different geometrical figures. In such cases, the classifier should be trained again with proper labels to solve the problem. Also, existing solutions such as Revolver [23] could be used to prevent evasion. The main idea behind it is to detect sites that used to be classified as malicious and are not classified as such anymore and then investigate the changes.

While every detection approach can be executed offline, according to the definition given in Section 2, we explored whether the API-based approaches, e.g., bag of words, could be deployed directly on the browser. To this end, we evaluated the impact on page-loading time induced by a prototypical API monitoring technique for the Alexa top 100 sites. Although this is just a worst-case scenario, i.e., a production-level implementation directly in the browser must be more efficient, results are somewhat encouraging. Since there are no significant detection performance differences between using 2-, 3- or 4-grams, we concluded that executing the bag of words or the 2-grams classifier would induce only 9.2% or 24.9% overhead on the speed index, respectively.

However, after analyzing the overhead on page-loading time and performance of each detection approach, we encountered several technical hurdles when attempting to implement the detection in browsers. The first issue we found was that even though Chromium is the best approach to instrument the resource-related APIs during the data collection phase, their security model makes the same kind of instrumentation in production difficult, i.e., through a Chrome extension. Extensions work on an isolated world, and changes are propagated through copies of the DOM [5]. Despite this, we implemented a proof of concept capable of executing the classifier using a WebAssembly port [24] of LibSVM [3]. To do this, we require an additional program instrumenting Chromium remotely through the Chrome Debugging Protocol. We must clarify that even though we did not implement a Chrome extension to apply the classifiers explained in this paper, this may be viable by placing a content script that modifies the browser APIs. Also, looking at OpenWPM [14] and how it uses a Firefox extension to perform JavaScript API instrumentation [15] would be a good start. However, even though this is an interesting engineering task, it does not weaken nor strengthen our analysis of the classifiers' detection performance and feasibility.

Also, we performed prediction based on a 35-second visit. This provided high-quality data and assumed a realistic scenario considering that visits take approximately 5 minutes on average [7]. However, the question of the minimum time required to achieve an accurate prediction remains open and can be explored as future work with the API events collected.

ACKNOWLEDGMENTS

This research has been supported by the EU under the H2020 AGILE (Adaptive Gateways for dIverse muLtiple Environments), grant agreement number H2020-688088.

REFERENCES

[1] Bitcoinplus. 2011. Bitcoinplus. https://web.archive.org/web/20170103133312/http://www.bitcoinplus.com/miner/embeddable. Accessed: 2018-04-06.

[2] Gavin C. Cawley and Nicola L.C. Talbot. 2010. On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation. *J. Mach. Learn. Res.* 11 (Aug. 2010), 2079–2107. http://dl.acm.org/citation.cfm?id=1756006.1859921

[3] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A Library for Support Vector Machines. *ACM Trans. Intell. Syst. Technol.* 2, 3, Article 27 (May 2011), 27 pages. https://doi.org/10.1145/1961189.1961199

[4] Kevin Zhijie Chen, Guofei Gu, Jianwei Zhuge, Jose Nazario, and Xinhui Han. 2011. WebPatrol: Automated Collection and Replay of Web-based Malware Scenarios. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11)*. ACM, New York, NY, USA, 186–195. https://doi.org/10.1145/1966913.1966938

[5] Content Scripts-Google Chrome. 2018. Work in Isolated Worlds. https://developer.chrome.com/extensions/content_scripts#isolated_world. Accessed: 2018-06-02.

[6] Catalin Cimpanu. 2018. In-Browser Cryptojacking Is Getting Harder to Detect. https://www.bleepingcomputer.com/news/security/in-browser-cryptojacking-is-getting-harder-to-detect/. Accessed: 2018-06-02.

[7] Clicktale. 2013. ClickTale's 2013 Web Analytics Benchmarks Report. https://research.clicktale.com/web_analytics_benchmarks.html. Accessed: 2018-04-06.

