Lecture #11: Adversarial Attacks #1

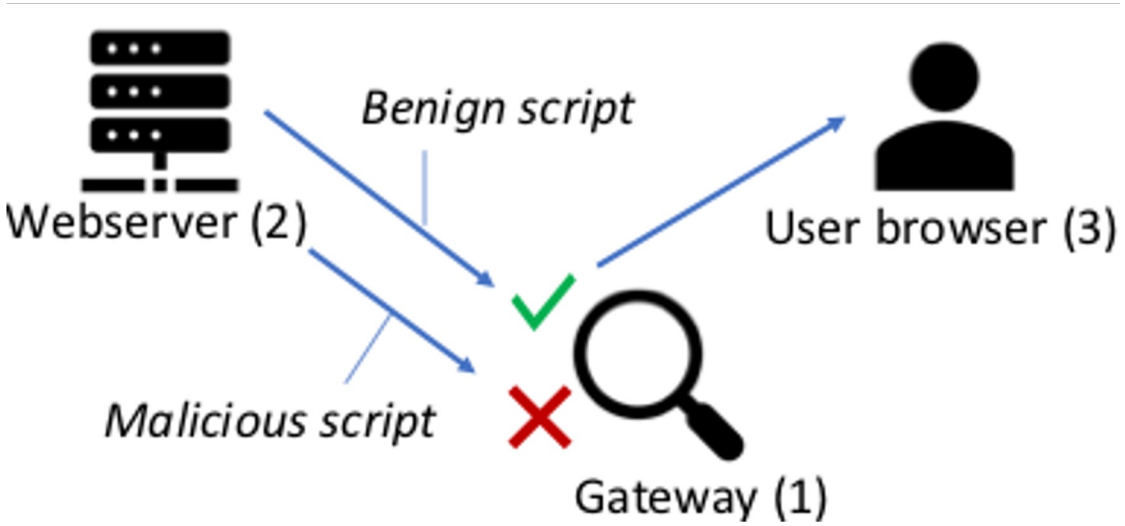
UCalgary ENSF619

Elements of Software Security

Instructor: Lorenzo De Carli (lorenzo.decarli@ucalgary.ca)

Scenario of interest

Server-side script filtering: a gateway is placed between a JavaScript source and the user. The gateway identifies malicious scripts that are not part of the application and sanitizes the pages by removing them/

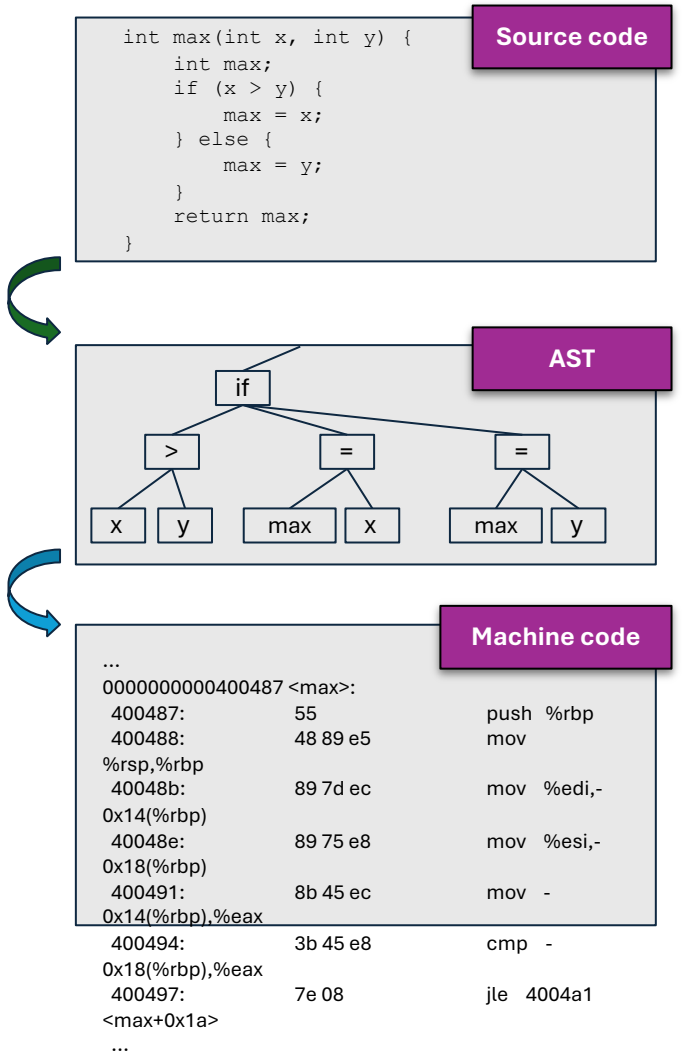


Server-side script filtering

- Various approaches (I'll talk about two), but generally works by:
  1. Learning a **model** of how benign scripts served from a web application should look like
  2. **Analyzing each script served to the client** before it is transmitted – if the classifier decides the script is malicious, it blocks the transmission
- **Script model:** based on **syntactic features** of the code extracted from its **abstract syntax tree**

Abstract Syntax Tree (AST)

