Early Detection of Malicious Behavior in JavaScript Code

Kristof Schütt
Technische Universität Berlin
Berlin, Germany

Marius Kloft
Technische Universität Berlin
Berlin, Germany

Alexander Bikadorov
Technische Universität Berlin
Berlin, Germany

Konrad Rieck
University of Göttingen
Göttingen, Germany

ABSTRACT

Malicious JavaScript code is widely used for exploiting vulnerabilities in web browsers and infecting users with malicious software. Static detection methods fail to protect from this threat, as they are unable to cope with the complexity and dynamics of interpreted code. In contrast, the dynamic analysis of JavaScript code at run-time has proven to be effective in identifying malicious behavior. During the execution of the code, however, damage may already take place and thus an early detection is critical for effective protection. In this paper, we introduce EarlyBird: a detection method optimized for early identification of malicious behavior in JavaScript code. The method uses machine learning techniques for jointly optimizing the accuracy and the time of detection. In an evaluation with hundreds of real attacks, EarlyBird precisely identifies malicious behavior while limiting the amount of malicious code that is executed by a factor of 2 (43%) on average.

1. INTRODUCTION

Malicious JavaScript code in web pages has become one of the major threats in the Internet. JavaScript code often provides the basis for drive-by-download attacks that unnoticeably infect users with malicious software during the visit of a web page. These attacks are conducted at a large scale and according to a recent study reach millions of victims per day [25]. Unfortunately, the JavaScript language provides a perfect platform for these attacks, as it allows for programmatically probing and exploiting vulnerabilities in web browsers and their extensions.

In contrast to other types of network attacks, malicious JavaScript code is particularly hard to detect in the content of web pages. As the code is directly interpreted during the visit of a web page, the attacker is able to literally program his attack. For example, JavaScript attacks often probe for the browser environment, check for particular vulnerabilities and use dynamic exploiting techniques, such as heap spraying, for compromising a victim's system. Even worse, the direct execution of the code also enables effectively obfuscating the attack, such that indicative patterns are only visible at run-time and not accessible by static detection methods, such as conventional anti-virus scanners.

As a result of this development, the detection of malicious JavaScript code at run-time has gained a focus in security research. Several methods have been developed for dynamically spotting malicious activity, which range from specific detection systems for certain attack types [e.g., 7, 19] to general-purpose systems that inspect the behavior using various analysis techniques [e.g., 4, 12, 13]. The difficulty of manually crafting rules for malicious behavior has further motivated the application of machine learning techniques. Several detection systems have been proposed that apply learning methods to automatically distinguish benign from malicious behavior, for example, the detectors Cujo [21], Zozzle [6] and IceShield [9].

While these learning-based detectors provide an accurate identification of malicious code at run-time, none of the detectors is optimized for an early detection of attacks. The longer a malicious code runs before it is detected, the more harm it can cause to the underlying system. Consequently, the aspired protection can be significantly weakened, if malicious behavior is identified too late. However, spotting attacks early is not a trivial task, as it imposes two challenges on a detector's design: First, malicious behavior should be detected as fast as possible, but never at the prize of accuracy. Second, the detection needs to be resistant against evasion that simply delays malicious activity to a later point in the execution of the code.

In this paper, we address the problem of detecting malicious behavior in JavaScript code as early as possible. We introduce EarlyBird: a learning method optimized for fast identification of malicious behavior. EarlyBird extends the learning algorithm of support vector machines [15, 24], such that the accuracy and the time of detection are jointly optimized during learning. The method is integrated into the Cujo detector [21] and allows for an early identification of malicious behavior with no loss of accuracy. Moreover, it is resistant against evasion attacks delaying malicious behavior during the execution.

AISEC 2012 Oct. 16-18, 2012, Raleigh, NC, USA

We demonstrate the capabilities of EarlyBird in an empirical evaluation with real JavaScript code from 100,000 benign web pages and 609 drive-by-download attacks. EarlyBird detects 93.2% of the attacks with less than 5 false alarms and performs on par with related detection approaches. In contrast to other approaches, EarlyBird limits the amount of malicious code that is executed during the analysis by a factor of 2 (43%), resulting in a better protection of the underlying system.

The rest of this paper is organized as follows: we consider the generic detection of malicious behavior using machine learning in Section 2. Our method EarlyBird is introduced in Section 3 and an empirical evaluation of its performance is presented in Section 4. Finally, Section 5 reviews related work and Section 6 concludes.

2. RUN-TIME DETECTION

Before introducing EarlyBird, we study the process of detecting malicious behavior using machine learning in the general case. We formalize some aspects of this process and thereby establish a mathematical view on the problem. We focus on the detector Cujo, as it provides the basis for the implementation of EarlyBird. Nonetheless, our approach is agnostic to the detector and could be easily incorporated in Zozzle or IceShield with minor modifications.

2.1 Run-time Monitoring

Identifying malicious activity in web pages requires a detection system to monitor the execution of JavaScript code at run-time. This monitoring is often realized by executing the code in a separate sandbox [e.g., 4, 21] or by interacting with the JavaScript engine of the web browser [e.g., 6, 9]. In both settings the flow of the execution is tracked using events that indicate changes in the state of the environment. Depending on the granularity of the monitoring, these events may range from calls to certain JavaScript functions to the observation of every state-changing action.

As an example, Figure 1 shows a snippet of a JavaScript attack and Figure 2 the corresponding events monitored by Cujo. The detector supports five basic types of events, where each type is recorded with respective arguments during the execution. For example, line 3 in Figure 2 shows a SET event that assigns the string "exe" to an internal object. The code snippet contains a trivial form of obfuscation that hides the download of an executable file. After a series of different events, this hidden download is revealed in the FUNCTIONCALL event at lines 11–12 of Figure 2.

```
a=new Array("XML", "foo", "exe")
try {
   o=new ActiveXObject("MS2"+a[0]+"."+a[0]+"HTTP")
   o.open("GET","http://"+a[1]+".com/x." + a[2],true);
} catch(e) {};
```

Figure 1: Obfuscated JavaScript code.