[8] Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2010. Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. ACM, New York, NY, USA, 281–290. https://doi.org/10.1145/1772690.1772720

[9] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2011. ZOZZLE: Fast and Precise In-browser JavaScript Malware Detection. In *Proceedings of the 20th USENIX Conference on Security (SEC'11)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=2028067.2028070

[10] Andreas Dewald, Thorsten Holz, and Felix C. Freiling. 2010. ADSandbox: Sandboxing JavaScript to Fight Malicious Websites. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*. ACM, New York, NY, USA, 1859–1864. https://doi.org/10.1145/1774088.1774482

[11] WebPagetest Documentation. 2017. WebPagetest Documentation: Speed Index. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index. Accessed: 2018-06-10.

[12] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. 2009. Defending Browsers against Drive-by Downloads: Mitigating Heap-Spraying Code Injection Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Ulrich Flegel and Danilo Bruschi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 88–106.

[13] ellenpli@chromium.org. 2018. Please consider intervention for high cpu usage js. https://bugs.chromium.org/p/chromium/issues/detail?id=766068.

[14] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1388–1401. https://doi.org/10.1145/2976749.2978313

[15] Steven Englehardt and Arvind Narayanan. 2018. OpenWPM Firefox extension Instrumenting JavaScript Code. https://github.com/citp/OpenWPM/blob/f3fc7884fd93a31c689a2228c2186500374 9cf27/automation/Extension/firefox/data/content.js#L480. Accessed: 2018-01-15.

[16] Steven Englehardt, Dillon Reisman, Christian Eubank, Peter Zimmerman, Jonathan Mayer, Arvind Narayanan, and Edward W. Felten. 2015. Cookies That Give You Away: The Surveillance Implications of Web Tracking. In *Proceedings of the 24th International Conference on World Wide Web (WWW '15)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 289–299. https://doi.org/10.1145/2736277.2741679

[17] Eset. 2018. Wayback Machine: Eset Virus Radar. https://web.archive.org/web/20180126135759/www.virusradar.com/en/statistics. Accessed: 2018-06-02.

[18] Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. 2018. *A first look at browser-based Cryptojacking*. Technical Report. Bad Packets.

[19] Mario Heiderich, Tilman Frosch, and Thorsten Holz. 2011. IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM. In *Recent Advances in Intrusion Detection*, Robin Sommer, Davide Balzarotti, and Gregor Maier (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 281–300.

[20] Paul Irish. 2016. Speedline. https://github.com/paulirish/speedline Accessed: 2018-06-10.

[21] Paul Irish. 2017. Debugging Protocol: Does 'Page.addScriptToEvaluateOnLoad' execute before the "load" event? https://groups.google.com/a/chromium.org/forum/#!topic/headless-dev/cD0iF2lpHeA. Accessed: 2018-01-15.

[22] Rafael K. 2017. NoCoin: blacklist.txt. https://raw.githubusercontent.com/keraf/NoCoin/master/src/blacklist.txt. Accessed: 2017-10-15.

[23] Alexandros Kapravelos, Yan Shoshitaishvili, Santa Barbara, Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2013. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In *Usenix security*. USENIX, Washington, D.C., 637–652. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/kapravelos

[24] Daniel Kostro. 2017. LIBSVM for the browser and nodejs. https://github.com/mljs/libsvm. Accessed: 2018-06-02.

[25] Chaoying Liu and Joseph C. Chen. 2018. Malvertising Campaign Abuses Google's DoubleClick to Deliver Cryptocurrency Miners. https://blog.trendmicro.com/trendlabs-security-intelligence/malvertising-campaign-abuses-googles-doubleclick-to-deliver-cryptocurrency-miners/

[26] Mark Maunder. 2018. WordPress Plugin Banned for Crypto Mining. https://www.wordfence.com/blog/2017/11/wordpress-plugin-banned-crypto-mining/. Accessed: 2018-01-15.