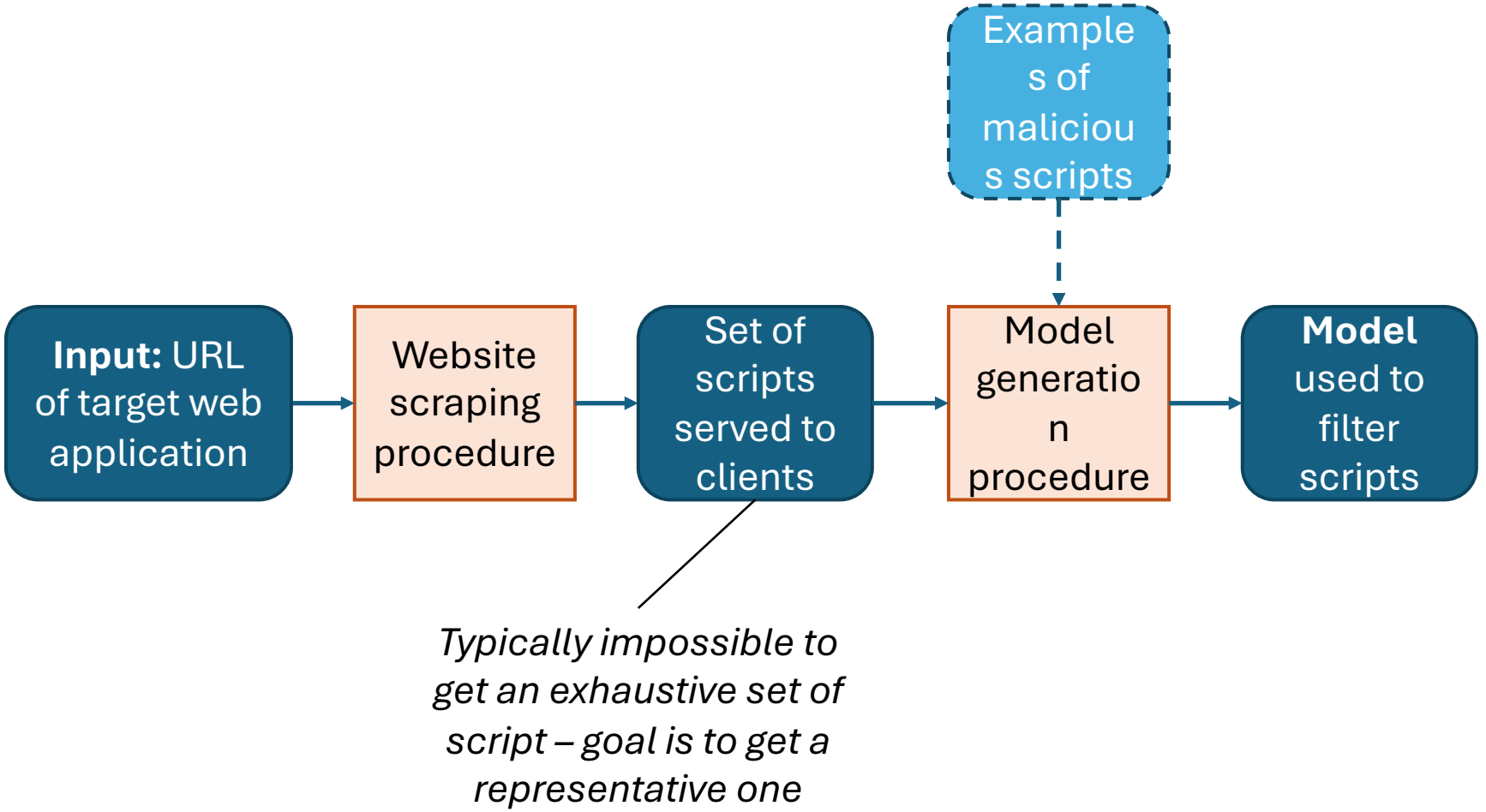
- Computers can't execute raw source code
- Source code must be translated to some form of machine code
- ASTs are an **intermediate representation** during translation
- Full translation process is called compilation



Abstract Syntax Tree (AST) - II

- ASTs “fingerprint” the program at a **desirable granularity**:
  - Provides **high-level program behavior** encoding
  - Abstracts away **unnecessary details** (e.g., comments, formatting)
- Similar code lead to similar ASTs

General defense workflow



Example #1: CSPAutoGen

CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites

Xiang Pan[1] Yinzhi Cao[2] Shuangping Liu[1] Yu Zhou [1] Yan Chen[3,1] Tingzhe Zhou[2]
[1]Northwestern University, Illinois, USA
[2]Lehigh University, Pennsylvania, USA
[3]Zhejiang University, Zhejiang, China
{xiangpan2011, shuangping-liu, yuzhou2016}@u.northwestern.edu
{yinzhi.cao, tiz214}@lehigh.edu ychen@northwestern.edu

ABSTRACT

Content security policy (CSP)—which has been standardized by W3C and adopted by all major commercial browsers—is one of the most promising approaches for defending against cross-site scripting (XSS) attacks. Although client-side adoption of CSP is successful, server-side adoption is far behind the client side: according to a large-scale survey, less than 0.002% of Alexa Top 1M websites enabled CSP.

To facilitate the adoption of CSP, we propose *CSPAutoGen* to enable CSP in real-time, without server modifications, and being compatible with real-world websites. Specifically, *CSPAutoGen* trains so-called templates for each domain, generates CSPs based on the templates, rewrites incoming webpages on the fly to apply those generated CSPs, and then serves those rewritten webpages to client browsers. *CSPAutoGen* is designed to automatically enforce the most secure and strict version of CSP without enabling "unsafe-inline" and "unsafe-eval", i.e., *CSPAutoGen* can handle all the inline and dynamic scripts.

We have implemented a prototype of *CSPAutoGen*, and our evaluation shows that *CSPAutoGen* can correctly render all the Alexa Top 50 websites. Moreover, we conduct extensive case studies on five popular websites, indicating that *CSPAutoGen* can preserve the behind-the-login functionalities, such as sending emails and posting comments. Our security analysis shows that *CSPAutoGen* is able to defend against all the tested real-world XSS attacks.

1. INTRODUCTION

Cross-site scripting (XSS) vulnerabilities—though being there for more than ten years—are still one of the most commonly found web application vulnerabilities in the wild. Towards this end, researchers have proposed numerous defense mechanisms [12, 14, 17, 21, 30, 32, 40, 41] targeting various categories of XSS vulnerabilities. Among these defenses, one widely-adopted approach is called Content Security Policy (CSP) [41], which has been standardized by W3C [1] and adopted by all major commercial browsers, such as Google Chrome, Internet Explorer, Safari, and Firefox.

