Assessing Adaptive Attacks Against Trained JavaScript Classifiers

Niels Hansen[1], Lorenzo De Carli[2], and Drew Davidson[1]

[1] University of Kansas
[2] Worcester Polytechnic Institute

Abstract. In this work, we evaluate the security of heuristic- and machine learning-based classifiers for the detection of malicious JavaScript code. Due to the prevalence of web attacks directed though JavaScript injected into webpages, such defense mechanisms serve as a last-line of defense by classifying individual scripts as either benign or malicious. State-of-the-art classifiers work well at distinguishing currently-known malicious scripts from existing legitimate functionality, often by employing training sets of known benign or malicious samples. However, we observe that real-world attackers can be *adaptive*, and tailor their attacks to the benign content of the page and the defense mechanisms being used to defend the page.
In this work, we consider a variety of techniques that an adaptive adversary may use to overcome JavaScript classifiers. We introduce a variety of new threat models that consider various types of adaptive adversaries, with varying knowledge of the classifier and dataset being used to detect malicious scripts. We show that while no heuristic defense mechanism is a silver bullet against an adaptive adversary, some techniques are far more effective than others. Thus, our work points to which techniques should be considered best practices in classifying malicious content, and a call to arms for more advanced classification.

1 Introduction

Developments in adversarial machine learning are deeply affecting the science of computer security. In recent years, the research community has shown how a variety of classification techniques are vulnerable to *adversarial samples*. An adversarial sample is a malicious object—its type depending on the classification task—which exhibits features causing a target classifier to misclassify it. Adversarial samples apply naturally to a special form of classification in which there are two categories: *malicious* and *benign* (we consider software defense frameworks to be instances of classifiers in this regard, and we will use the term classifier to refer to such systems throughout this work). By subverting the training sets of security mechanisms to misclassify attacks as benign, adversarial samples have been shown to be effective against such classifiers in several contexts [2].

The potential impact of adversarial samples is significant in the domain of website protection. In this context, the task is to analyze each individual script, particularly JavaScript, being served from a page. The analysis deems a script as allowed if the script is legitimate content, or disallowed otherwise. The research community has produced classification techniques that can (i) learn descriptive features for benign and malicious JavaScript samples from a corpus, and (ii) efficiently use these features to distinguish scripts from the two classes, so that malicious scripts can be stopped before they are executed or served to the user. A key benefit of these classifiers is that they do not need to have access to an explicit whitelist of scripts, but can instead rely on heuristics or learned models. Indeed, such a classifier labels a script as benign as long as it is sufficiently similar to other scripts that are known to be benign (and, in the case of some classifiers, if it is sufficiently dissimilar from a script that it known to be malicious). In other words, the power of the classifier comes from its ability to learn general characteristics of benign code, and look for those characteristics in unknown scripts. However, trained classifiers may misclassify a script with malicious behavior but descriptive features similar enough to the benign training. An *adaptive* adversary may attempt to exploit this limitation by altering an attack script to encourage misclassification.

The degree to which an adaptive adversary can succeed in forcing misclassification in the context of JavaScript has not been fully investigated. Although previous work has been successful by proposing specific ways to mask malicious scripts, more work is needed to understand the effect of factors such as the type of classifier being attacked, the knowledge the adversary has about the benign scripts, and the type of malicious content that the adversary wants to inject. While the security literature presents various instances of adaptive attacks (e.g., [19, 9]), their effectiveness relative to each other has been understudied. In this work, we analyze several forms of adaptive attacks, and evaluate their strength against different styles of classifier. Our goal is to build a more complete characterization of the landscape of threats faced by trained classifiers, and to better characterize their ability to withstand various attempts at evasion. We consider two types of classifiers, those that directly use a script's syntactic structure for classification, and those that use scalar features indirectly derived by the syntactic structure. As representative and recently-proposed examples of classifiers, we consider CSPAutoGen, which is a structural classifier [19], and JaSt, which is a feature-based classifier [9]. We choose CSPAutoGen and JaSt for two reasons. First, they both achieve near-0% false negative rate (against a non-adaptive adversary), and they can be considered the state-of-art in malicious JavaScript detection. Second, they are based on fundamentally different mechanisms, allowing us to compare two distinct approaches to classification. Both approaches use syntactic features derived from a script's abstract syntax tree: CSPAutoGen generates a whitelist of generalized ASTs served from a website, while JaSt uses frequencies of n-grams extracted from ASTs to train a tree ensemble classifier to distinguish between malicious and benign email JavaScript snippets.

A key insight of our paper is that the strength of an adaptive adversary depends on his or her knowledge about the target. We present three threat models, corresponding to attackers with different levels of knowledge about the target site. We also introduce three novel, domain-specific mimicry attacks: the *subtree editing mimicry attack*, the *gadget composition attack*, and the *script stitching attack*.

This paper makes the following contributions:

- We examine state-of-the-art algorithms for malicious JavaScript code detection, determining their vulnerability to mimicry attacks.
- We articulate a number of threat models for attacks against current syntax-based JavaScript classifiers. These threat models both explore the design space of attacks and highlight realistic adversarial capabilities.
- We identify three classes of adaptive attacks at the AST level: *subtree editing*, *script stitching*, and *gadget composition*.
- We implement and evaluate the above attacks, characterizing their effectiveness in realistic scenarios.

The rest of this paper is structured as follows. Section 2 provides background on malicious JavaScript detection. Section 3 presents our characterization of various threat models that correspond to our domain. Section 4 describes our newly-discovered attack techniques, and Section 5 describes their implementation. Section 6 experimentally validates these attacks on a realistic JavaScript corpus. Section 7 reviews related work, and Section 8 concludes the paper.

2 Problem Overview

There are a variety of attack scenarios in which automatic JavaScript classification might be deployed. In order to focus the discussion on the case most commonly proposed in the literature, we consider the case of malicious JavaScript code injected in a website or web application, and served to the web application's users. Such malicious code may have various goals, such as extracting information from the webpages visited by the user, or downloading and executing additional payloads. The first line of defense against these attacks is to prevent the code from being added to the victim website. However—as recently demonstrated by the outbreak of infections due to the MageCart attack code [16]—relying on injection prevention alone is insufficient.

Due to the persistent vulnerability of web applications to code injection, the security community has produced forms of defense-in-depth. The goal of these defenses is to render injected JavaScript harmless. The most popular of such solutions is the content security policy (CSP) - a set of directives that can be added to a webpage to limit the set of scripts that can run in the context of that webpage. Unfortunately, CSP has been plagued by a perceived lack of flexibility and semantics that do not match the way web applications are developed in practice, and has seen limited adoption [3]. For these reasons, researchers have also investigated heuristic- and machine learning-based classifiers that automatically learn what scripts are acceptable on a webpage, and prevent anything that does not fit the model from reaching the end-user. Those classifiers are our object of study, and we discuss them next.