[27] Jose Nazario. 2009. PhoneyC: A Virtual Client Honeypot. In *Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More (LEET'09)*. USENIX Association, Berkeley, CA, USA, 6–6. http://dl.acm.org/citation.cfm?id=1855676.1855682

[28] Shaun Nichols. 2018. Guys, you're killing us! LA Times homicide site hacked to mine crypto-coins on netizens' PCs. https://www.theregister.co.uk/2018/02/22/la_times_amazon_aws_s3/.

[29] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. 2013. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. In *2013 IEEE Symposium on Security and Privacy*. IEEE, Berkley, CA, USA, 541–555. https://doi.org/10.1109/SP.2013.43

[30] Scipy Lecture Notes. 2018. Coordinate Format (COO). http://www.scipy-lectures.org/advanced/scipy_sparse/coo_matrix.html. Accessed: 2018-06-02.

[31] Mark O'Neill, Scott Ruoti, Kent Seamons, and Daniel Zappala. 2016. TLS Proxies: Friend or Foe?. In *Proceedings of the 2016 Internet Measurement Conference (IMC '16)*. ACM, New York, NY, USA, 551–557. https://doi.org/10.1145/2987443.2987488

[32] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (Nov. 2011), 2825–2830. http://dl.acm.org/citation.cfm?id=1953048.2078195

[33] primiano@chromium.org. 2017. Chromium Source Code Comment on DBUS Bug and Xvfb. https://chromium.googlesource.com/chromium/src.git/+/2fc330d0b93d4bfd7bd04b9fdd3102e529901f91/services/service_manager/embedder/main.cc#352. Accessed: 2018-01-15.

[34] primiano@chromium.org. 2017. dbus autolaunch causes chrome to hang. https://bugs.chromium.org/p/chromium/issues/detail?id=715658. Accessed: 2018-01-15.

[35] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. 2008. All Your iFRAMEs Point to Us. In *Proceedings of the 17th Conference on Security Symposium (SS'08)*. USENIX Association, Berkeley, CA, USA, 1–15. http://dl.acm.org/citation.cfm?id=1496711.1496712

[36] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. 2009. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *Proceedings of the 18th Conference on USENIX Security Symposium (SSYM'09)*. USENIX Association, Berkeley, CA, USA, 169–186. http://dl.acm.org/citation.cfm?id=1855768.1855779

[37] Reddit. 2016. Why are Chinese sites slow/inaccessible from outside China? https://www.reddit.com/r/China/comments/4pfhv5/why_are_chinese_sites_slowinaccessible_from/?st=j7rp5ul3&sh=ec919f8d. Accessed: 2016-09-15.

[38] Konrad Rieck, Tammo Krueger, and Andreas Dewald. 2010. Cujo: Efficient Detection and Prevention of Drive-by-download Attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10)*. ACM, New York, NY, USA, 31–39. https://doi.org/10.1145/1920261.1920267

[39] Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA. 744–746 pages.

[40] Kristof Schütt, Marius Kloft, Alexander Bikadorov, and Konrad Rieck. 2012. Early Detection of Malicious Behavior in JavaScript Code. In *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence (AISec '12)*. ACM, Raileigh, North Carolina, USA, 15–24. https://doi.org/10.1145/2381896.2381901

[41] Christian Seifert and Ramon Steenson. 2006. Capture - Honeypot Client (Capture-HPC). , Available from https://projects.honeynet.org/capture-hpc; accessed on 22 September 2008 pages.

[42] Chengyu Song, Jianwei Zhuge, Xinhui Han, and Zhiyuan Ye. 2010. Preventing Drive-by Download via Inter-module Communication Monitoring. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS '10)*. ACM, New York, NY, USA, 124–134. https://doi.org/10.1145/1755688.1755705

[43] Symantec. 2018. *Internet Security Threat Report*. Technical Report 23. Symantec. Available at http://resource.symantec.com/LP=5538?cid=70138000000rm1eAAA, Accessed: 2018-06-02.

[44] Coinhive Team. 2017. Coinhive Blog: First Week Status Report. https://coinhive.com/blog/status-report. Accessed: 2018-06-02.

[45] TeleGeography. 2018. Submarine Cable Map. https://www.submarinecablemap.com/. Accessed: 2018-06-02.