To formally describe this process of run-time monitoring, we consider a set of events E, such that the execution of JavaScript code in web pages results in a sequence x of these events, that is,

$$x = (e_1, e_2, \dots, e_n) \quad \text{with} \quad e_i \in E.$$

```
 1 SET CUSTOM_OBJECT_22.0 TO "XML"
 2 SET CUSTOM_OBJECT_22.1 TO "foo"
 3 SET CUSTOM_OBJECT_22.2 TO "exe"
 4 SET global.a TO CUSTOM_OBJECT_22
 5 CONVERT ActiveXObject TO A FUNCTION
 6 CONSTRUCTOR ON CUSTOM_OBJECT_24 CALLED
 7 SET global.o TO NEW_OBJECT_FROM_CONSTRUCTOR
 8 CONVERT NEW_OBJECT_FROM_CONSTRUCTOR TO A OBJECT
 9 CALL open
10 CONVERT NEW_OBJECT_FROM_CONSTRUCTOR.open TO A FUNCTION
11 FUNCTIONCALL open ("GET", "http://foo.com/x.exe",
12                    "BOOLEAN PRIMITIVE true")
```

Figure 2: Example of monitored events.

Different features can be derived from the monitored events, for example, Zozzle constructs abstract syntax trees from executed code and Cujo extracts q-grams of tokens from the events and their arguments. Without loss of generality these derived features can be viewed as events themselves and thus it suffices in our case to consider sequences of events for modeling the detection process.

2.2 From Events to Vectors

Sequences are a natural representation of behavior, yet they are not directly suitable for the application of learning methods, as these usually operate on vectorial data. To address this problem, we map each sequence x of monitored events to a vector $\boldsymbol{\phi}(x)$. Our map $\boldsymbol{\phi} : x \mapsto \mathbb{R}^{|E|}$ embeds x in a binary vector space spanned by all observable events E. A dimension in $\boldsymbol{\phi}(x)$ is 1 if the corresponding event occurs in the sequence x and 0 otherwise. Using an indicator function $\mathbb{I}_e$, we can express this map as follows

$$\boldsymbol{\phi} : x \rightarrow (\mathbb{I}_e(x))_{e \in E} \quad \text{with} \quad \mathbb{I}_e(x) = \begin{cases} 1 & \text{if } e \text{ occurs in } x \\ 0 & \text{otherwise.} \end{cases}$$

Note that depending on the size of E the resulting vector space may yield a huge dimensionality. Fortunately, a sequence of n events, only induces n non-zero dimensions in the vector space and thus the map $\boldsymbol{\phi}$ is sparse. This sparsity can be exploited to efficiently operate in the vector space and apply learning methods with little overhead [see 20].

The map $\boldsymbol{\phi}$ enables us to describe the problem of detecting malicious behavior using the geometry of the mapped sequences. Instead of matching patterns directly, we can use geometric models, such as hyperplanes and hyperspheres, to derive detection models. Moreover, the mapping enables us to apply standard optimization techniques to tune the detection for an early identification of malicious activity.

2.3 Learning and Detection

Once the sequences of events have been mapped to a vector space, several algorithms from the area of machine learning can be applied for detecting malicious behavior of JavaScript code. For analyzing behavior dynamically, linear detection methods are favorable over involved learning algorithms, as they allow to efficiently update the detection function during operation.

For the construction of EarlyBird we focus on linear *support vector machines* [SVM, 5, 8] which are also used in the Cujo detector. Given vectors of two classes as training data, a linear SVM learns a hyperplane in the vector space that separates the two classes with maximum margin. In our setting, these classes correspond to sequences of monitored events for benign (−) and malicious (+) JavaScript code. The hyperplane is learned by determining a weight vector $\boldsymbol{w}$ specifying its direction and a bias b specifying its offset from the origin. The detection function of an SVM for an unknown sequence x of events is given by

$$f(x) = \langle \boldsymbol{\phi}(x), \boldsymbol{w} \rangle + b = \sum_{e\in E} \mathbb{I}_e(x)\, w_e + b.$$

where $f(x)$ simply returns the orientation of $\boldsymbol{\phi}(x)$ with respect to the hyperplane. That is, $f(x) > 0$ indicates malicious behavior in the sequence x and $f(x) \leq 0$ corresponds to benign activity.

The computation of f amounts to a simple summation of linear terms over the events E, where the indicator function $\mathbb{I}_e(x)$ picks only those e occurring in x. Consequently, we can unroll this summation and express $f(x)$ as follows

$$f(x) = \underbrace{w_{e_1} + w_{e_2} + \ldots + w_{e_n}}_{\text{events occurring in } x} + b.$$

The indication function $\mathbb{I}_e(x)$ can be omitted here, as it is only 1 for events occurring in x. This formulation allows us to compute f incrementally and thereby provides the basis for an early detection of attacks. Whenever a new event e_i is monitored, $f(x)$ is updated by adding the corresponding weight w_{e_i} from the vector $\boldsymbol{w}$ to $f(x)$. If the update results in the condition $f(x) > 0$, malicious behavior is detected and an alert can be raised.

Similar learning methods are also applied in other detectors for malicious JavaScript code. For example, IceShield uses linear discriminant analysis for detection of malicious code. Similar to SVMs, this learning method also determines a vector $\boldsymbol{w}$ and a bias b, and thus with minor modifications can also be updated at run-time.