2.1 Existing Classification Approaches

Existing approaches to JavaScript classification cannot rely on an exhaustive list of scripts to whitelist. Indeed, in many cases websites generate scripts dynamically in a content-dependent manner, which means that the set of benign scripts may potentially be infinite [19]. Instead, the classifier labels scripts as benign if they are *similar* to a training set of benign scripts, which is usually a subset of all of the benign scripts that may be served. Some tools also use a training set of known attack scripts to serve as negative examples. These detectors are necessarily approximate, and can incur both false positives and false negatives during deployment. In order to be usable, automatic classifiers need to make allowances to reduce both types of errors. In practice, tools have attempted to recognize aspects of benign scripts that are characteristic of acceptable functionality in order to reduce false negatives. To reduce false positives, they allow some tolerances for similarity to a benign script. To illustrate these tradeoffs, we describe our two representative classifiers.

CSPAutoGen [19] is a defense framework which aims at preventing execution of malicious injected JavaScript code. Although (as the name implies), CSPAutoGen automatically generates Content Security Policies, it also includes a core template-based algorithm recognizing and allowing benign scripts (for brevity's sake we refer to this algorithm as "CSPAutoGen"). CSPAutoGen automatically learns a model of which scripts are benign using only a training set of allowed sample scripts, for which it builds generalized templates that capture the structure of the benign script's abstract-syntax tree (AST). At runtime, a client-side library component of CSPAutogen determines whether each loaded script should be allowed to run. This library parses the script under test, extracts its AST, and checks for a match against a template. Ultimately, the framework considers a script to be benign if it has a template match, and malicious otherwise. Based on its template behavior, we view CSPAutoGen as effectively solving the benign/malicious script classification problem. To our knowledge, CSPAutoGen is the most recent of classifiers based on the syntactic structure of scripts. As such, CSPAutoGen is a good representative of structural classifiers. Although it is intended to run with client and server components, its core algorithm can be evaluated in a command-line batch mode through tooling included in its open-source distribution.

JaSt [9] is representative of feature-based classifiers. It trains a random forest classifier on a corpus of labeled benign and malicious scripts. Vectorization is accomplished by computing the frequency of various n-grams in each script's AST. In order to reduce feature sparsity, only n-grams appearing in the evaluation dataset are considered, and only n-grams of length ≤ 5 are considered.

2.2 Objectives and Challenges

Consider a classifier $f : P \to \{M, B\}$ mapping any JavaScript program $p \in P$ to one of two possible classes: malicious (M) or benign (B). Given a malicious program p_M s.t. $f(p_M) = M$, an adaptive attack is a transformation $T(p_M)$

which generates a second program p_M^* which is functionally equivalent to p_M but $f(p_M^*) = B$.

There are two important points to note about our attacker's goal. First, as program equivalence is in general undecidable, care must be taken so that the transformation $T(p_M) = p_M^*$ maintains the semantic effect of the original p_M. In practice, we use a weak form of program equivalence that postulate that two programs are identical *if, when run under the same conditions, they accomplish the same security exploit.* Second, the above formulation is different from the ones traditionally used in adversarial machine learning [2, 25], which tend to stress minimization of some notion of distance $d(p_M, p_M^*)$ in feature space. In fact, in our case p_M and p_M^* can be arbitrarily distant as long as they are functionally equivalent, and a good attack sample does not necessarily lie in the vicinity of p_M (intuitively, this is a consequence of syntactic features not being a good proxy for program behavior and semantics).

3 Threat Models

A key contribution of our work is that we consider various ways in which an adversary might attack a JavaScript classifier. These threat models are intended to explore the range of scenarios that a website maintainer should consider when evaluating defense measures against an adaptive adversary.

We note that a common capability shared by adversaries in all of our threat models is the ability to inject an arbitrary script onto the target page. Thus, we assume that the adversary is capable of bypassing any form of script sanitization or input validation. While the ability to embed malicious content onto a benign page is a strong capability for overall web security, it is exactly the threat model in which malicious JavaScript classifiers operate: if a malicious script is never embedded, the classifier is unnecessary. As noted in Section 2, these tools are designed to be a last line of defense when input validation and sanitization has failed. Furthermore, the prevalence of such attacks in the wild show that real adversaries can and do hold the capability to inject content.

We differentiate our different threat models based on the knowledge that an attacker has about the system that they are attempting to evade, as well as the power of the system itself. In adversarial machine learning, the knowledge that an attacker may have about a system is typically classified as knowledge about the following aspects of the target system [2]:

- The dataset $\mathcal{D}$ upon which the classifier was trained. We note that it is highly likely that the attacker will have at least partial knowledge of $\mathcal{D}$, because the adversary has interactive access to the website that they are attacking. It is reasonable to assume that the classifier will be trained on the benign scripts that it is serving, and those scripts are freely accessible to the adversary, either by crawling the site or by recording the results of reconnaissance sessions of browsing the target site. If $\mathcal{D}$ also include sample malicious scripts, an attacker may attempt to replicate this by collecting publicly-available JavaScript exploit datasets (e.g., [1]).

– The (weighted) feature set $\mathcal{X}$ that the classifier uses to determine syntactic similarity between scripts. This information is valuable to an attacker, as they may choose to mount their mimicry by prioritizing similarity to those features that are most highly valued by the classifier.
– The learned model $\mathcal{F}$ that is used by classifier f to label a script as benign or malicious. Implicit in the $\mathcal{F}$ is the style of classification used by f.

We articulate several threat models that correspond to various levels of knowledge that the adversary has about the classifier under attack.

Scenario 1: Non-adaptive Attacker For the sake of comparison, we first consider the case of an attacker with no knowledge of the target site. This attacker essentially serves as a baseline for adaptive attacks: although the attacker may attempt to obfuscate their malicious script to make it appear more benign, the obfuscation must, by definition, be done without knowledge of what constitutes a benign script in the context of the victim. The non-adaptive attacker threat model constitutes limited evasion capabilities: the attacker has no access to the dataset $\mathcal{D}$, the learned model $\mathcal{F}$, or the feature set $\mathcal{X}$. Nevertheless, this attacker is the one modeled (implicitly or otherwise) by most classifiers.