Though client-side adoption has been successful, server-side adoption of CSP proves more worrisome: according to an Internet-scale survey [45] of 1M websites, at the time of the study, only 2% of top 100 Alexa websites enabled CSP, and 0.00086% of 900,000 least popular sites did so. Such low adoption rate of CSP in modern websites is because CSP[1] requires server modifications. That is, all the inline JavaScript and `eval` statements need to be removed from a website without breaking its intended functionality, which brings extensive overhead for website developers or administrators.

To facilitate server deployment, in related work, deDacota [12] and AutoCSP [14] analyze server-side code using program analysis, infer CSPs, and modify those code to enable the inferred CSPs. Another related work, autoCSP[2] [17], infers CSPs based on violation reports and enforces the inferred CSPs later on. However, deDacota and AutoCSP—due to their white-box property—require server modification. Additionally, both approaches are specific to websites written in certain web languages. Another approach, autoCSP, does not support inline scripts with runtime information and dynamic scripts, and thus websites with those scripts cannot work properly. According to our manual analysis, 88% of Alexa Top 50 websites contain such script usages.

In this paper, we propose *CSPAutoGen*, a real-time, black-box enforcement of CSP without any server modifications and being compatible with real-world websites. The key insight is that although web scripts may appear in different formats or change in runtime, they are generated from uniform templates. Therefore, *CSPAutoGen* can infer the templates behind web scripts and decouple web contents in a script from the script's inherent structure.

Specifically, *CSPAutoGen* first groups scripts in webpages under a domain, and infers script templates, defined as training phase. Next, in the so-called rewriting phase, *CSPAutoGen* generates CSPs based on the webpage and templates, and then modifies webpages on the fly—which could be at a server gateway, an enterprise gateway or a client browser—to insert the generated CSPs and apply them at client browsers. Lastly, a client-side library added in the rewriting phase will detect additional scripts generated at client side during runtime and execute these scripts that match the templates. Below we discuss two important mechanisms used in *CSPAutoGen*:

CSPAutoGen – Overview

- **Goal:** prevent unknown/malicious code to be served by a given web application
- **Approach:**
  1. Intercept web pages before they reach the client; match JavaScripts against templates
  2. Copy all matching JavaScripts to a trusted domain and rewrite HTML page to include them
  3. Add a CSP to the page only allowing to run JavaScripts from trusted domains

CSPAutoGen - Templates

- Before deployment, CSPAutoGen undergoes a **training phase**
  - Website to protect is scraped to determine all benign snippets of JavaScript code
- The output of the training phase is a **set of templates** describing all possible forms that legitimate JavaScript may take

CSPAutoGen – Templates/2

- **Template generation** procedure:
  1. Extract a large number of JavaScript programs from web application
  2. Generate ASTs for each program
  3. Generalize each AST variable **replacing it with its type**
     1. Use **rule-based type-inference algorithm** (nodes w/ only one value are labeled CONST; nodes with a restricted set of values are labeled ENUM; etc.)
- Generalized ASTs (gASTs) are used as templates and matched w/ gASTs of served scripts

CSPAutoGen - observation

- Uses **domain specific learning algorithm** (template learning) as opposed to a generic one
  - Motivated by **keeping false positives low**
  - However, it generates **false negatives!**
- Use entire (generalized) AST as the single feature
- Requires exact matching (no notion of distance threshold)

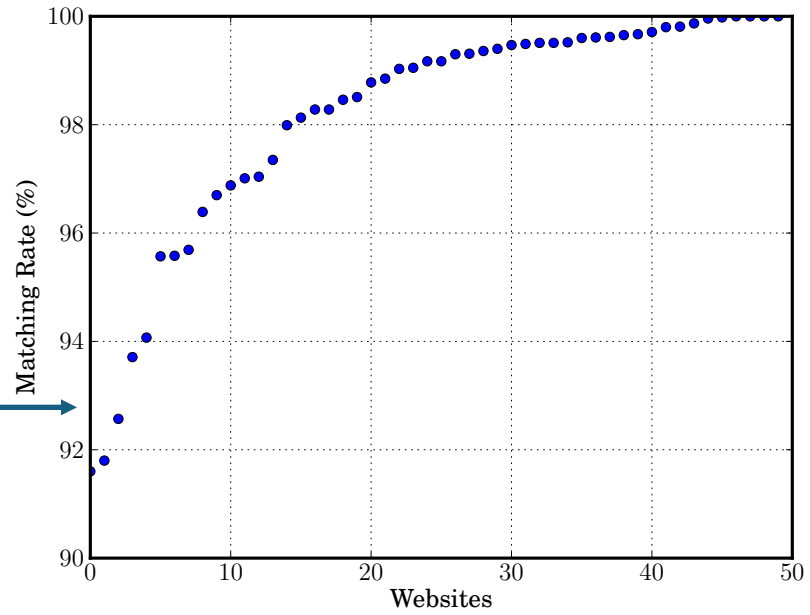


Figure 3: Template Matching Rate (Medium Rate 99.2%).

Example #2: JaST

JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript

Aurore Fass[1(✉)], Robert P. Krawczyk[2], Michael Backes[3], and Ben Stock[3]

[1] CISPA, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
aurore.fass@cispa.saarland
[2] German Federal Office for Information Security (BSI), Bonn, Germany
robert.krawczyk@bsi.bund.de
[3] CISPA Helmholtz Center i.G., Saarland Informatics Campus, Saarbrücken, Germany
{backes,stock}@cispa.saarland

Abstract. JavaScript is a browser scripting language initially created to enhance the interactivity of web sites and to improve their user-friendliness. However, as it offloads the work to the user's browser, it can be used to engage in malicious activities such as Crypto Mining, Drive-by Download attacks, or redirections to web sites hosting malicious software. Given the prevalence of such nefarious scripts, the anti-virus industry has increased the focus on their detection. The attackers, in turn, make increasing use of obfuscation techniques, so as to hinder analysis and the creation of corresponding signatures. Yet these malicious samples share syntactic similarities at an abstract level, which enables to bypass obfuscation and detect even unknown malware variants.