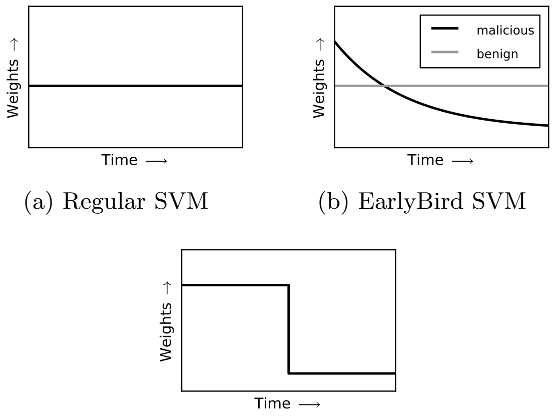
3. EARLY DETECTION

After discussing the generic of detection malicious behavior, we turn to the temporal aspect of our work. Since the detection is to be applied during the execution of possibly malicious code, an alert should be raised as early as possible. On the other hand, the overall performance in regard to a high detection rate and few false alarms shall not be sacrificed for earliness. As a remedy, we propose EarlyBird, a method for early detection of malicious code based on the paradigm of support vector machines [3, 24].

3.1 Core Idea: Temporal Weighting

The general idea of our approach is to favor dimensions in the vector space corresponding to malicious events that occur early in time, while penalizing those corresponding to events that occur in the final execution phase of malicious code and thus are not of much use for early detection.

For an illustration consider Figures 3 (a) and (b), which depict the temporal weightings of a regular SVM and the proposed EarlyBird method, respectively. We can observe from the figure that EarlyBird uses a temporal weighting that smoothly decreases over time (as detailed in the next section). Note, that we only penalize events from *malicious* code to avoid overfitting to early benign behavior. For comparison with EarlyBird we also consider a truncated SVM as depicted in Figure 3(c). The truncated SVM is identical to a regular SVM except that all sequences of events for learning are truncated at a specified point. As a consequence, the truncated SVM is forced to pick early events for learning a detection model.



(a) Regular SVM

(b) EarlyBird SVM

(c) Truncated SVM

Figure 3: Schematic depiction of temporal weightings for the regular SVM, EarlyBird SVM and the truncated SVM.

3.2 The EarlyBird SVM

In this section, we mathematically introduce the employed learning method. To this end, we use the following notation for weighted L_p-norms ($p \geq 1$), where the circle operator represents the element-wise product between vectors,

$$\begin{aligned}\|\boldsymbol{w}\|_{(p,\boldsymbol{\lambda})} &= \|\boldsymbol{\lambda}\circ\boldsymbol{w}\|_p \\ &= \sqrt[p]{\sum_{e\in E}|\lambda_e w_e|^p}.\end{aligned}$$

Given a vector $\boldsymbol{\lambda} = (\lambda_e)_{e\in E}$ of time penalties for each observed event, the core learning problem of this paper, EarlyBird, is given as follows:

$$\min_{\boldsymbol{w}} \underbrace{\|\boldsymbol{w}\|_p^p}_{\text{regularizer}} + C_1 \underbrace{\sum_{i=1}^{N} \ell\left(y_i, \langle \boldsymbol{w}, \boldsymbol{\phi}(x_i)\rangle\right)}_{\text{training loss}} + C_2 \underbrace{\|\boldsymbol{w}\|_{(q,\boldsymbol{\lambda})}^q}_{\substack{\text{time}\\ \text{penalty}}},$$

where $\ell : \mathbb{R} \to \mathbb{R}$ denotes a convex loss function that incurs a penalty if a sequence of events is misclassified.

We can interpret the above mathematical optimization problem as follows: the goal is to minimize the *training loss* ($\approx$ error on the training data), while the *regularizer* avoids overfitting and ensures that we obtain a small error on unseen data. Additionally, the *time penalizer* discourages selecting late occurring events. Its influence depends on the time penalty vector $\boldsymbol{\lambda}$ and the weight vector $\boldsymbol{w}$. The parameters C_1 and C_2 trade off loss against time and need to be tuned empirically on a hold-out set. If a data set is unsuitable for early prediction, a proper tuning will simply let EarlyBird degenerate to the regular SVM with $C_2 = 0$.

The norm parameters p and q regulate the sparseness of the weights. Small values of q lead to EarlyBird focusing on a few high-weighted early events, while larger values tend to suppress those in favor of more events with moderate time penalty and weight. In this case, early events will be affected only marginally. A good compromise it thus setting $q = 2$, which is our choice for the experiments.

The formulation of EarlyBird can be significantly simplified if we set $p = q$. In this case, we can define a vector $\boldsymbol{h} = (\sqrt[p]{1 + C_2\lambda_e^p})_{e\in E}$ and rewrite the optimization problem as follows

$$\min_{\boldsymbol{w}} \quad \underbrace{\|\boldsymbol{w}\|^p_{(p,\boldsymbol{h})}}_{\substack{\text{time-weighted}\\ \text{regularizer}}} \quad + \quad \underbrace{C_1 \sum_{i=1}^{N} \ell\left(y_i, \langle \boldsymbol{w}, \boldsymbol{\phi}(x_i)\rangle\right).}_{\text{training loss}}$$

Instead of having independent terms for the time penalty and the regularization, we obtain a formulation consisting of a *time-weighted regularizer* and the training loss. This formulation shares similarities with a so-called regularized risk minimization problem [26]. In fact, by introducing a weighted feature map $\tilde{\boldsymbol{\phi}}(x) = \boldsymbol{h}^{-\boldsymbol{1}} \circ \boldsymbol{\phi}(x)$ and substituting $\boldsymbol{w}$ with $\tilde{\boldsymbol{w}} := \boldsymbol{h} \circ \boldsymbol{w}$, we arrive at a classic, regularized risk minimization problem

$$\min_{\tilde{\boldsymbol{w}}} \quad \underbrace{\|\tilde{\boldsymbol{w}}\|^p_p}_{\substack{\text{time-weighted}\\ \text{regularizer}}} \quad + \quad \underbrace{C_1 \sum_{i=1}^{N} \ell\left(y_i, \langle \tilde{\boldsymbol{w}}, \tilde{\boldsymbol{\phi}}(x_i)\rangle\right).}_{\text{training loss}} \tag{1}$$

Similar formulations of learning problems have been widely studied and there exists several techniques for efficiently determining their optima. As we are interested in SVMs, all that needs to be done is setting the loss function in Eq. (1) to the loss of the SVM, $\ell(y, f(x)) = \max(0, 1 - yf(x))$, and we obtain a formulation that can be easily minimized using standard SVM solvers.