Scenario 2: Reconnaissance Attacker The next attacker that we consider is designed to represent realistic capabilities for an attacker with regular access to the target website. This attacker has the ability to observe the behavior of pages on the victim site, including the ability to collect scripts embedded in the site such as inline JavaScript, event handlers, and callbacks. This reconnaissance phase allows the attacker to collect a partial dataset of benign scripts $\mathcal{S}'^B$, which were observed during interaction with the target. We assume that $\mathcal{S}'^B$ is nearly a proper subset of $\mathcal{S}^B$, because we expect that the classifier will necessarily be trained on the benign scripts that are served to the user (modulo minor variations, such as those caused by dynamic scripts) in order to ensure the lowest possible rate of false alarms. However, because the attacker only has access to the data served from the pages visited during the reconnaissance phase, and makes no attempt to make inferences about the target classifier through aggregate observations on the scripts in $\mathcal{S}'^B$, the attacker has no knowledge of $\mathcal{F}$ or $\mathcal{X}$. We believe that the reconnaissance attacker should be considered the minimal bar for a classifier to address in order to be considered effective. For most web sites and web apps, the target site will be publicly accessible, and therefore the reconnaissance phase can be achieved by crawling public pages or by simply collecting all scripts encountered during a regular browsing session on the target site. We note any attacker that encounters a script classifier is likely to have gone through a reconnaissance phase in the due course of finding a vulnerability that allows for malicious content to be injected onto a page. As such, attacker access to S'^B should be expected.

Scenario 3: Classifier-aware Attacker The third attack scenario we consider extends the capabilities of the reconnaissance attacker with (potentially partial) knowledge of the classifier being used to defend the target site. Because of this access, we call the attacker in this threat model the classifier-aware attacker. The classifier-aware attacker seeks to mount more sophisticated attacks than the prior two threat models. Rather than being constrained to mimicry of exactly those scripts that are observed during reconnaissance, the classifier-aware attacker can construct a mimicry attack based purely on a notion of what the classifier will accept. Note that the classifier-aware attacker is stronger than the attacker in the previous two threat models. However, this attacker still represents a realistic set of capabilities. As noted in the previous scenario, the attacker has interactive access to the target site. Thus, the attacker can build a *surrogate* by training a classifier similar to the one used to protect the site on the set S'^B that was collected during reconnaissance. While a training set S^M of malicious scripts (only necessary for some classifiers) cannot be directly obtained from the target, an attacker may build a surrogate S'^M from public sources as discussed above. In a typical attack scenario, it is unlikely that the attacker will have direct access to $\mathcal{F}$ or to $\mathcal{X}$. However, the attacker can make reasonable assumptions about $\mathcal{X}$ based on training a surrogate classifier on S'^B (and if necessary S'^M). Furthermore, previous work on *model inversion* [10] has shown that interactive attacker can gain significant details about the classifier (and indeed the underlying model) by making repeated queries to the classifier. In this case, the attacker can simply make naive attempts to inject content into the page and observe the success of the attack. In doing so, they can likely infer the type of classifier $\mathcal{F}$. Coupled with the ability to train on the surrogate data, the attacker can likely build a classifier which is quite close to that used by the site. We note that the classifier-aware attacker represents a strong adversary. Nevertheless, we feel that this model is important to consider if a classifier is to be relied upon in a real deployment, and to assess the strength of a classification system.

4 Attacks

In this section, we introduce several types of adaptive attacks against JavaScript classification. We ignore the non-adaptive, non-adversarial attacker (ref. Section 3) as this is the attacker normally assumed by target classifiers [19, 9]. The *subtree editing* and *script stitching* attacks are within the capabilities of a reconnaissance attacker (ref. Section 3), while the *gadget composition* attack assumes a classifier-aware attacker (ref. Section 3). We discuss each attack below.

4.1 Subtree Editing Mimicry Attack

The *subtree editing* mimicry attack assumes the attacker has partial knowledge of the dataset D. In particular, the attacker knows a subset of the benign scripts used for training, $\mathcal{S}'^B \subseteq \mathcal{S}^B$. Consider a malicious script M the attacker wishes to be classified as benign by the target classifier $\mathcal{F}$. The core idea of this attack, depicted in Figure 1a, is to find a benign script whose AST contains a subtree which is isomorphic to the AST of M, and replace such AST with that of M.

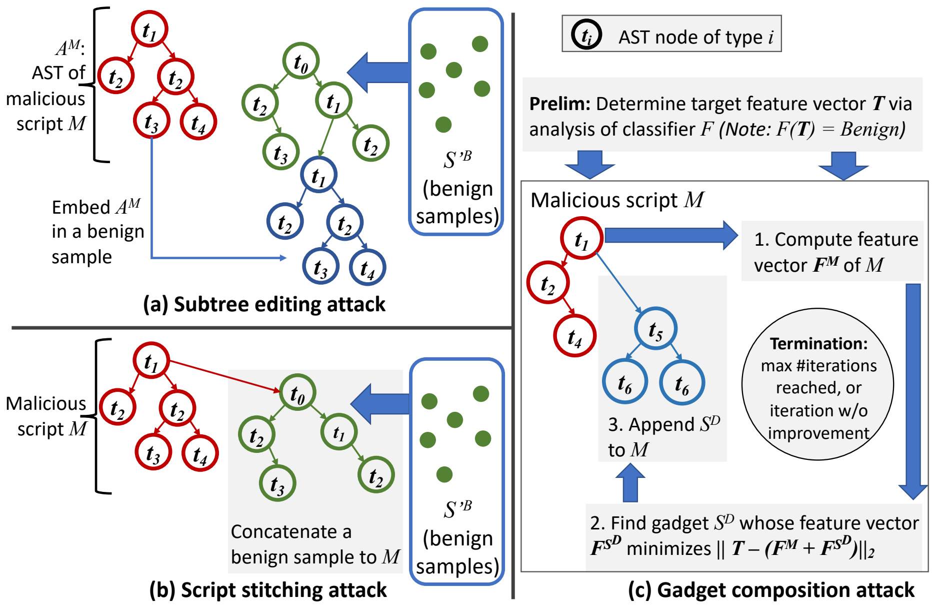

A^M: AST of malicious script M
Embed A^M in a benign sample
S'^B (benign samples)
(a) Subtree editing attack
Malicious script M
Concatenate a benign sample to M
S'^B (benign samples)
(b) Script stitching attack
AST node of type i
Prelim: Determine target feature vector T via analysis of classifier F (Note: $F(T)$ = Benign)
Malicious script M
1. Compute feature vector F^M of M
Termination: max #iterations reached, or iteration w/o improvement
3. Append S^D to M
2. Find gadget S^D whose feature vector F^{S^D} minimizes $\|T-(F^M+F^{S^D})\|_2$
(c) Gadget composition attack