In this paper, we present JaSt, a low-overhead solution that combines the extraction of features from the abstract syntax tree with a random forest classifier to detect malicious JavaScript instances. It is based on a frequency analysis of specific patterns, which are either predictive of benign or of malicious samples. Even though the analysis is entirely static, it yields a high detection accuracy of almost 99.5% and has a low false-negative rate of 0.54%.

1 Introduction

Information Technology is constantly under threat with the amount of newly found malware increasing permanently: over 250,000 new malicious programs are registered every day [11]. Moreover, our Internet-driven world enables malware to rapidly infect victims everywhere, anytime (e.g., Mirai [26], NotPetya [27]),

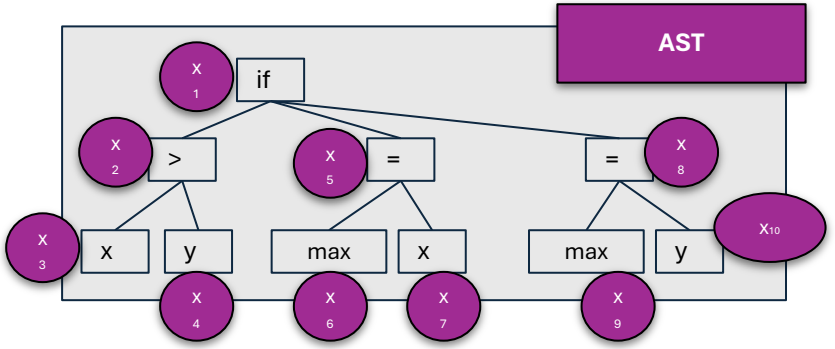
JaST overview

- **Goal:** programmatically learn to distinguish between malicious and benign JavaScript programs
  - Not just for web JS! (in fact, paper mostly looks at emails, applications, etc.)
- **Approach:**
  1. Extract **n-gram-based AST features** for a corpus of training samples (both malicious & benign)
  2. Train a **random forest classifier** to distinguish between the two classes
  3. Apply the classifier to unseen samples

JaST feature generation

- ASTs need to be summarized as feature vectors to be fed into a learning algorithm
- Features should be computable efficiently but still encode relevant aspects of the AST
- JaST uses **n-gram frequencies**

N-gram Extraction



- Given a set of possible AST node labels, S = {s_1, … , s_k}
- For a particular AST, **traverse (preorder here) and record the labels encountered**, call this list X
  - The set of unique labels in X is a subset of S
- Example: the set of bigrams (2-grams), B, over the list X is:
  - B = { (x_i, x_{i+1}) | for 0 < i < |X| }
  - B is a subset of S^2
- In our example (shown on top right):
  - `S = {'expr', 'func', 'var', 'block', 'if', '>', '<',…}`
  - `X = ['if', '>', 'var', 'var', '=', …, 'var']`
  - `B = {'(if,>)', '(>,var)', '(var,var)', '(var,=)', '(=,var)'}`
- Next - use S, X, and B to construct the feature vector.

N-gram Extraction

- If only using bigrams then feature vector V has d = $|S|^2$ components
  - For generic n-grams, d = $|S|^n$
- The value of the i^{th} component of V is the number of times the i^{th} n-gram appeared in the AST
- Feature vectors based on n-grams are typically sparse (i.e. many components are 0) since many n-grams are not encountered while traversing the AST
- **V represents our AST as a point in d-dimensional feature space**
- Feature reduction
  - Not all features are actually needed to attain a sufficient model
  - **Information-gain filtering** keeps only features with the most "discerning power" (i.e. splits the data most effectively)

JaST - observations

- Generally, very high TPR/TNR

Table 3. Detection accuracy of JaSt

JS type	#Misclassified	#Correctly classified	Detection accuracy
Emails	443	81,116	99.46%
Microsoft	71	14,097	99.50%
Games	10	1,997	99.52%
Web frameworks	4	430	99.03%
Atom	1	136	98.98%
Average benign	86	16,660	99.48%

TPR:
99.46%

Other approaches and applications

- ZOZZLE (Curtsinger et al., USENIX Security 2011) uses a generalization of n-grams (**string in AST node+context** where the string appears) and a **naïve Bayes classifier** to distinguish benign & malicious javascripts
- Revolver (Kapravelos et al., USENIX Security 2013) uses **AST similarities** between scripts to **identify source-code-level obfuscation**

Other approaches and applications/2

- **Code Stylometry** (Caliskan-Islam et al., USENIX Security 2015) uses AST-based features and random forest to **identify programmers from source code**
  - Node depths, unigram and bigram frequencies
  - Uses information gain to reduce the set of features
  - Pretty effective!