For efficient computation, we additionally derive the dual problem of Eq. (1) and thereby realize the time-weighting solely in terms of a kernel function

$$\begin{aligned} k(x_i, x_j) &= \langle \boldsymbol{h}^{-\boldsymbol{1}} \circ \boldsymbol{\phi}(x_i), \boldsymbol{h}^{-\boldsymbol{1}} \circ \boldsymbol{\phi}(x_j)\rangle \\ &= \langle \tilde{\boldsymbol{\phi}}(x_i), \tilde{\boldsymbol{\phi}}(x_j)\rangle \end{aligned}$$

Note that we can still calculate our original weight vector using back-substitution and the representer theorem [22],

$$\begin{aligned} \boldsymbol{w} &= \boldsymbol{h}^{-1} \circ \tilde{\boldsymbol{w}} \qquad (2) \\ &= (\boldsymbol{h}^{-1} \circ \boldsymbol{h}^{-1}) \circ \sum_{i=1}^{N} y_i\alpha_i(\boldsymbol{\phi}(x_i)). \end{aligned}$$

The technical derivation of the dual problem for Eq. (1) is provided in Appendix B. After this, there is no need for individual time-weighting of incoming events, as this is done implicitly by the learned $\boldsymbol{w}$. Consequently, the explicit time-weighting of events is only needed during the training of EarlyBird.

3.3 Time Penalty

So far, we have shown how our method EarlyBird jointly optimizes the error (training loss) and the time of detection (time-weighted regularizer). However, we have not discussed how the time penalties $\boldsymbol{\lambda}$ for events in our method are defined. For this definition we first need to specify a measure of time. In the case of JavaScript, a natural measure is the position of an event in the monitored sequence of events. We deliberately do not consider the time passing between events but just their positions, as this measure is robust against temporal noise and independent of the concrete execution speed of the detector.

Formally, we define the time $t(x, e)$ of an event e in a sequence x as the first position of e occurring in $x = (e_1, \ldots, e_n)$, that is,

$$t(x, e) := \begin{cases} i & \text{if } \exists e_i = e \text{ with } e_i \neq e_j \text{ for } j < i \\ 0 & \text{otherwise.} \end{cases}$$

Note that further occurrences of events do not need to be considered here, since they provide no additional information when using a binary vector space. If an event is not part of the sequence x, its time value will be zero, so that the penalty will not be influenced.

Moreover, given a set X^+ of sequences containing malicious behavior, we define $E^+ \subseteq E$ to be the set of events occurring in at least one of these sequence. For each event $e \in E^+$, we calculate the average position μ_e of its occurrences in the set X^+ as follows

$$\mu_e = \frac{1}{M} \sum_{x\in X^+} t(x, e),$$

where M is the number of sequences in X^+ that contain the event e. For all other events not occurring in X^+, we simply set $\mu_e = 0$. Based on this definition of μ_e, we can define the time penalty vector $\boldsymbol{\lambda}$ as

$$\boldsymbol{\lambda} := \left(\mu_e^{\frac{a}{p}}\right)_{e\in E}.$$

The parameter a controls the steepness of the penalty and is tuned empirically in our experiments. A higher choice of a implies increased suppression of late events relative to early events. Finally, we obtain the following time-weighting of events which is implemented in EarlyBird,

$$\boldsymbol{h}^{-\boldsymbol{1}} = \frac{1}{\sqrt[p]{1 + (\mu_e^a)_{e\in E}}}.$$

4. EMPIRICAL EVALUATION

After discussing the rather technical details of our method, we proceed to present an empirical evaluation using real JavaScript code of malicious and benign web site. Besides studying the overall detection performance of EarlyBird, we focus on experiments concerning the performance over time. Furthermore, we examine the robustness against simple evasion attacks and provide exemplary explanation for the earlier detection compared to the regular SVM.

4.1 Evaluation Data

As a data set of (mostly) benign JavaScript code, we consider the 100,000 most visited web sites according to the Alexa ranking[1]. Each of these web sites is visited automatically and its JavaScript code is executed using the dynamic analysis implemented in the Cujo detector. While we can not rule out the presence of some malicious behavior in this set, our experiments do not indicate any influence from such behavior on the final results.

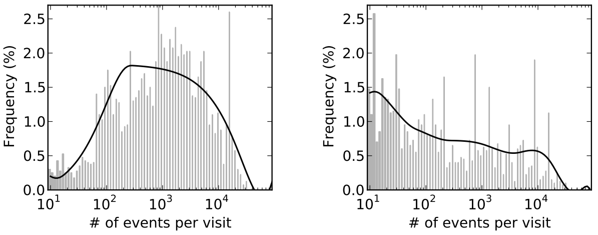
Table 1 lists the data sets of malicious JavaScript code used in our experiments together with their origin and size. These attacks have already been used to evaluate Cujo. *Malware Forum*, *Spam Trap*, *SQL Injection* and *Wepawet* are taken from Cova et al. [4], whereas the *Obfuscated* set consists of 84 additionally generated obfuscated attacks from the other sets [see 21].

[1] Alexa Top Sites, `http://www.alexa.com/topsites`

Data sets	Origin	# attacks
Malware Forum	Internet forums	256
Spam Trap	URLs spread by spam	22
SQL Injection	Injection into benign web sites	201
Wepawet	Submissions to Wepawet service	46
Obfuscated	Additionally obfuscated attacks	84

Table 1: Description of the attack data sets.

To emphasize the need for an early detection of malicious activity, Figure 4 presents histograms of the number of monitored events for both data sets. We observe that there are short and long sequences for both malicious and benign web sites with up to 10^6 events. Clearly, there is potential to reduce the ratio of executed malicious code and limit possible damage with our approach to early detection.



(a) Benign web pages. (b) Malicious web pages.

