Towards a Lightweight, Hybrid Approach for Detecting DOM XSS Vulnerabilities with Machine Learning

William Melicher
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
wrmelicher@gmail.com

Clement Fung
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
clementf@andrew.cmu.edu

Lujo Bauer
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
lbauer@cmu.edu

Limin Jia
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
liminjia@cmu.edu

ABSTRACT

Client-side cross-site scripting (DOM XSS) vulnerabilities in web applications are common, hard to identify, and difficult to prevent. Taint tracking is the most promising approach for detecting DOM XSS with high precision and recall, but is too computationally expensive for many practical uses.

We investigate whether machine learning (ML) classifiers can replace or augment taint tracking when detecting DOM XSS vulnerabilities. Through a large-scale web crawl, we collect over 18 billion JavaScript functions and use taint tracking to label over 180,000 functions as potentially vulnerable. With this data, we train a deep neural network (DNN) to analyze a JavaScript function and predict if it is vulnerable to DOM XSS. We experiment with a range of hyperparameters and present a low-latency, high-recall classifier that could serve as a pre-filter to taint tracking, reducing the cost of stand-alone taint tracking by 3.43× while detecting 94.5% of unique vulnerabilities. We argue that this combination of a DNN and taint tracking is efficient enough for a range of use cases for which taint tracking by itself is not, including in-browser run-time DOM XSS detection and analyzing large codebases.

CCS CONCEPTS

• **Security and privacy** → **Web application security**; • **Information systems** → **World Wide Web**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

web security, DOM XSS vulnerabilities, neural networks

ACM Reference Format:
William Melicher, Clement Fung, Lujo Bauer, and Limin Jia. 2021. Towards a Lightweight, Hybrid Approach for Detecting DOM XSS Vulnerabilities with Machine Learning. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3442381.3450062

WWW '21, April 19–23, 2021, Ljubljana, Slovenia

ACM ISBN 978-1-4503-8312-7/21/04.
https://doi.org/10.1145/3442381.3450062

1 INTRODUCTION

Web applications that fail to correctly sanitize their inputs can be vulnerable to cross-site scripting (XSS) vulnerabilities [16], which are becoming increasingly common [9, 15, 18, 32]. A specific type of XSS vulnerability, client-side XSS (DOM XSS), is caused by bugs in a website's JavaScript code; their prevalence is rising with the increase in complexity in client-side code [32]. In recent years, DOM XSS vulnerabilities have been reported at high-profile organizations such eBay, Yahoo, IBM, and Facebook [32].

DOM XSS vulnerabilities can be prevented by filtering out content using signatures and heuristics [11, 13, 29], but such defenses can be evaded by modern attack strategies [7, 8, 17, 33]. Other defenses detect DOM XSS vulnerabilities using static or dynamic analyses. In principle, static analysis can detect code-injection vulnerabilities before they are exploited or even released. However, static-analysis tools have difficulty reasoning about the dynamic features of JavaScript [14, 35, 41], have high error rates [26], or may not scale to large codebases [14], making them impractical as DOM XSS defenses.

In contrast, dynamic analyses—specifically, taint tracking—have shown promise for detecting DOM XSS vulnerabilities [22, 26, 38]. In dynamic taint tracking approaches, code is analyzed to detect DOM XSS vulnerabilities at execution time. This adds substantial overhead (16.8% increase in page load times) suggesting that such approaches are unlikely to be adopted in many settings, e.g., as in-browser defenses [12, 34].

Leveraging the observation that many DOM XSS vulnerabilities are syntactically similar and of low complexity [26, 39], we propose an alternative approach that uses machine learning (ML) to greatly reduce the overhead imposed by dynamic taint tracking to detect DOM XSS vulnerabilities. We also investigate the feasibility of composing ML with an existing analysis. Specifically, we address two primary research questions:

RQ1: Can ML act as a pre-filter for taint tracking to detect DOM XSS vulnerabilities with far less overhead than taint tracking alone *while* maintaining a high recall rate?

RQ2: Can ML be used on its own to detect DOM XSS vulnerabilities with recall and precision comparable to or better than other techniques?