[46] The Telegraph. 2018. YouTube shuts down hidden cryptojacking adverts . http://www.telegraph.co.uk/technology/2018/01/29/youtube-shuts-hidden-crypto-jacking-adverts/

[47] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. Mccoy, A. Nappa, V. Paxson, P. Pearce, N. Provos, and M. A. Rajab. 2015. Ad Injection at Scale: Assessing Deceptive Advertisement Modifications. In *2015 IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, USA, 151–167. https://doi.org/10.1109/SP.2015.17

[48] Yi-Min Wang, Doug Beck, Xuxian Jiang, and Roussi Roussev. 2005. *Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities*. Technical Report. Microsoft Research. 12 pages. https://www.microsoft.com/en-us/research/publication/automated-web-patrol-with-strider-honeymonkeys-finding-web-sites-that-exploit-browser-vulnerabilities/

[49] x25. 2017. CoinHive Stratum Mining Proxy. https://github.com/x25/coinhive-stratum-mining-proxy. Accessed: 2018-06-02.

[50] xd4rker. 2017. MinerBlock: filters.txt. https://github.com/xd4rker/MinerBlock/blob/master/assets/filters.txt. Accessed: 2017-10-15.

A THEORETICAL UPPER BOUND FOR FALSE POSITIVES AND NEGATIVES

To bring the precision and recall values of the smallest class into context, we use the known precision and recall values for each classifier to calculate a **theoretical upper bound for the total number of false positives and negatives** that formalizes about how false positives and false negatives have higher impact on smaller classes, i.e., the reasoning explained in Section 4.1.

Based on footnote 4, the precision formula can be transformed to Equation 1. We should consider that every true positive needs to belong to the class, i.e., it needs to to be in a class and to be correctly classified; thus, the number of true positives is always smaller than the size of the class, i.e., $|TP| \leq |Class|$. With this constraint in mind, we can put an upper bound to the number of false positives using the size of the class as shown in Equation 2. Likewise, we can make the similar statement about false negatives and recall, i.e., Equation 3.

$$FP = |TP| * \left(\frac{1}{precision} - 1 \right) \tag{1}$$

$$FP \leq |Class| * \left(\frac{1}{precision} - 1 \right) \tag{2}$$

$$FN \leq |Class| * \left(\frac{1}{recall} - 1 \right) \tag{3}$$

By applying Equations 2 and 3, we created a Table 5 for the maximum number of false positives and negatives with respect to the whole dataset, i.e., 285.919 sites in total.

Classif.	False Positives	False Negatives
resources	1.493 %	0.015 %
words html5	1.282 %	0.016 %
2grams html5	0.008 %	0.007 %
3grams html5	0.008 %	0.011 %
4grams html5	0.008 %	0.005 %
5grams html5	0.008 %	0.011 %

Table 5: Upper Bound of False Positives and False Negatives Based Equation 2 and 3 Using Precision and Recall from Table 2 and 3

B ADDITIONAL PLOTS AND TABLES

This section includes all additional plots and labels mentioned in the main body of the paper.

Window	WSocket Destination
streamcherry.com/	kdowqlpt.info:443
openload.co/	azvjudwr.info:443
www.tvnetil.net/	prv.co.il:2052/proxy
mydream.lk/	googlescripts.ml:8892/proxy
www.comicsporn.net/de/	163.172.103.203:8892/proxy
duroos.org/	googlescripts.ml:8892/proxy
pokemgo.info/358kqm	abc.pema.cl/socket
www.watchfaces.be/	wp-monero-miner.de:2096
multfun.com/	m.anisearch.ru/proxy
shopon.noip.us/	googlescripts.ml:8892/proxy
www.homfurniture.com/	www.homfurniture.com/
bonkersenergy.com/	minr.pw/socket
torrentbada.net/	moneone.ga/proxy
www.harveker.com/	io.truconversion.com
gifmixxx.com/9lG0	ws001.chapi.co
www.vipmatureporn.com/	163.172.103.203:8892/proxy
anunciosintimos.pt/	googlescripts.ml:8892/proxy
www.coinfloor.co.uk/	api.coinfloor.co.uk/
madnessbeat.com/	ws001.chapi.co