Figure 4: JavaScript events when visiting benign and malicious web pages. The distribution is smoothed using a cubic spline (black line).

4.2 Experimental Setting

In our experiments, the detection performance is measured in terms of true-positive rate and false-positive rate. As our method requires labeled data for learning, we split each data set into a known partition (75%) and an unknown partition (25%). The known partition is used to train the classifier and determine the parameters. The detection performance is measured on the unknown data set, thus all detected attacks are unknown during the learning phase. To achieve statistical sound results, we repeat this procedure 50 times and average the results.

EarlyBird operates over time so that *all* attacks detected during run-time need to be taken into account. It is not sufficient to only evaluate the attacks a posteriori, i.e., after all events of a script have been observed. To tackle this, we take snapshots of detected attacks at predefined numbers of already observed events. The accumulated true-positive rates TPR_t and false-positive rates FPR_t are then determined on the union of snapshots taken until t events have been observed.

Furthermore, we need an appropriate target function for parameter selection. Given the vector of true-positive rates on the snapshots $\mathbf{TPR} = (\mathrm{TPR}_i)_{i=1\ldots N}$ and the numbers of observed events at those snapshots $\boldsymbol{t} = (t_i)_{i=1\ldots N}$ with $t_i < t_{i+1}$, we choose the parameters of EarlyBird that maximize the following function on the known partition

$$f_{target}(\boldsymbol{TPR}, \boldsymbol{t}) = \sum_{i=1}^{N-1} \mathrm{TPR}_i \log\left(\frac{t_{i+1}}{t_i}\right).$$

The detection threshold of the SVM (bias b) for each snapshot is then determined so that there are no false alarms on the known partition. The snapshot times $\boldsymbol{t}$ for the experiments are chosen to be on a logarithmic scale. In our experimental setting, they are used not only for parameter selection, but also for testing. The logarithmic scale is chosen to emphasize early detection. Besides that, later occurring events have less influence relative to the events already seen, so later detections are less frequent.

4.3 Detection Performance

We first study the overall performance of EarlyBird, that is, the accumulated true-positive and false-positive rates. As a simple baseline, we use the truncated SVM that only considers the first 100 observed events during training. We further evaluate two setups for the regular SVM: Testing over time with accumulated detections as described in the previous section and testing with the full behavior as realized in Cujo. Moreover, we run the anti-virus scanner AVG Anti-Virus against our data as a baseline for static detection of malicious JavaScript code.

Table 2 shows the final detection performances of the considered methods for each data set individually as well as averaged over all sets. The left part of the table presents methods capable of detecting malicious code at run-time (EarlyBird, truncated SVM and the SVM over time), while the right part shows approaches only detecting attacks at the end of monitoring (the SVM as implemented in Cujo and the anti-virus scanner AVG Anti-Virus).

EarlyBird achieves the highest averaged true-positive rate with 93.2%. The method outperforms each of the other methods in three out of five attack sets. In contrast to results reported in [21], the second best method is the anti-virus scanner. As all of the considered attacks are over one year old, it is not surprising that corresponding signatures have been added to anti-virus scanners by now. While EarlyBird can not quite reach the low false-positive rates of approaches applied to the full monitored behavior, it achieves the lowest false-positive rate of the methods operating over time. It is an important insight that the increased false-positive rates are not caused by the emphasis on early prediction but are a generic problem of detection methods operating over time.

4.4 Detection Time

After demonstrating the overall performance of EarlyBird, we explore its time-based detection in more detail. The truncated SVM and the regular SVM over time will serve as comparison, again.

Data sets	EarlyBird	Truncated SVM	SVM
Malware Forum	46.4%	49.4%	68.2%
Spam Trap	33.7%	42.4%	70.0%
SQL Injection	55.7%	47.9%	78.7%
Wepawet	54.8%	50.7%	80.8%
Obfuscated	57.2%	28.9%	69.5%
Average	43.6%	43.8%	70.5%

Table 3: Ratios of executed code of EarlyBird, regular SVM and truncated SVM.

First, we want to study which ratio of malicious code is executed before an attack is detected. Attacks that are not detected are considered to be 100% executed. Table 3 shows

Data sets	EarlyBird		Truncated SVM		SVM (over time)		SVM (end)		AVG	
	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR
Alexa-100k	-	0.005%	-	0.012%	-	0.010%	-	0.000%	-	0.002%
Malware Forum	81.9%	-	75.5%	-	75.2%	-	70.5%	-	93.0%	-
Spam Trap	100.0%	-	95.8%	-	98.3%	-	98.1%	-	88.3%	-
SQL Injection	99.0%	-	86.3%	-	99.7%	-	99.0%	-	95.5%	-
Wepawet	92.8%	-	90.3%	-	89.0%	-	87.5%	-	100.0%	-
Obfuscated	98.9%	-	97.5%	-	99.0%	-	97.0%	-	25.0%	-
Average	93.2%	0.005%	88.5%	0.012%	90.1%	0.010%	88.0%	0.000%	91.8%	0.002%

Table 2: Accumulated true-positive and false-positive rates of all evaluated detection approaches. Methods on the left detect attacks over time, while methods on the right identify attacks at the end of their execution.

those ratios for each attack set. This experiment is performed on snapshots, so the values can be understood as upper bounds. As expected, the SVM detects attacks the latest of the three compared methods because it is not optimized for early detection. With an average execution ratio of 70.5%, it executes more than 1.5 times the code of EarlyBird and the truncated SVM. Those lie level at 43.6% and 43.8% execution ratio, respectively.

We additionally study the performance over time in terms of true-positive rate. Since it does not matter *when* a benign example is misclassified as an attack, the false-positive rate does not need to be analyzed time-wise.