Fig. 1: Attacks discussed in this paper

More formally, let $\mathcal{S}'^B = \{S_1^B, S_2^B, \ldots, S_n^B\}$ be the set of benign scripts present upon a target website, where each S_i^B induces AST A_i. The attack seeks to inject a malicious script S^M onto the page with an AST indistinguishable from some A_j. If the attack succeeds, then the matching defense must either mark both S^M and S_j^B as benign, or mark both S^M and S_j^B as malicious. The attack takes part in two phases:

- **Phase I:** Let M be a JavaScript snippet that realizes the attacker's goal, and let A^M be the AST for M. Note that M need not be a self-contained JavaScript program. Find a benign script $S_j^B \in \mathcal{S}'^B$ s.t. its AST A_j is isomorphic to A^M. In the most basic formulation of the attack, A^M needs only exhibit a subtree isomorphism to a subtree K of S_i^B. However, the search for a satisfying K can be subject to additional constraints if the web defense mechanism employs extra attributes in script matching. For example, the search for K may also require that nodes of K exhibit the same AST type as A^M. In practice, requiring ASTs A^M and A_j to be isomorphic may make it impossible to find viable benign host scripts. For that reason, our approach allows benign candidates that are similar to, but not exactly isomorphic to A^M. This is accomplished by requiring that, for a viable candidate benign AST A_j, $TED(A_j, A^M) < DT$, where TED computes the tree-edit distance, and DT is an arbitrary threshold.
- **Phase II:** Create a new script S^M by replacing the code represented by K in S_j^B by M.

While we were completing our analysis of subtree editing , we became aware of HideNoSeek, a similar attack due to Fass et al. [8]. The two algorithms are based on the same principle (replacing subsections of the AST), although they differ significantly in their approach. Differently from the attack described here, HideNoSeek requires exact matching but can perform swaps against non-contiguous sequences of statements. Given these considerations, we believe high-level conclusions drawn from the analysis of our attack qualitatively apply to HideNoSeek too. We defer a full quantitative evaluation to future work.

4.2 Script Stitching Mimicry Attack

While subtree editing generates scripts that are, feature-wise, virtually indistinguishable from benign ones, it relies upon the existence of a suitable benign host script within a targeted website, which may not always exist. In practice, we found that for certain classifiers simpler forms of mimicry, which relax the reliance on the existence of suitable host scripts, are sufficient.

Like the subtree editing attack, the *script stitching* attack (summarized in Figure 1b), assumes partial knowledge of the training dataset $\mathcal{D}$. In particular, it is sufficient to have access to a subset $\mathcal{S}'^B \subseteq \mathcal{S}^B$ of benign samples used for training (which in the domain of interest can typically be obtained by crawling the target website).

Let $\mathcal{S}'^B = \{S_1^B, S_2^B, \ldots, S_n^B\} \in \mathcal{S}^B$ be the set of benign scripts available to the attacker, and M a JavaScript snippet that realizes the attacker's goal. In a script stitching attack, the attacker randomly select a benign script $S_C^B \in \mathcal{S}'^B$ and generate an adversarial sample $S^M = M \cdot S_C^B$, i.e. she concatenates the selected benign script to the malicious code. Although this attack is simple, in Section 6 we show that a script generated with this approach can bypass a state-of-the-art classifier.

4.3 Gadget Composition Mimicry Attack

One limitation of script stitching is that it does not provide guidance to the attacker in selecting benign scripts in a manner that maximizes the probability of success. This limitation is unavoidable if the attacker does not have knowledge of the classifier model $\mathcal{F}$. However, in some cases it may be possible to gain such knowledge, e.g. by training a model on a surrogate dataset. In this case, the model can offer guidance on selecting appropriate transformations to the malicious code M to increase the probability of success. We leverage this insight for the *gadget composition* attack, depicted in Figure 1c. This attack assumes knowledge of the model $\mathcal{F}$, and specifically of (i) ranking of features by their importance (e.g. using Gini importance), and (ii) full or partial knowledge of the values of such features should achieve for a sample to be classified as benign (this can be inferred by repeatedly querying a model, or using model-specific attacks such as the one by Kantchelian et al. [17] for random forests). Assume the attacker has access to the set of N highest-ranked features for $\mathcal{F}$, and define F^S as the vector containing the values of such features for a given script S.

Furthermore, assume the attacker is able to produce a target vector T containing values of features in H that cause a script to be classified as benign with high probability. Finally, assume the attacker has a dictionary $\mathcal{D}$ of *gadgets*, i.e. self-contained snippets of JavaScript code. Given a malicious snippet M, at every iteration, this attack "grows" M by appending the gadget $S^D \in \mathcal{D}$ whose feature vector F^{S^D} minimizes $||F^B - (F^M + F^{S^D})||_2$. In other words, at every iteration the attack chooses the gadget S^D that brings the concatenation $M \cdot S^D$ closest to the target T (according to the Euclidean distance), and then updates M to $M \cdot S^D$. Note that $\mathcal{D}$ does not have to be limited to the corpus $\mathcal{S}^B$ of benign scripts for the domain, and indeed does not even have to contain any such scripts. This makes the attack also viable for the case where the attacker has access to the target model $\mathcal{F}$ but not the training dataset $\mathcal{D}$ (which is only relevant from a theoretical point of view, because in our domain of interest building $\mathcal{D}$ can trivially be achieved by crawling the victim website.)

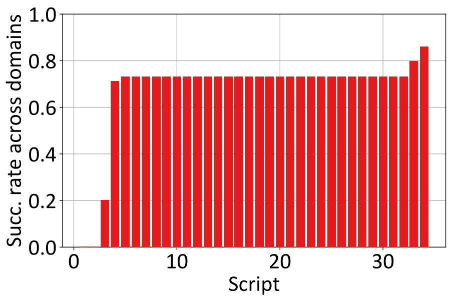
4.4 Correctness

An important question is whether the generated adversarial code successfully executes the exploit. Script stitching concatenates two valid programs, generating code which is syntactically correct by construction. Similarly, gadget composition grows a script by adding syntactically correct program snippets. We empirically verified syntactic correctness by parsing each stitched script; this resulted in only a handful of corner cases being discarded. We note that both operations may still introduce semantic inconsistencies (e.g., variable aliasing); however, due to the dynamic nature of JavaScript interpretation, any such issue will cause an error *after* the malicious code has been executed. For subtree editing, we parse each generated script to ensure syntactic correctness. We also manually checked a number of generated samples and determined that the script would indeed execute up to the entrypoint of the malicious code. We note that all methods can generate multiple adversarial samples from a single malicious one, maximizing the probability of obtaining one or more working scripts.