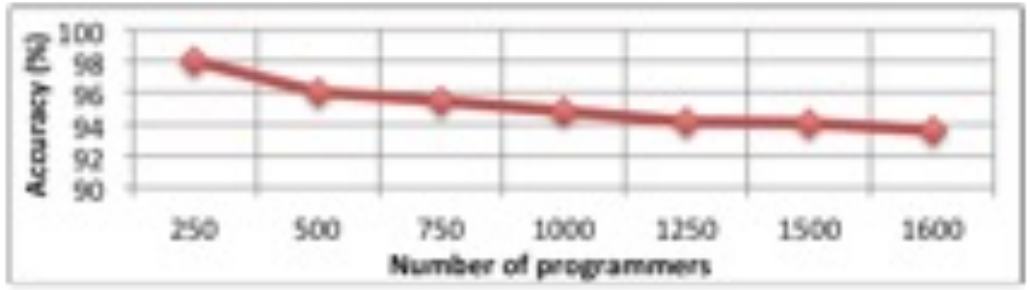


Figure 3: Large Scale De-anonymization

The problem with AST-based features

- The **structure of the AST** of a program is obviously correlated with its **functionality** and **goals**
  - But only **loosely** correlated!
- Many of the techniques outlined above rely on the fact that malicious JavaScript code “tend to look different”, at the AST level, from benign JavaScript code
- However, given a particular computation that an attacker wishes to perform, there are in general **multiple ASTs** describing that computation

Enter… **adversarial machine learning**

- Large (and active) area of research, includes many different problems
- The general research question:
  - given a classifier trained to identify different object classes…
  - It is possible to **cause the classifier to mislabel certain samples** (i.e., classify them as belonging to the wrong class)?
- Particularly relevant case: given **a classifier trained to identify malicious JS**, it is possible to **modify malicious code** so that it is **considered benign**?
  - (AKA **adversarial sample generation**)

The paper

Assessing Adaptive Attacks Against Trained JavaScript Classifiers

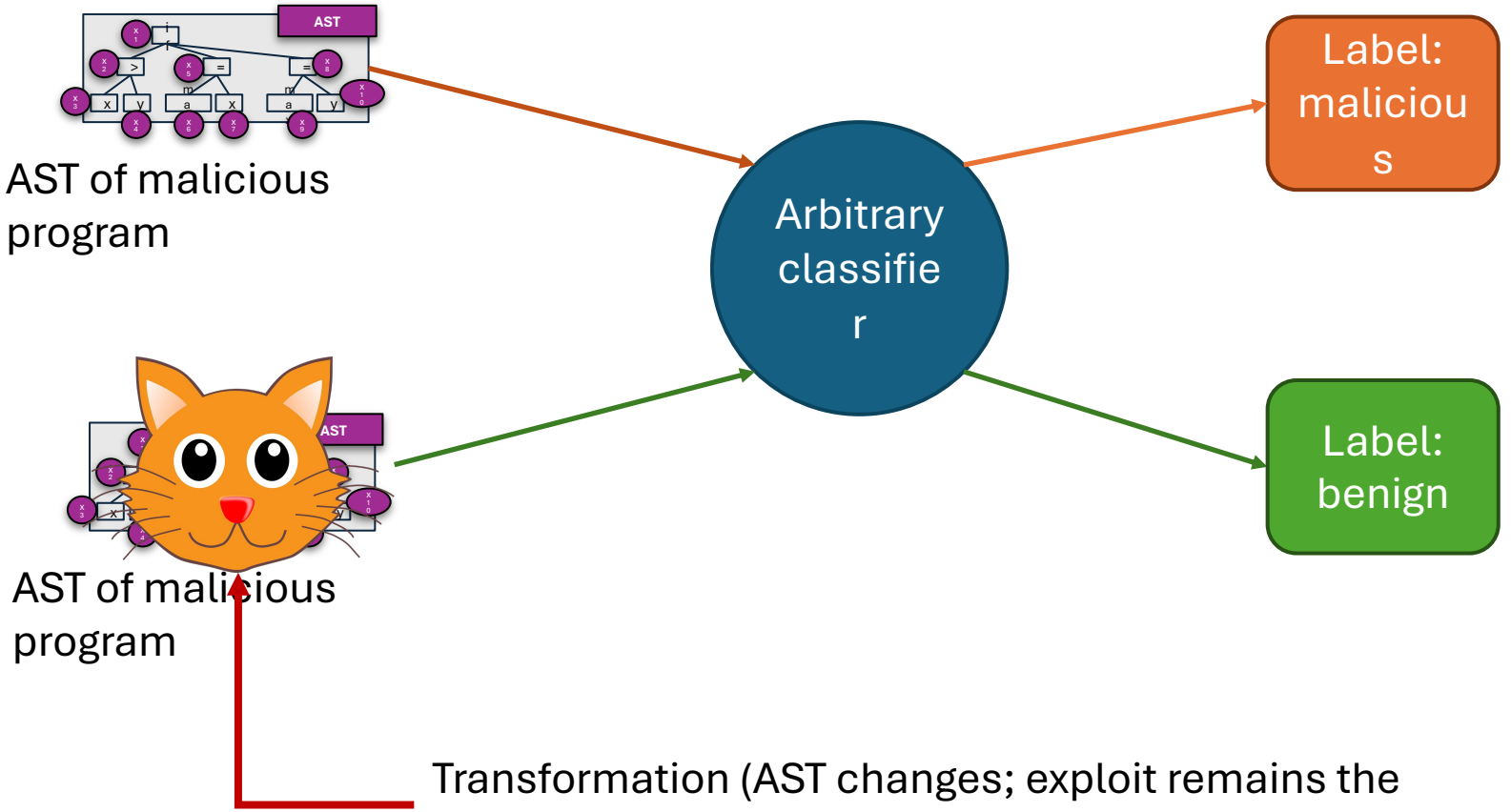
Niels Hansen[1], Lorenzo De Carli[2], and Drew Davidson[1]

[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.