Figure 5 shows the results of this experiment. Most of the time, EarlyBird performs best and it succeeds in its main objective of early detection. The truncated SVM is only for a short period time better than our method, while the regular SVM is outperformed completely. After 1,000 observed events, our method detects 91.4% of the attacks, while the regular SVM achieves a rate of 65.5%. Only after more than 30.000 observed events, the SVM attains a true-positive rate of 90%. Thus, EarlyBird reaches this detection performance more than 30 times faster than the regular SVM in this experiment.

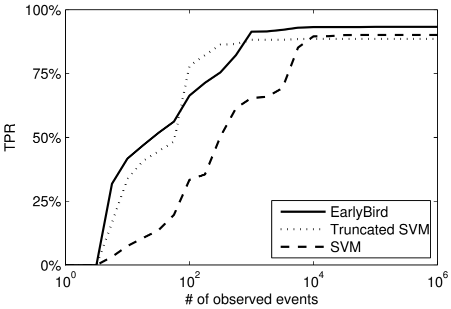


Figure 5: Accumulated true-positive rates over time of EarlyBird, regular SVM and truncated SVM.

4.5 Explanation

An advantage of the linear SVM over other learning algorithms is that the learned weight vector can be analyzed for explaining the detection of attacks [see 21]. In particular, we use this technique to study why EarlyBird identifies attacks faster than the regular SVM. In contrast to the previous experiments, we now have to look at the specific events that achieved high weights during learning. Since EarlyBird builds on Cujo each event here corresponds to a 3-gram of tokens from the dynamic analysis.

Change	**Feature**	**Time**
$\Delta w_s \cdot \phi_s(x)$	$s \in S$ (3-grams of words)	μ_e
0.059	`OBJECT CALL unescape`	8.5
0.051	`NATIVE FUNCTIONCALL unescape`	9.5
0.051	`unescape NATIVE FUNCTIONCALL`	9.5
0.051	`CALL unescape NATIVE`	9.5
0.044	`FUNCTIONCALL unescape SET`	10.5

(a) Top 5 contribution increases

Change	**Feature**	**Time**
$\Delta w_s \cdot \phi_s(x)$	$s \in S$ (3-grams of words)	μ_e
-0.795	`CALL substring SET`	3068.3
-0.628	`TO HEAPSPRAYING DETECTED`	2700.5
-0.103	`TO "..." SET`	3706.2
-0.102	`TO "..." SET`	3692.5
-0.101	`TO "..." SET`	3685.2

(b) Top 5 contribution decreases

Figure 6: Exemplary explanation of faster detection by EarlyBird.

Figure 6 shows the five highest increases and decreases of contributions of an attack from the *Malware Forum* set in comparison with the regular SVM. Non-printable binary data occurring has been replaced by ”. . . ”. All top 5 increases include the word `unescape` which signals an obfuscation. Those events occur very early at an average of 8.5 to 10.5 observed events. In contrast, the top 5 decreases in Figure 6(b) occur quite late and represent the features indicating the actual attack. `HEAPSPRAYING DETECTED` is an event triggered by the sandbox used in Cujo reflecting unusual memory activity. Obviously we want to detect the attack before the sandbox does.

For the presented example, EarlyBird seems to detect the attack faster than the regular SVM, because it focuses on the obfuscation instead of the heap spraying. However, this is only a small extract from a far more complex detection model with several millions of events. Neither is the detected obfuscation the only factor taken into consideration by EarlyBird for this example nor does the result imply a general detection preference of obfuscation over full attacks.

4.6 Evasion Attempts

Finally, we explore how robust EarlyBird is against simple evasion attempts. Using the average time of events for early detection could possibly expose new vulnerabilities. An attack that is padded with benign events at the beginning might exploit the increased emphasis on early events and evade the detection.

For this experiment, we modify the malicious behavior such that it is padded at the beginning with the 100 events taken randomly from benign behavior. We then explore two different evasion setups:

- Padding of 10% of both the unknown and known partition, corresponding to the case where training data is poisoned.
- Padding of 10% of the unknown partition only, corresponding to the case where the evasion attack starts after the training.

The results for this experiment are presented in Table 4 and Figure 7. The truncated SVM breaks down severely as soon as the training data is padded. The reason is that this method takes only the first 100 events in account, i.e. it considers only the added benign events of the padded attacks in our setting. All other methods handle the evasion attempts reasonably well. While the false-positive rates remain obviously unaffected when only the unknown attack data is modified, the impact on true-positive rates is more severe than with all data padded.

Padding	10% attack data		10% unknown attack data	
	TPR	FPR	TPR	FPR
EarlyBird	90.4%	0.010%	89.6%	0.005%
Truncated SVM	51.1%	0.008%	80.4%	0.012%
SVM (over time)	87.1%	0.011%	85.6%	0.010%
SVM (end)	84.3%	0.000%	83.7%	0.000%

Table 4: Comparison of performances for the evasion experiments with either the unknown data padded or all data padded.

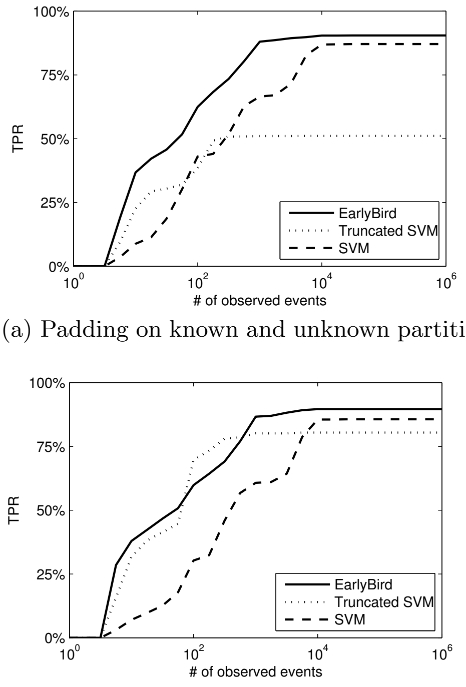
An evasion attack with 50% or more padded data causes a large performance impact on EarlyBird, so that the regular SVM would in this case achieve a better performance if the modified attacks influence the training process. In case of 100% padded test data but unmodified training data, the performance of EarlyBird would decrease to 66.5%. However, even in this setting our method would still outperform the regular SVM by 8%.