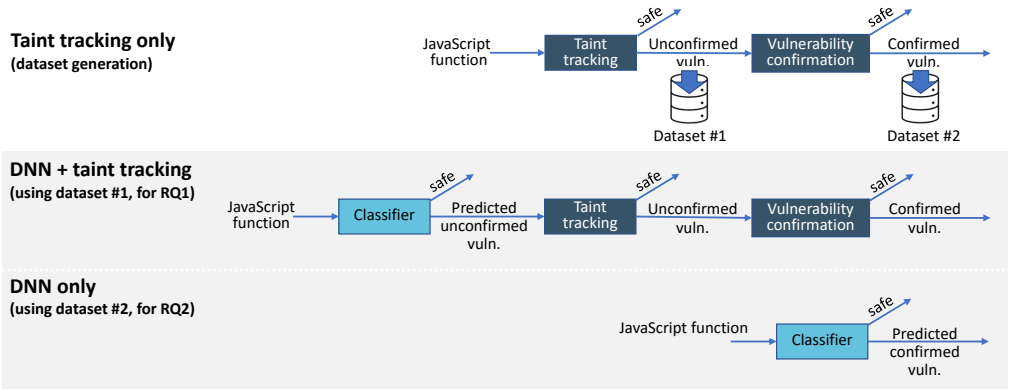


Figure 1: Illustrating how we collect training data, and how ML models trained on these datasets can be used in two different ways to reduce overhead in the dynamic taint tracking pipeline.

To train and evaluate these ML classifiers, we first obtain a sufficient volume of ground-truth data. We use an open-source taint-tracking-enabled web browser from prior work [26] to collect instances of DOM XSS vulnerable JavaScript functions in a two-step process (see Fig. 1). First, we use taint tracking to identify JavaScript functions that invoke dangerous sinks (e.g., `document.write`) with seemingly unsanitized arguments. Taint tracking itself cannot confirm that these functions are exploitable, so we label these functions as *unconfirmed vulnerabilities* (dataset #1). To prune this set of functions to only those that are exploitable, we use heuristics from prior work [26] to perform proof-of-concept exploits, and we label the exploitable functions as *confirmed vulnerabilities* (dataset #2).

Our classifier trained on *unconfirmed vulnerabilities* can be used as a pre-filter for taint-tracking-based DOM XSS detection (RQ1) to classify 97.5% of unique functions as non-vulnerable, while maintaining 94.5% unique recall of vulnerabilities. In this configuration, taint tracking is only used on the remaining 2.5% of functions, decreasing the vulnerability detection overhead by 3.43× when compared to taint tracking alone.

Alternatively, our classifier trained on *confirmed vulnerabilities* can be used as the sole means of judging whether a JavaScript function is vulnerable to DOM XSS attacks (RQ2), and captures 50% of confirmed vulnerabilities at a precision of 57.8%. In general, we have not found a tuning that delivers a combination of high recall and high precision sufficient for such a classifier to be the sole method of detecting DOM XSS vulnerabilities.

In exploring the classifier design space to answer these research questions, we experimented with two model types (linear models and deep neural networks (DNNs)); multiple representations of source code (based on scripts, functions, or semantic distance); varied model architectures (embedding sizes and DNN layer sizes); and adjusted training regimes to compensate for imbalanced ground truth (i.e., in the wild, non-vulnerable functions far outnumber vulnerable ones). The contributions of this paper are:

(1) We design and train classifiers to detect DOM XSS vulnerabilities and investigate trade-offs in inference time, precision, and recall (Sec. 5). We find that a relatively small DNN (4 fully connected layers of ≤ 100 units) can be an effective pre-filter for taint tracking, reducing the overhead of detection by 3.43× over taint tracking alone and enabling new use cases for taint tracking (Sec. 6.2).

(2) We manually examine a sample of functions in our dataset and uncover that the performance of our classifiers may be better than what we report (Sec. 6.3), because taint tracking fails to find unexecuted vulnerable functions.

(3) We manually inspect the performance and properties of our baseline linear models to provide an initial view into the characteristic differences between DOM XSS vulnerabilities that can be found by linear models and vulnerabilities that require DNNs (Sec. 6.3).

(4) We generate a dataset of 32 million JavaScript functions labeled as vulnerable or not via taint tracking and proof-of-concept exploit confirmation; we have made datasets and trained models publicly available[1].

2 BACKGROUND AND RELATED WORK

2.1 DOM XSS vulnerabilities

Cross-site scripting (XSS) vulnerabilities occur when input is improperly sanitized, allowing attackers to inject arbitrary JavaScript code into a victim's browser. An attacker could exfiltrate private information or compromise a victim's machine by redirecting to a malicious website.