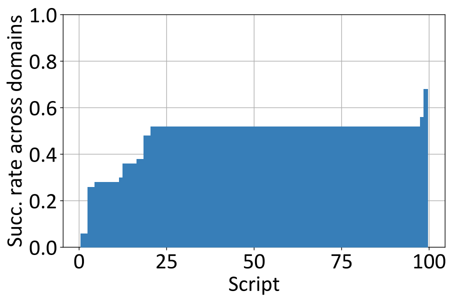
The only tricky point concerns the generation of the gadget dictionary $\mathcal{D}$ for the gadget composition attack. In our experiments, we generated such dictionary by randomly extracting 250,000 AST subtrees, ranging in size from 1 to 100 nodes, from our script dataset (ref. Section 6). To reduce the chance of breaking JavaScript semantic by concatenating malformed gadgets, when mining gadgets the algorithm only considers AST subtrees whose root node can appear as child of the AST root node type. The set of suitable AST node types is automatically learned by analyzing the structure of scripts in the JavaScript dataset.

5 Implementation

In order to generate the script corpus used for our evaluation (Section 6), we adapted the web scraper open-sourced by the CSPAutoGen project [19]. This scraper crawls a list of target websites, saving extracted JavaScript snippets



(a) Script stitching attack



(b) Gadget composition attack

Fig. 2: Success rate of each script against JaSt for different attacks

into a MongoDB instance. For convenience, we used the same approach to store adversarial scripts generated by our attacks; all of our attack tools store their output as MongoDB collections.

The components implementing our subtree editing attack (ref. Section 4.1) and the dictionary generation for our gadget composition attack (Section 4.3) were implemented as a set of Java classes totaling 7222 lines of code. This codebase also includes a suite of AST analysis tools by Falleri et al. [7], which we used for their efficient implementation of tree-edit distance computation. Internally, our program analysis routines leverage Java's Nashorn JavaScript parser. Interestingly, Nashorn defines a set of AST node types which is both quantitatively and qualitatively different from that defined by Esprima, which is the parser used by the JaSt classifier for feature computation. However, we found that this fact does not prevent isomorphic scripts generated by our tool to be effective against JaSt. This observation suggests that the effectiveness of this attack may not depend on the specific choice of AST-based features, instead reflecting an inherent fragility of such features in general. The script-stitching attack (Section 4.2) and gadget composition attacks were implemented as suite of Python tools totaling ∼300 lines of code.

In order to evaluate the performance of CSPAutoGen [19] and JaSt [9] in identifying malicious JavaScript code, we used the respective implementations open-sourced by their authors.

6 Experimental Evaluation

In this section we evaluate the robustness of two state-of-the-art techniques for malicious JavaScript detection, CSPAutoGen [19] and JaSt [9]. We aim to answering the following questions:

1. **Are the attacks identified in this paper effective in generating adversarial scripts?** Section 6.3 shows that subtree editing, script stitching,

Dataset	#Domains	#Scripts	Avg script len [char]
Benign	306	35632	2763
Malicious	N/A	1327	5068

Table 1: JavaScript dataset summary

and gadget composition can generate adversarial scripts that go undetected by JaSt and CSPAutoGen between 3% and 47% of the cases, depending on the combination of classifier and attack method.

2. **What is the degree to which different domains are vulnerable to adversarial JavaScript samples?** Our analysis in Section 6.4 shows that a per-domain JaSt classifier would be vulnerable to one or more attack script in 96% of the domains under consideration. A CSPAutoGen classifier would be vulnerable for 27% of the domains.
3. **Does knowledge of classifier model lead to the creation of more effective adversarial scripts compared to knowledge of training dataset alone?** Section 6.5 shows that knowledge of the model leads to a small but significant increase in the number of successfully obfuscated scripts (45 additional adversarial variants across 50 domains).

6.1 Dataset and Infrastructure

To extensively evaluate the robustness of the target classifiers, we randomly selected 300 websites from Alexa's Top-500 and crawled them as described in Section 5. To ensure a realistic corpus of malicious JavaScript snippets, we used a publicly-available collection by the security collective GeeksOnSecurity [1]. Both datasets (benign and malicious) are summarized in Table 1. Our tooling and implementation are described in Section 5. We ran all experiments on a MacBook Pro laptop with a quad-core 2.6Ghz Intel Core i7 CPU, 16GB of RAM, and 500GB of storage. The laptop runs MacOS Mojave 10.14.

6.2 Baseline Classifier Performance

JaSt: In order to evaluate the best-case performance of JaSt in the absence of adaptive adversarial samples, we trained it on 303 domains from our dataset (the remaining 3 domains triggered bugs in the classifier code which prevented their evaluation). In each instance, we used all the scripts scraped from a domain as the benign dataset, and our collection of malicious scripts as the malicious dataset. We first used the same dataset for training and evaluation, asking JaSt to re-label each script on which it was trained. JaSt performs consistently well in this experiment, achieving on average **0% false negatives (evasion rate)** and a **2.3% false positives (false alarm rate)** across all domains.

Since assuming that JaSt has access of the entirety of both the benign and malicious script sets of interest is unrealistic, we also performed a round of experiments where we split the dataset in disjoint training and evaluation subsets using 10-fold cross validation. This results in a **0.5% evasion rate** and a **6% false alarm rate**.

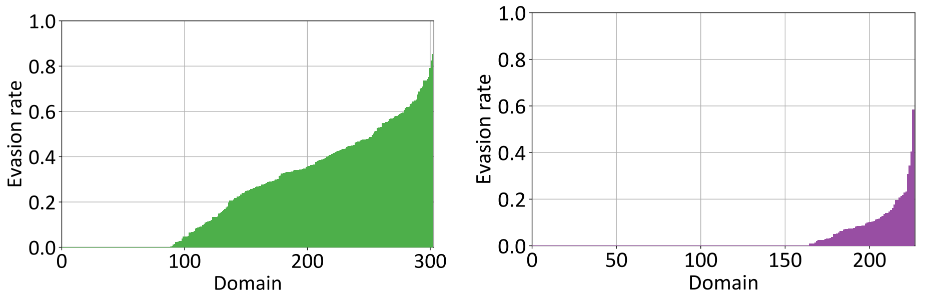
CSPAutoGen: We generated CSPAutoGen templates for each of 227 domains. We could not test the remaining 79, as the current implementation of CSPAutoGen crashes while processing their scripts. Since CSPAutoGen generates templates using benign scripts exclusively, no malicious script was used for training. We then used CSPAutoGen to analyze each script in the training corpus, and all the scripts in the malicious corpus. CSPAutoGen achieves on average **0.06% evasion rate** and a **1.5% false alarm rate** across all domains. Splitting the dataset in training and evaluation and performing 10-fold cross-validation results in **0.06% evasion rate** and **11% false alarm rate**.