We need to note here that we have studied only a simple evasion scenario and more involved settings can be thought of. In fact, there is a branch of machine learning research exclusively concerned with learning on poisoned training data. Some of the concepts of this research can be implemented into EarlyBird, but such extensions are out of the scope of this paper.

5. RELATED WORK

With the proliferation of malicious JavaScript code in the Internet, the detection and mitigation of this threat has been a vivid area of research. Several approaches have been proposed for observing, analyzing, and detecting JavaScript attacks in the wild, for example, using high-interaction honeypots [17, 23, 27] and low-interaction honeypots [1, 10, 16]. Similarly, different systems have been proposed for offline analysis of JavaScript code [2, 4, 11, 12]. While all these approaches are effective in detecting malicious code, they usually require considerable analysis time and thus are inapplicable for protecting users at run-time.

Concurrently to these offline approaches, other work has studied the detection of particular attack types, such as heap-spraying attacks [7, 18] and drive-by downloads [14]. This work rests on identifying symptoms of certain attacks, for example, the presence of shellcode in JavaScript strings. Although these approaches mitigate the threat of web-based attacks to some extent, they are limited to specific attack types and unable to cope with the evolving domain of malicious JavaScript code.



(a) Padding on known and unknown partition

(b) Padding on unknown partition only

Figure 7: Accumulated true-positive rates for the evasion experiment with 10% padded attacks.

Recent work has thus studied combining JavaScript analysis with techniques from machine learning for deriving automatic defenses. Most notably are the learning-based detection systems Cujo [21], Zozzle [6], and IceShield [9]. Our method EarlyBird is closely related to these approaches, as it also uses dynamic analysis to detect malicious code at run-time. Similarly, EarlyBird also builds on machine learning for automatically learning a discrimination between benign and malicious code. Nonetheless, EarlyBird significantly differs from previous approaches in that not only the accuracy is considered but also the time of detection. To the best of our knowledge, EarlyBird is the first method to provide an early detection of malicious JavaScript code, while attaining the same accuracy as related approaches.

The concepts developed in this paper can also be applied to other detection systems. In particular, it is possible to extend Zozzle and IceShield by replacing their underlying learning algorithm with EarlyBird. As a result, our approach is agnostic to the concrete realization of a detector and can be applied in several scenarios where an early detection improves the quality of a security system.

6. CONCLUSIONS

In this paper, we have proposed a flexible detection method for early identification of malicious JavaScript behavior. The method uses machine learning techniques for optimizing the accuracy as well as the time of detection It allows for using arbitrary loss functions and ℓ_p-norm regularizers as well as time penalties. Empirically, the early detection of malicious code has been demonstrated in different experiments, where our method EarlyBird outperforms regular detectors and limits the execution of malicious code to a fraction of 43.6%. Furthermore, the method is robust against evasion attempts that try to exploit its time dependency.

The present work gives a technological foundation for the early detection of malicious events in general and can be applied well beyond the setting considered in this paper, for example, for classic network intrusion and behavior-based malware analysis. It can also be used beyond computer security in settings where appropriately filtering temporal or sequential data is crucial, such as in brain computer interfacing and bioinformatics.

References

[1] A. Büscher, M. Meier, and R. Benzmüller. Throwing a monkeywrench into web attackers plans. In *Proc. of Communications and Multimedia Security (CMS)*, pages 28–39, 2010.

[2] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proc. of the International World Wide Web Conference (WWW)*, pages 197–206, Apr. 2011.

[3] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[4] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proc. of the International World Wide Web Conference (WWW)*, 2010.

[5] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines*. Cambridge University Press, Cambridge, UK, 2000.

[6] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *Proc. of USENIX Security Symposium*, 2011.

[7] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 88–106, 2009.

[8] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research (JMLR)*, 9:1871–1874, 2008.

[9] M. Heiderich, T. Frosch, and T. Holz. IceShield: Detection and mitigiation of malicious websites with a frozen dom. In *Recent Adances in Intrusion Detection (RAID)*, Sept. 2011.

[10] A. Ikinci, T. Holz, and F. Freiling. Monkey-spider: Detecting malicious websites with low-interaction honeyclients. In *Proc. of Conference "Sicherheit, Schutz und Zuverlässigkeit" (SICHERHEIT)*, pages 891–898, 2008.

[11] S. Karanth, S. Laxman, P. Naldurg, R. Venkatesan, J. Lambert, and J. Shin. ZDVUE: Prioritization of javascript attacks to surface new vulnerabilities. In *Proc. of CCS Workshop on Security and Artificial Intelligence (AISEC)*, Oct. 2011.

[12] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. Technical Report MSR-TR-2011-94, Microsoft Research, Aug. 2011.

[13] Z. Li, Y. Tang, Y. Cao, V. Rastogi, Y. Chen, B. Liu, and C. Sbisa. WebShield: enabling various web defense techniques without client side modifications. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2011.

[14] L. Lu, V. Yegneswaran, P. A. Porras, and W. Lee. BLADE: An attack-agnostic approach for preventing drive-by malware infections. In *Proc. of Conference on Computer and Communications Security (CCS)*, pages 440–450, Oct. 2010.

[15] K. Muller, S. Mika, G. Ratsch, K. Tsuda, and B. Schölkopf. An introduction to kernel-based learning algorithms. *Neural Networks, IEEE Transactions on*, 12(2):181–201, 2001.

[16] J. Nazario. A virtual client honeypot. In *Proc. of USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[17] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *Proc. of USENIX Security Symposium*, 2008.

[18] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. Technical Report MSR-TR-2008-176, Microsoft Research, 2008.

[19] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proc. of USENIX Security Symposium*, 2009.

[20] K. Rieck and P. Laskov. Linear-time computation of similarity measures for sequential data. *Journal of Machine Learning Research*, 9(Jan):23–48, Jan. 2008.