Our research goal



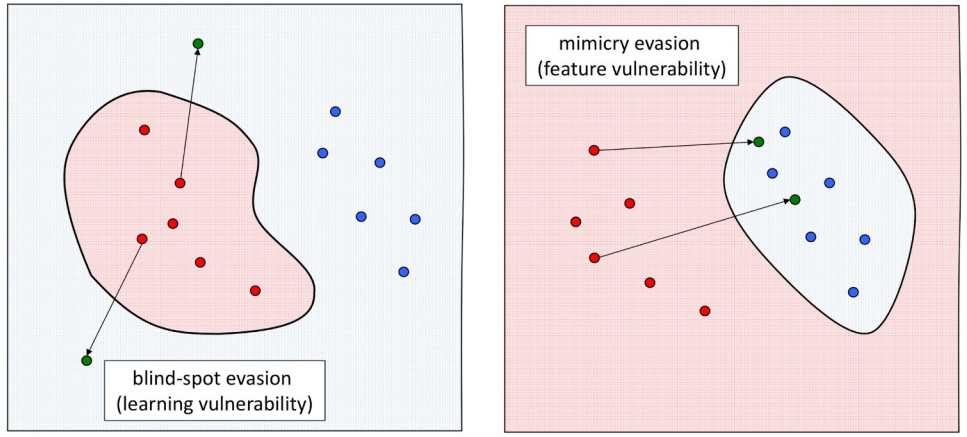
Why work on this?

- Understanding limitations of proposed defenses can lead to **better defenses**
- Other classifier-based approaches to identify malicious software have been broken:
  - PDF files (Xu et al., NDSS 2016; Srndic et al., S&P 2014)
  - Android malware (Yang et al., ACSAC 2017; Demontis et al., IEEE TDSC, 2018)
  - Flash malware (Demontis et al., 2017, on arXiv)
  - …
- This suggest that programmatically generate adversarial JS programs may be feasible

Some more context

- There are really two forms of adversarial samples: those which exploit **learning vulnerability**, and those which exploit **feature vulnerability**

Learning vulnerability: the boundaries around practical benign samples in feature space are not tight (i.e., there exist ample portions of feature space that are mapped to the "benign" class and can be used for adversarial samples)

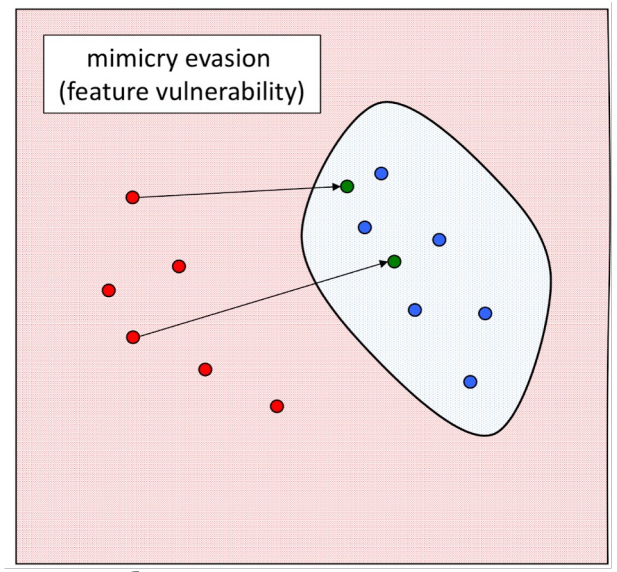


From Maiorca et al., arXiv:1710.10225 [cs], 2017

Feature vulnerability: the choice of feature does not represent meaningful aspects of the problem that allow to distinguish between classes. It is possible to devise malicious samples that look exactly like benign ones feature-wise

Some more context/2

From Maiorca et al., arXiv:1710.10225 [cs], 2017



- We hypothesize that AST-based detectors are susceptible to **feature vulnerability**, AKA **mimicry attacks**
- In other words, it is possible to transform a JavaScript program implementing an exploit so that **it generates the same features as a benign JavaScript program** (while retaining function)

Attack assumption & threat model

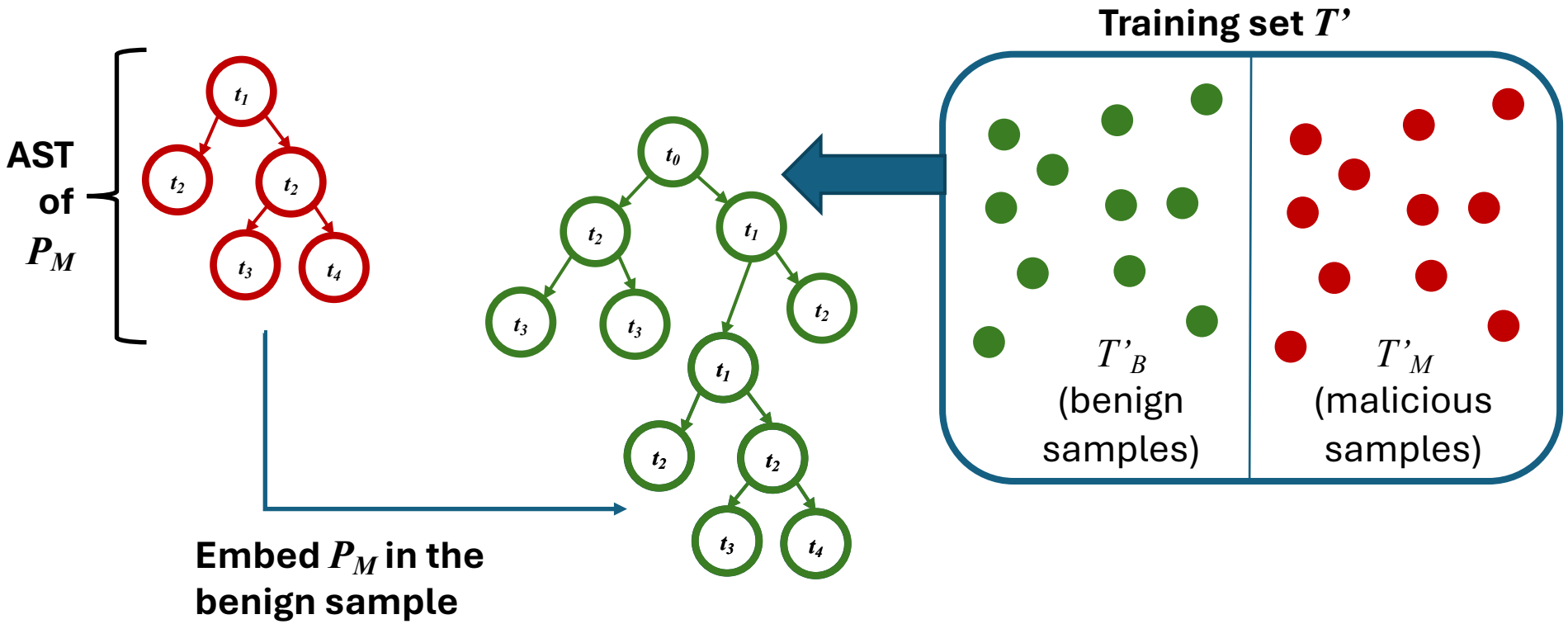
- The attacker's goal is to get a specific malicious program P_M (i.e., JavaScript exploit) misclassified as benign by a target classification function f
- The attacker has no access to f and its training set T, but can take a reasonable guess at a surrogate training set T' and train a surrogate classifier f' (**graybox attack**)