Discussion: Although the evasion rate remains negligibly low for both classifiers, both suffer from a somewhat high false alarm rate. This is a significant hurdle to deployment, as even a handful of false positives may disrupt website functionality. Even when using the same dataset for training and evaluation, we note that both classifiers present occasional significant deviations from the false alarm average. This results in extremely high false alarm rates for some domains. For JaSt, these occurrences mostly occur for pathological domains where the scraper could only successfully extract a few scripts. The most extreme case is `theglobeandmail.com`, which only includes 2 benign scripts, both misclassified, giving a 100% false positive rate. CSPAutoGen occasionally exhibits unacceptably high false positive rate even for domains for which a significant amount of data is available. For example, CSPAutoGen misclassifies 29 of the 70 benign scripts scraped from `360.cn`, yielding a false positive rate of 41%.

6.3 Evaluation of Attacks

Attacks Setup To launch subtree editing and script stitching attacks for a given domain, we simply attempted to combine each malicious script of interest with each benign script in the domain, according to each attack's semantics. This involves attempting to perform a suitable tree-swap in subtree editing - we include each distinct point of the benign script at which an attack can be injected as a separate sample. For script stitching, combination consists of concatenating the given attack script and the target benign one. The tree-edit distance threshold for subtree editing was set to 20.[3] For gadget composition, we mutated each malicious script towards a target feature vector by adding gadgets from the dictionary described in Section 4.4. We generate the target feature vector by extracting the highest-ranked (by feature importance) 35 features from each model, and computing the average value of each feature across the benign scripts for that domain. We note that this is a rather crude way to generate adversarial feature vectors, however it serves as a lower-bound on attack effectiveness.

[3] Experimentally, we determined that increasing the maximum tree-edit distance above 20 results in a sharp increase in the detection rate for the generated samples. This applies to both classifiers in our examination.



(a) Evasion rates on JaSt (b) Evasion rates on CSPAutoGen

Fig. 3: Evasion rates for subtree editing attack

Attacks against JaSt While the performance of our current toolchain is practical for attack generation (see Section 6.7), the unoptimized nature of the code and the file-based interface used for some operations result in high sample generation times for some combinations of attacks/domains. To keep experiment times manageable, we used different randomly selected sets of malicious scripts in each attack: 1291 malicious scripts for subtree editing, 34 for script stitching, and 100 for gadget compositions. For gadget composition, we also limited the number of considered domains to 100. Figure 2 depicts, for each malicious script, the number of domains for which the corresponding JaSt classifier is vulnerable. We consider a domain to be vulnerable to a script if one or more adversarial variants of that script successfully go undetected. Figure 2a depicts the results for script stitching, and Figure 2b for gadget composition. On average, script stitching results in a **15% evasion rate** (considering adversarial scripts only) across domains. Gadget composition achieved **47% evasion rate** across domains.

In practice, the low evasion rate result for script stitching is misleading as the attack generates numerous variants of each malicious script; even if most of those variants are not successful, the attack is still effective when at least one working variant is found. Indeed, on average script stitching can generate at least one successful adversarial variant for **68%** of all considered scripts. The percentage is **46%** for gadget composition (note that gadget composition only generates one variant per script per domain, so this result is identical to the evasion rate above.)

We do not plot per-script results for subtree editing as that attack only generates successful adversarial samples from 8 scripts out of 1291. 7 of these scripts generate one or more samples for 237 (out of 304) domains, and 1 for 236 domains. The set of vulnerable domains is the same for each script. On aggregate, this attack achieves **46% evasion rate** across domains against JaSt. Figure 3a shows evasion rate per domain for subtree editing.

Attacks against CSPAutoGen To analyze the robustness of CSPAutoGen to adversarial samples, we only considered the subtree editing attack. By construction, CSPAutoGen looks at the structure of the AST itself and not at feature counts, therefore feature-based attacks like stitching and gadget composition are largely ineffective (moreover, access to a set of CSPAutoGen templates does not provide useful guidance for the gadget composition attack). The price for this increased robustness is the high false positive rate highlighted in Section 6.2.

Figure 3b shows the evasion rate per domain for this attack. Overall, this attack achieves **3.1% evasion rate** across domains against CSPAutoGen. Note that this value, and the evasion rates shown in Figure 3b, are likely to *underestimate* the attack effectiveness. The implementation of CSPAutoGen appears to have a bug that occasionally results in partial crash and a reported evasion rate of 0% for some domains.

Number of Variants We now consider the overall number of successful variants generated by each method, i.e., the number of malicious scripts that successfully get misclassified as benign by the victim classifier. For the attacks against JaSt, we found, on average **891** working attack variants per domain using subtree editing, **606** using script stitching, and **47** using gadget composition. For CSPAutoGen, subtree editing generated on average **43** successful variants per domain. While the ability to generate even a single working variant per domain is troubling, these numbers indicate a highly-effective set of attacks.

6.4 Per-domain Analysis

Given that the attacks above prove to be effective in a significant percentage of cases, we move to looking at how these attacks affect different domain. We first evaluated JaSt. We only consider the 34 attack scripts and 49 domains on which we executed all attacks in our evaluation. Figure 4a plots the cumulative distribution of the number of adversarial scripts per domain which go undetected by JaSt. In this case, we aggregate scripts produced by all three attack methods. Overall, **47 out of 49** domains (96%) are vulnerable to one or more attacks. The maximum number of working attack scripts per domain is **16515**.

We then performed the same evaluation for CSPAutoGen; in this case we only considered the subtree editing attack. Our data cover 227 domains. Figure 4b plots the cumulative distribution of number of working attack scripts per domain. Overall, **62 out of 227** domains (27%) are vulnerable to at least one attack script. The maximum number of working attack scripts per domain is **702**.

6.5 Knowledge of Dataset vs Model

The final part of our evaluation asks whether knowledge of both the training dataset $\mathcal{D}$ and a victim model $\mathcal{F}$ can provide an attacker with an advantage over knowledge of $\mathcal{D}$ only. For this, we look at our results for both the stitching and the gadget composition attack, considering a set of 34 scripts and 50 domains on which both attacks were executed.