[21] K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *26th Annual Computer Security Applications Conference (ACSAC)*, pages 31–39, Dec. 2010.

[22] B. Schölkopf and A. Smola. *Learning with kernels: Support vector machines, regularization, optimization, and beyond.* the MIT Press, 2002.

[23] C. Seifert and R. Steenson. Capture – honeypot client (Capture-HPC). Victoria University of Wellington, NZ, `https://projects.honeynet.org/capture-hpc`, 2006.

[24] J. Shawe-Taylor and N. Cristianini. *Support Vector Maschines and other kernel-based learning methods.* Cambridge University Press, 2000.

[25] Symantec. Symantec Internet Security Threat Report: Trends for 2010. Vol. 16, Symantec, Inc., 2011.

[26] V. Vapnik. *The nature of statistical learning theory.* Springer-Verlag New York Inc, 1995.

[27] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2006.

APPENDIX

A. MODEL SELECTION

EarlyBird is able to use arbitrary losses, L_p-norm regularizers and L_q-norm time penalizers. For this work, we restrict our model selection to the simple case $p = q$.

We use Liblinear [8], a library for large-scale linear classification, and try various available classifiers. The results of these experiments are presented in Table 5. They are produced with the procedure described in Section 4.2. All evaluated EarlyBird variants achieve true-positive rates above 90%. However, the variant with 2-norm regularizer and hinge loss, corresponding to an adapted SVM formulation, detects the most attacks and raises substantially less false alarms than the other variants.

	EarlyBird	
Regularizer/Loss	TPR	FPR
2-norm regularizer, hinge loss	93.2%	0.005%
2-norm regularizer, logistic loss	91.6%	0.020%
1-norm regularizer, squared hinge loss	90.1%	0.030%

Table 5: Comparison of accumulated true-positive and false-positive rates of various EarlyBird variants.

Figure 8 shows the true-positive rates of the EarlyBird candidates over time. The variant with logistic loss performs best in terms of early prediction. However, its false-positive rate is too high for detection malicious JavaScript code in practice. Hence, we choose the 2-norm regularization and hinge loss as default setting for this work.

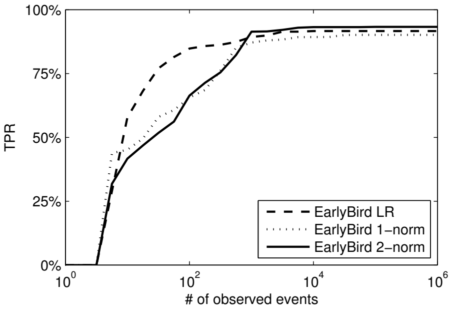


Figure 8: Accumulated true-positive rates over time of various EarlyBird variants.

B. DUALISATION

We derive the the dual problem of EarlyBird for the special case using the hinge loss and L_2-regularization. This is similar to the standard setting for an SVM and was used to obtain the results presented in this paper. It is possible to use the substitution derived in Section 3.2 and just plug the time-weighted map into the regular SVM. However, we dualize the original problem and show that we gain the time-weighted kernel implicitly.

As shown in (1), we employ a regularizer weighted with $\boldsymbol{h} = (\sqrt{1 + C_2\lambda_e^2})_{e\in E}$. The hinge loss is written using the slack variable $\boldsymbol{\xi}$,

$$\begin{aligned} \min_{\boldsymbol{w},\boldsymbol{\xi}} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2_{(2,\boldsymbol{h})} + C_1\sum_{i=1}^{N}\xi_i \\ s.t. \quad & y_i\langle \boldsymbol{w}, \boldsymbol{\phi}(x_i)\rangle \geq 1 - \xi_i \quad \forall i \\ & \xi_i \geq 0. \end{aligned}$$

We incorporate the constraints using the Lagrange multipliers $\alpha \geq 0$, $\beta \geq 0$ and gain the following Lagrangian:

$$\begin{aligned} L(\boldsymbol{w}, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}\|\boldsymbol{w}\|^2_{(2,\boldsymbol{h})} + C_1\sum_{i=1}^{N}\xi_i \\ + \sum_{i=1}^{N}\alpha_i(1 - \xi_i - y_i\langle \boldsymbol{w}, \boldsymbol{\phi}(x_i)\rangle) \\ - \sum_{i=1}^{N}\beta_i\xi_i. \end{aligned}$$

Since the SVM is convex and Slater's condition is fulfilled, the KKT conditions are necessary and sufficient for the dual solution to be optimal.

$$\begin{aligned} \frac{\partial L}{\partial w_e} &= \frac{1}{h_e^2}w_e - \sum_{i=1}^{N}\alpha_i y_i \phi_e(x_i) \overset{!}{=} 0 \\ &\Rightarrow \boldsymbol{w} = (\boldsymbol{h}^{-1} \circ \boldsymbol{h}^{-1}) \circ \sum_{i=1}^{N}\alpha_i y_i \boldsymbol{\phi}(x_i) \\ \frac{\partial L}{\partial \xi_i} &= C_1 - \alpha_i - \beta_i \overset{!}{=} 0 \\ &\Rightarrow 0 \leq \alpha_i \leq C_1 \end{aligned}$$

We gain a constraint and the formula for the weights that we also derived in (2) by back-substitution. Plugging this into the Lagrangian yields the dual optimization problem

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \sum_{i=1}^{N}\alpha_i \\ & -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j k(x_i, x_j) \\ s.t. \quad & 0 \leq \alpha_i \leq C_1 \quad \forall i \end{aligned}$$

with

$$\begin{aligned} k(x_i, x_j) &= \langle \boldsymbol{h}^{-1} \circ \boldsymbol{\phi}(x_i), \boldsymbol{h}^{-1} \circ \boldsymbol{\phi}(x_j)\rangle \\ &= \langle \tilde{\boldsymbol{\phi}}(x_i), \tilde{\boldsymbol{\phi}}(x_j)\rangle. \end{aligned}$$