Table 6: WS Connections Opnened by Frames Labeled as "Flase Postives"

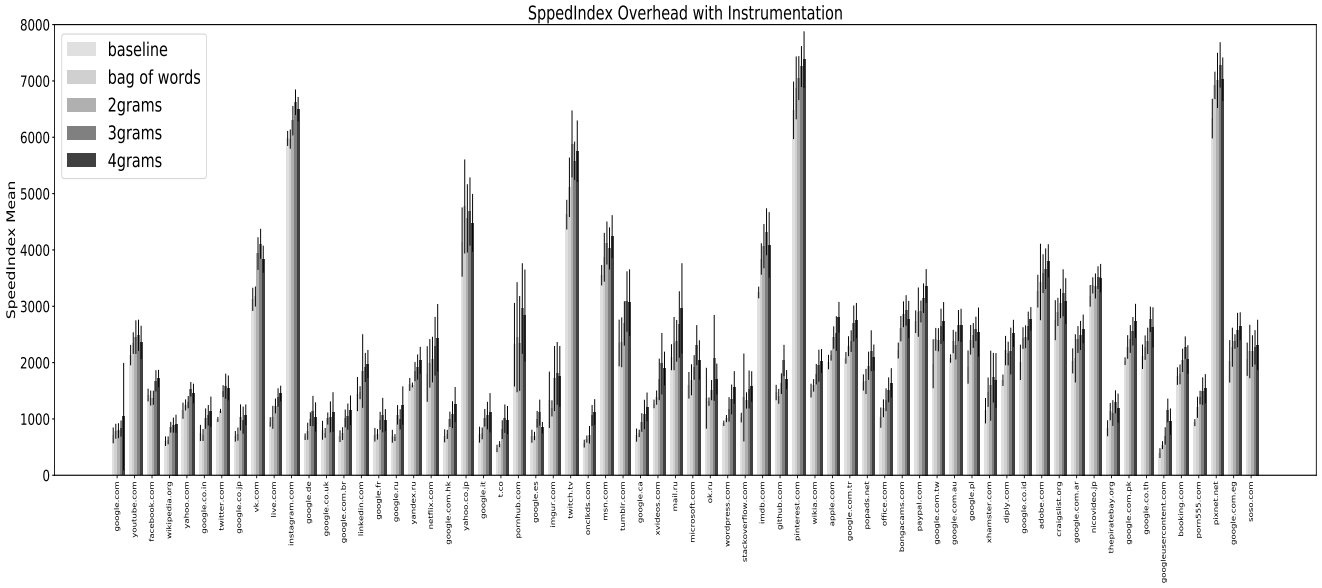

Figure 7: Speed Index comparison vs Instrumentation Method (top Alexa 100 sites)