Due to its simplicity, our gadget composition attack is likely to achieve a lower bound for a model-aware attack. In particular, the current version of the algorithm attempts to generate at most one malicious variant per domain; due to this limitation, gadget composition generates *less* working adversarial variants per domain than the simpler script stitching attack[4]. Even with this limitation, we find **45** combinations of script and domain for which stitching could not generate any working adversarial variants, while gadget composition could. We therefore conclude that knowledge of $\mathcal{F}$ can provide an attacker with additional power to generate undetectable script mutations.

6.6 Impact of Adversarial Training

While a full discussion of defenses is outside the scope of this paper, we briefly consider the approach which is most readily available to a defender, which is *adversarial training*. With adversarial training, adversarial scripts generated via an attack method of interest are added back into the training dataset and labeled as malicious. The typical drawback of adversarial training is that it tends to decrease the discerning power of a classifier, increasing the false alarm rate, due to the fact that some malicious samples in the dataset have features are by construction very similar to those of benign samples.

In order to assess the effectiveness of adversarial training, we evaluate JaSt against subtree editing, before and after adding a subset of the generated attack scripts to the training set. Since it is unrealistic to assume that a defender could generate all possible adversarial samples of interest, we used 10-fold cross validation on the dataset.

While adversarial training does allow the classifier to identify most adversarial samples, it also causes the (already high) false alarm rate to double from **6% to 12%**. Since false alarms have the potential to break website functionality (by preventing legitimate scripts from being served), this result suggests that adversarial training cannot protect against this class of attacks.

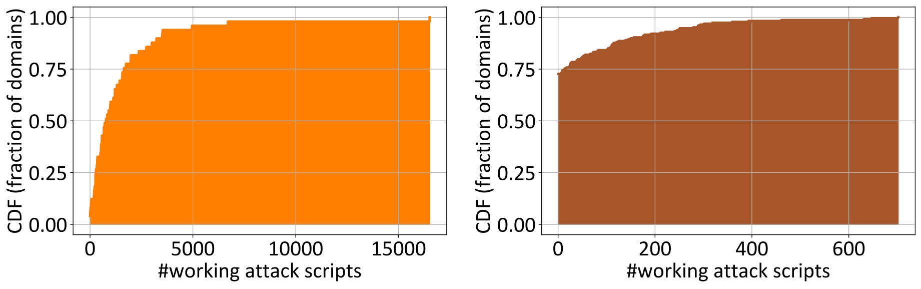
6.7 Execution Times

Our subtree editing can compare a malicious script with a candidate benign host (and embed the malicious script in the benign one, if successful) at a rate of 1 comparison per second. For script stitching, adversarial scripts can be generated at 1 per 5.7 seconds, and for gadget composition, 1 per 57 seconds.

6.8 Analysis of Results

The results in this section highlight two important observations. First, existing state-of-the-art AST-based classifiers are extremely effective in identifying

[4] It is in principle possible to extend the algorithm with backtracking during the gadget search process, enabling it to generate an arbitrary number of variants.



(a) JaSt vs all attacks (b) CSPAutoGen vs subtree editing

Fig. 4: CDF of successful number of attacks per domain

non-adversarial malicious scripts, although they also appear to incur a high false positive rate which may hamper their use in practice. Second, the same classifiers appear to be relatively fragile to low-complexity adversarial attacks. The considered attacks were able to generate tens to hundreds of successful adversarial scripts per domain, without requiring any customization of the source code of the seed scripts. There are also specific findings related to the type of classifier:

Structural Classifiers As we show in Section 6, mimicry of structural properties of benign scripts can frequently be achieved on a variety of sites. Through the use of our subtree editing attack, we show that 27% of sites can admit a structural replacement that is undetectable by our representative structural classifier. We also note that the only successful attack against the structural classifier was the subtree editing attack, which uses the threat model of the reconnaissance attacker, which relies on a weak set of capabilities. Indeed, note that denying the attacker the capabilities requires restricting the access of a visitor to benign scripts on a page, which is likely to break the public functionality of the site.

Feature-Based Classifiers In the conceptual framework proposed by Maiorca et al. [18], classifiers may exhibit *learning vulnerability*, *feature vulnerability*, or both. Learning vulnerabilities stem from artifacts of the training process: if the region of feature space mapped to benign classes is unnecessarily large, an attacker has significant freedom in generating malicious samples. However, this vulnerability can generally be remediated by adversarial training (which causes the boundaries of benign classes to tighten around actual practical benign samples). Feature vulnerability instead derives from the classifier features being intrinsically unable to separate benign and malicious classes; i.e., it is possible for an attacker to generate malicious samples that are, for the classifier purpose, indistinguishable from benign ones. This second class of vulnerabilities cannot be prevented without changing the features themselves. Results in Section 6.6 suggest that the feature-based classifier under consideration may suffer, at least in part, of the latter type of vulnerability (i.e., there exist some attack scripts which are feature-wise indistinguishable from benign ones).

7 Related Work

To our knowledge, our work is the first to systematically build and apply a system of threat models against JavaScript syntactic detectors. We also believe that our attacks constitute novel techniques to cause misclassification by existing JavaScript detectors. In this section, we discuss previous work that is similar in spirit or approach to various aspects of this paper.

Malicious JavaScript Classifiers. Our work is inspired by the proliferation of tools to identify malicious JavaScript. While we focus on [19, 9], we use these tools as representative of a broad range of previous work in the area. Some related tools include the feature-based Bayesian analysis classifier [4, 13], which operate over features of the AST or lexical tokens. Cujo [20] uses Support Vector Machines (SVM) trained over static script features. Synode [24] is another template-based JavaScript security tool, though it is intended to work server-side to identify malicious NPM packages. Although Synode is a multi-component system for defending from injections, it does use a template-based mechanism to restrict what code can be run by injection. Although we note that the SyNode templates may fall victim to the same attacks we describe, defeating other components (such as script injection sanitization) is out of scope for this work.

Ultimately, we focused on CSPAutoGen and JaSt because they represent the most recent and distinct work. We believe that our results are likely to translate to these other systems, though we leave the analysis to future work.

Mimicry Attacks. Attempts to fool automatic security classification tools are by no means new. Wagner et al. introduced the term mimicry attacks to describe an intrusion that is obfuscated in order to avoid detection [26] and automatic mimicry attacks have also been used to mask malicious system call sequences [11]. Conceptually-related attacks have also been demonstrated against PDF [23] and Flash-based [18] classifiers. Ersan et al. [6] evaluate AST-based evasion of detectors for HTML-based malware; however their approach by design does not guarantee that semantic correctness of malicious code is preserved.