Attacking the model: formalization

- **Attack formalization:**
  - Consider the surrogate classifier $f': P \rightarrow \{B, M\}$ mapping a JS program P to one of two classes B, M
  - **Goal:** given a malicious program P_M s.t. $f'(P_M) = M$, find another program P_M^* s.t. $f'(P_M^*) = B$
  - Typically a constraint on the distance in feature-space $d(P_M, P_M^*) < d_{max}$ is added to ensure P_M^* retain its "maliciousness"
  - However, for our problem we replace distance constraint with a *functional equivalence* constraint
  - We say P_M, P_M^* are functionally equivalent if they both successfully accomplish the same exploit when run under the same conditions

Attack #1: subtree isomorphism

- **General idea:** search for a benign sample with an AST subtree which is isomorphic to the AST of P



Subtree isomorphism/2

- **Issue:** finding a subtree that is isomorphic to P_M may be difficult and likely unnecessary
  - It is generally enough to find subtree that is **similar** to P
  - Would also like to allow the **similarity threshold** to be **configurable**
- **Solution:** instead of finding a subtree S which is isomorphic to P_M, find S s.t. $TED(P_M, S) < \delta$, where TED is the **tree edit distance** between P and S and δ is an arbitrary threshold
- (Note that I am abusing the notation and use P to also denote the AST of P)