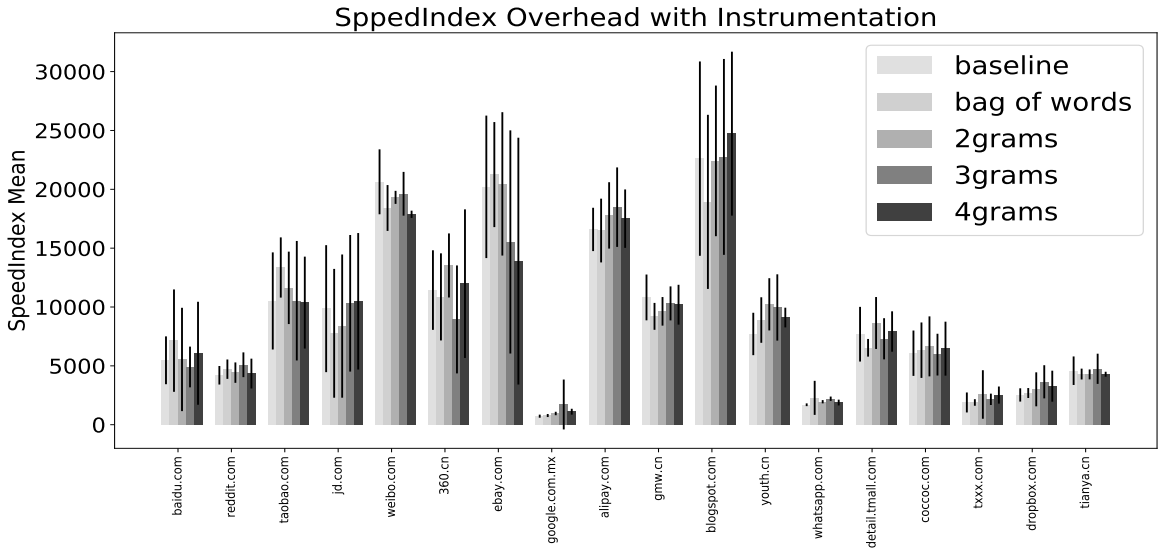

Figure 8: Speed Index values Ignored for the Alexa top 100

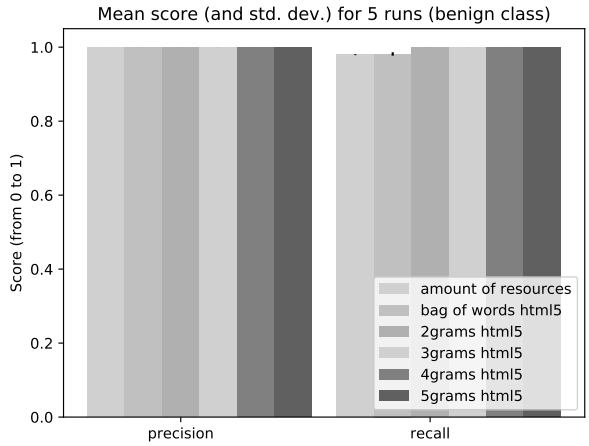

Figure 9: Detection Performance for Benign Class

Domain Match
*2giga.link
*afminer.com
*cloudcoins.co
*coinblind.com
*coinerra.com
*coin-have.com
*coin-hive.com
*coinhive.com
*coinhive-manager.com
*coinlab.biz
*coinnebula.com
*crypto-loot.com
*edgeno.de
*inwemo.com
*joyreactor.cc
*jsecoin.com
*jyhfuqoh.info
*kissdoujin.com
*kisshentai.net
*kiwifarms.net
*listat.biz
*lmodr.biz
*mataharirama.xyz
*minecrunch.co
*minemytraffic.com
minero-proxy.sh
*minero.pw
*miner.pr0gramm.com
*monerominer.rocks
*papoto.com
*ppoi.org
*reasedoper.pw
*webmine.cz

Table 7: Expressions for Mining Domains

Domain	Sites Using It
coinhive.com	611
crypto-loot.com	11
coin-hive.com	8
coin-have.com	5
minemytraffic.com	3
ppoi.org	2
papoto.com	1
2giga.link	1

Table 8: Mining Sites Grouped by Type

Alexa Domain	Ranking	Possible Reason
baidu.com	4	located in China
reddit.com	7	dynamic UI (GIF images)
taobao.com	10	located in China
tmall.com	13	located in China
jd.com	20	located in China
weibo.com	21	located in China
360.cn	22	located in China
ebay.com	36	dynamic UI (banner)
alipay.com	47	located in China
google.com.mx	44	dynamic UI (canvas)
gmw.cn	48	located in China
aliexpress.com	54	located in China
hao123.com	56	located in China
blogspot.com	61	dynamic UI (banner)
youth.cn	68	located in China
whatsapp.com	74	dynamic UI (banner)
detail.tmall.com	78	located in China
coccoc.com	82	dynamic UI (banner)
txxx.com	84	not verified (videos?)
dropbox.com	86	dynamic UI (banner)
tianya.cn	97	located in China

Table 9: Reasons for High Std. Dev. of Sites Not Considered