Finally, HideNoSeek by Fass et al. [8] is a work of which we became aware immediately after it was published, and subsequently after the development of our attacks. The HideNoSeek attacker fits within the general framework of our reconnaissance attacker (ref. Section 3), but their paper does not consider the other threat models. The HideNoSeek attack itself is conceptually similar to the subtree-editing attack described here (the differences are discussed in Section 4.1). A preliminary evaluation of HideNoSeek on a subset of our data suggests that their attack has a higher success rate than subtree editing (on a set of 100 randomly-selected malicious/benign scripts pairs, HideNoSeek found one or more embeddings in 66 cases), at the price of a higher execution time (2 orders of magnitude slower than subtree editing, with execution times affected by script size; we discarded one additional pair on which HideNoSeek executed for 17 hours without terminating). This is consistent with the designs of both algorithms: HideNoSeek can split a malicious AST into subtrees and embed each separately; this relaxes the constraints on the candidate benign host but increases complexity and computation time. Conversely, subtree editing needs to find a near-exact AST match for the malicious script, but can leverage decades of research in efficient tree-edit distance algorithms to quickly find a candidate. **Adversarial Machine Learning.** Related work in AML has already shown the fragility of program classifiers in other domains [12, 5, 18, 15]. However, features in these work typically consist of presence/absence of context-independent entities—such as specific system calls [15] or entries in an application's manifest [12]. In these cases, there tend to be trivial mappings between an adversarial feature vector and a concrete attack program (e.g., add appropriate entries to a malicious manifest to masquerade it as a benign one). In our case, features include presence or absence of specific subtrees in a program's abstract syntax tree. When restructuring a program to alter its features, the set of transformations is bound by the requirement that the resulting AST must be correct. For example, it is not possible to alter features by appending a function subtree to a integer variable, as the latter is bound to appear exclusively as leaf node. Our solution is to ensure that the procedure used to generate candidate adversarial programs guarantees correctness by construction.

Finally, there exist a number of works that focus on altering *dynamic* program behavior in order to evade classifiers [14, 22, 21]. None of these works focus on JavaScript, and they are orthogonal to the goal of this paper.

8 Conclusion

In this work, we show that adversaries can leverage their knowledge of the target to better disguise malicious JavaScript code as benign, and propose a framework of threat models to capture different real-world adversarial capabilities. We believe that this work makes an important contribution in motivating the creation of new defensive measures and best practices for classifiers in realistic settings.

9 Acknowledgments

We thank the anonymous reviewers and our shepherd, Yuan Zhang, for their insightful comments. We further thank: Louis Narmour and Devin Dennis for their help in building infrastructure and dataset; Bruce Kapron and Somesh Jha for informative early discussions on the problems tackled in this paper.

References

1. GitHub - geeksonsecurity/js-malicious-dataset (Dec 2019), https://github.com/geeksonsecurity/js-malicious-dataset
2. Biggio, B., Roli, F.: Wild patterns: Ten years after the rise of adversarial machine learning. Pattern Recognition **84**, 317–331 (Dec 2018)
3. Calzavara, S., Rabitti, A., Bugliesi, M.: Content Security Problems?: Evaluating the Effectiveness of Content Security Policy in the Wild. In: CCS (2016)
4. Curtsinger, C., Livshits, B., Zorn, B.G., Seifert, C.: Zozzle: Fast and precise in-browser javascript malware detection. In: USENIX Security Symposium (2011)

5. Demontis, A., Melis, M., Biggio, B., Maiorca, D., Arp, D., Rieck, K., Corona, I., Giacinto, G., Roli, F.: Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection. IEEE Transactions on Dependable and Secure Computing pp. 1–1 (2018)
6. Ersan, E., Malka, L., Kapron, B.M.: Semantically non-preserving transformations for antivirus evaluation. In: FPS (2016)
7. Falleri, J.R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: ASE (2014)
8. Fass, A., Backes, M., Stock, B.: HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In: CCS (2019)
9. Fass, A., Krawczyk, R.P., Backes, M., Stock, B.: JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In: DIMVA (2018)
10. Fredrikson, M., Jha, S., Ristenpart, T.: Model inversion attacks that exploit confidence information and basic countermeasures. In: CCS (2015)
11. Giffin, J.T., Jha, S., Miller, B.P.: Automated discovery of mimicry attacks. In: RAID (2006)
12. Grosse, K., Papernot, N., Manoharan, P., Backes, M., McDaniel, P.: Adversarial Examples for Malware Detection. In: ESORICS (2017)
13. Hao, Y., Liang, H., Zhang, D., Zhao, Q., Cui, B.: Javascript malicious codes analysis based on naive bayes classification. In: International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (2014)
14. Hu, W., Tan, Y.: Black-Box Attacks against RNN based Malware Detection Algorithms. arXiv:1705.08131 [cs] (May 2017), http://arxiv.org/abs/1705.08131
15. Hu, W., Tan, Y.: Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN. arXiv:1702.05983 [cs] (Feb 2017)
16. John Leyden: Payment-card-skimming Magecart strikes again: Zero out of five for infecting e-retail sites (Oct 2018), https://www.theregister.com/2018/10/09/magecart_payment_card_malware/
17. Kantchelian, A., Tygar, J.D., Joseph, A.D.: Evasion and hardening of tree ensemble classifiers. In: ICML (2016)
18. Maiorca, D., Biggio, B., Chiappe, M.E., Giacinto, G.: Adversarial Detection of Flash Malware: Limitations and Open Issues. arXiv:1710.10225 [cs] (Oct 2017)
19. Pan, X., Cao, Y., Liu, S., Zhou, Y., Chen, Y., Zhou, T.: CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites. In: CCS (2016)
20. Rieck, K., Krueger, T., Dewald, A.: Cujo: efficient detection and prevention of drive-by-download attacks. In: ACSAC (2010)
21. Rosenberg, I., Shabtai, A., Elovici, Y., Rokach, L.: Query-Efficient GAN Based Black-Box Attack Against Sequence Based Machine and Deep Learning Classifiers. arXiv:1804.08778 [cs] (Apr 2018), http://arxiv.org/abs/1804.08778
22. Rosenberg, I., Shabtai, A., Rokach, L., Elovici, Y.: Generic Black-Box End-to-End Attack Against State of the Art API Call Based Malware Classifiers. In: RAID (2018)
23. Srndic, N., Laskov, P.: Practical Evasion of a Learning-Based Classifier: A Case Study. In: IEEE S&P (2014)
24. Staicu, C.A., Pradel, M., Livshits, B.: Synode: Understanding and automatically preventing injection attacks on node. js. In: NDSS (2018)
25. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I.J., Fergus, R.: Intriguing properties of neural networks. In: ICLR (2014)
26. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: CCS (2002)