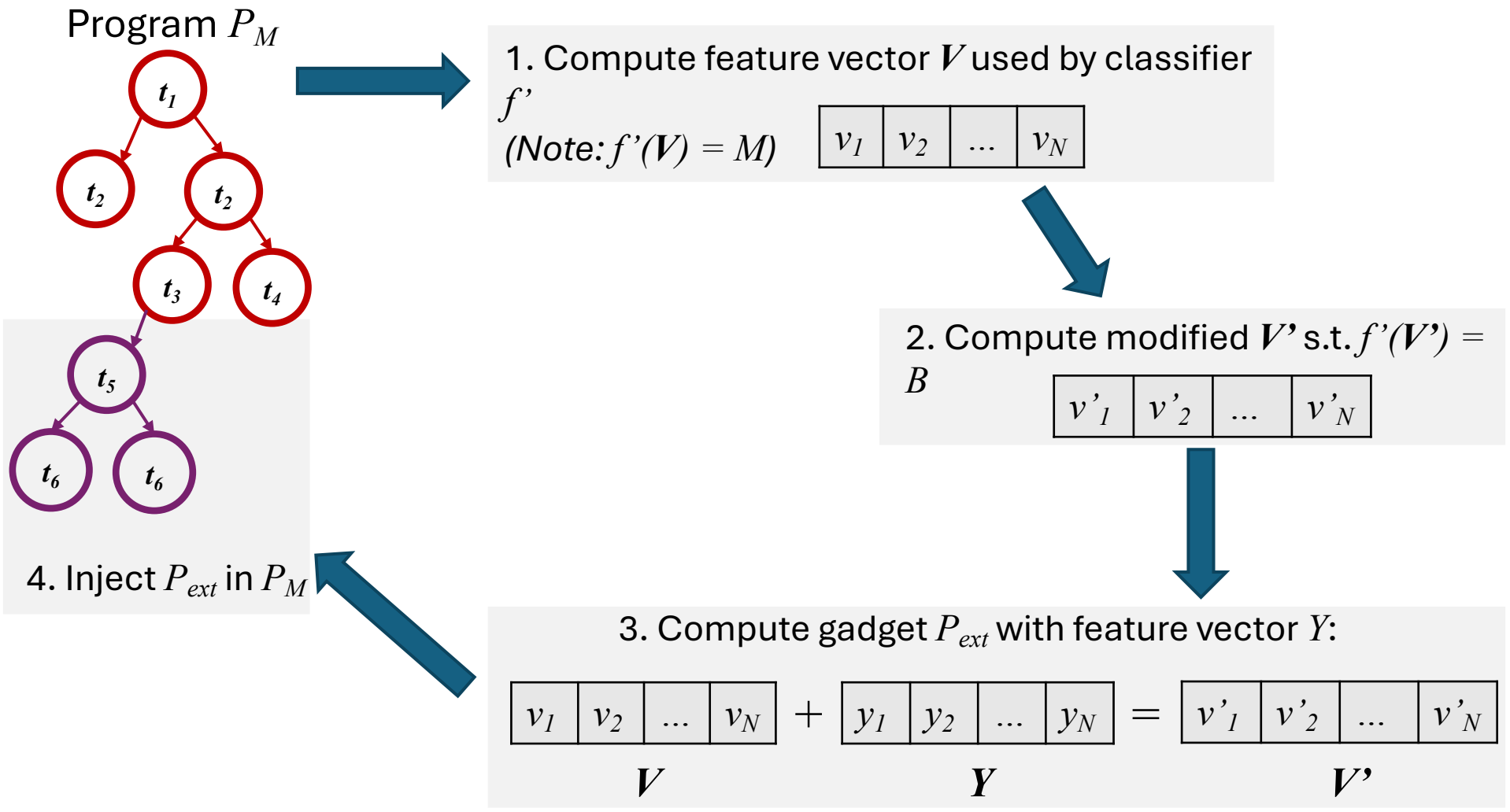
Subtree isomorphism/3

- For efficiency, when searching for a candidate subtree S we bound $|S| \leq M$ (with M close to $|P|$)
- TED has time complexity $O(M^3)$
- If L is the size of the largest AST in T'_B, then the comparison must be operated at most $O(L*M^2)$ per each program in T'_B
- Overall complexity: $O(|T'_B|*L*M^5)$
- **Take-away:** best if limited to small exploit programs

Attack #2: gadget injection

- Less powerful than subtree isomorphism, but easier to carry
- **Idea:** given a malicious program P_M, **synthesize** another program P_{ext} and inject it into P_M
- P_{ext} is designed to alter feature values of P_M so that it is classified as benign by a target classifier

Gadget injection/2



Gadget injection/3

- **Problem #1:** computing Y and V'
  - Can use statistical analysis of benign VS malicious feature vectors, or classifier-specific attacks (e.g., MILP formulation for random forest, gradient descent for DNN, etc.)
- **Problem #2:** given Y, generate P_{ext}
  - Mine dataset T' for gadgets (subprograms) that generate the desired feature values

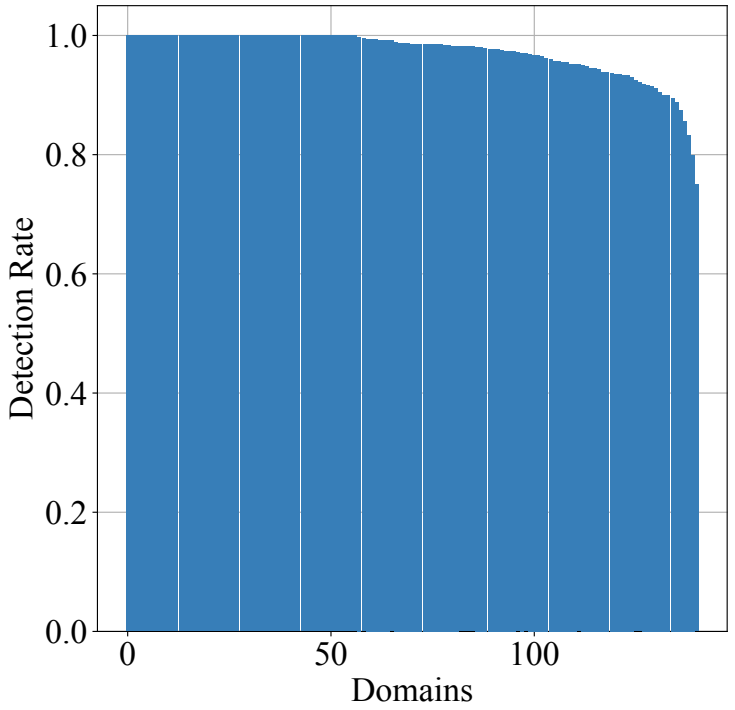
Results

- **Benign script dataset:** obtained from scraping Alexa top websites in Spring 2017; include 39091 scripts from 306 popular domains
- **Malicious script dataset:** online publicly-available JS exploit repositories from *geeksonsecurity.com*; include 1357 scripts performing various attack-related operations (collected in January 2017)

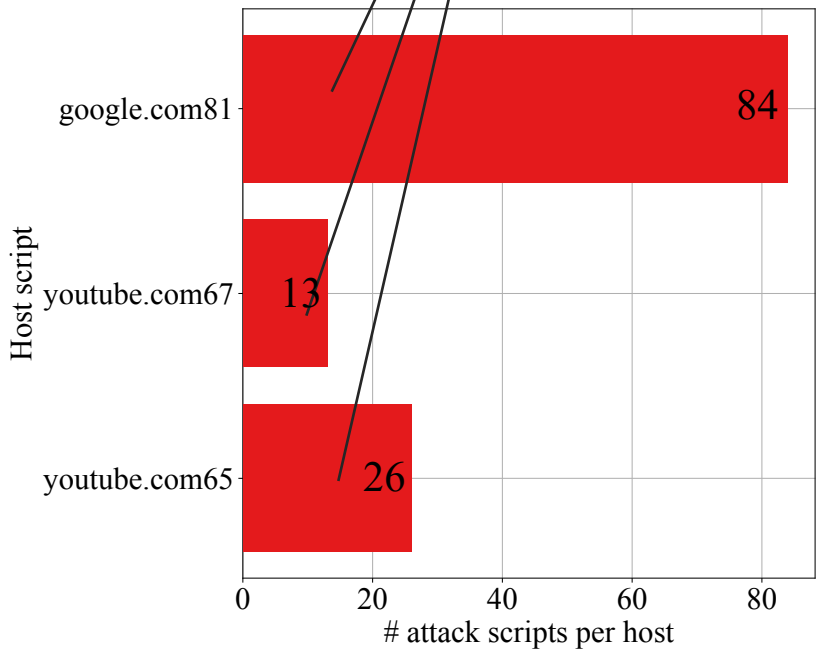
Results/2

A JaST classifier trained on the youtube/ malicious corpus (accuracy: 96.7%) detects none of these!

- Subtree isomorphism attack vs JaST



Performance of JaST trained on our dataset (w/o adversarial samples)



Number of generated attack scripts for three benign host scripts

See you next time!