In this work we focus on client-side vulnerabilities that result from client-side manipulation of the browser's Document Object Model (DOM). Attackers could inject exploits into sources such as the `document.location` object (the URL), the web page referrer, or the `postMessage` API. When information from these attacker controllable sources is used in sensitive code-executing functions (known as sinks), an XSS vulnerability may be present. Examples of such sinks include: the `innerHTML` property of DOM nodes, the `eval` method, or `javascript:` URLs.

An exploitable flow from a source to a sink does not necessarily imply an vulnerability, as programmers may sanitize information before use in the sensitive sink. Common sanitization methods include built-in browser APIs such as `encodeURI` and

[1]Data and trained models available at https://doi.org/10.1184/R1/13870256

`encodeURIComponent` or manually checking that user input matches a safe regular expression (e.g., alphanumeric characters).

In our work, we train a model for predicting DOM XSS vulnerabilities based on source code. We train separate models to predict *unconfirmed vulnerabilities* (that would be flagged by taint analysis and require further investigation) and *confirmed vulnerabilities* (that are confirmed by executing a proof-of-concept exploit).

2.2 Content filtering

Browsers and web servers have sought to employ filters for DOM XSS exploits. Content security policies (CSP) can restrict the allowed scripts on a website [13] but are often misconfigured in a way that does not substantially limit DOM XSS exploits [8]. An experimental browser API based on the concept of trusted types has been introduced [20], but this API is incompatible with legacy applications. Web application firewall filters are another common defense against XSS vulnerabilities, but can be bypassed by tweaking the exploit to evade the filter detection patterns [7, 17].

Client-side filters (such as the XSS auditor[11]) have also been used as a defense, but they too suffer from similar evasion attacks [33]. Researchers examined a list of known DOM XSS vulnerabilities, and showed that in 73% of cases the XSS auditor fails to filter an attack [38].

In contrast to using such heuristics, we train ML models to learn the filter policy and detect DOM XSS vulnerabilities. ML models may infer deeper relations between source code and vulnerabilities and could be more difficult for attackers to bypass.

2.3 Static analysis of JavaScript

Static analysis techniques can detect properties of interest by analyzing source code, allowing insight to all the possible execution paths that a program may take. Several implementations of static analysis for JavaScript are available in commercial tools including IBM Security AppScan, Trustwave App Scanner, Coverity's JavaScript scanner, and Burp Suite Pro [40]. However, it is particularly challenging to statically analyze JavaScript, as it is a dynamic language and lacks strict typing information [14, 45]. Furthermore, static analysis is prohibitively expensive for our setting, and thus we do not consider using static analysis in our solution.

2.4 Dynamic analysis of JavaScript

Dynamic analyses are also commonly used on Javascript [22], but only operates on observations collected during program execution and does not have analyze non-executed code. Furthermore, such methods incur run-time overhead [38] and require significant engineering work to modify a complex run-time environment (in our case, the JavaScript engine).

2.4.1 Taint tracking. The most relevant dynamic analysis for identifying DOM XSS vulnerabilities is taint tracking. This technique flags data from potentially attacker-controlled sources as tainted and propagates taint information at execution time. When a tainted string is used in a sensitive sink, the taint-tracking engine flags this flow of information as a potential DOM XSS vulnerability; we call this an *unconfirmed vulnerability*.

The first tool that used taint tracking to discover DOM XSS vulnerabilities was Firefox-based DOMinator [30]. Later, its precision was improved by adding byte-precise taint tracking, which attaches taint information to specific bytes in the JavaScript engine, reducing false positives [22, 38].

While taint tracking can effectively defend against DOM XSS vulnerabilities at run time, this is at the cost of overhead of, e.g., between 7% and 17% in certain benchmarks [38]. Browser vendors are exceptionally sensitive to performance overhead [12, 34] and our solution provides an opportunity to mitigate this performance degradation by using ML to selectively enable taint tracking when a classifier decides that the code may be vulnerable.

2.4.2 Confirming potentially vulnerable flows. Because tainted data may be sanitized by the programmer, such flows may not necessarily be exploitable. Researchers use heuristics to automatically generate exploits to confirm these vulnerabilities. In prior work, researchers generated exploits by analyzing the context around the tainted string [22], using a pre-configured list of exploit-causing injections [31] or symbolic analysis [43] to confirm flows with test injections. In our work, we leverage the above solutions to generate labeled instances of *confirmed vulnerabilities*.

2.5 Machine learning in program analysis

Several projects have used ML to analyze programs in JavaScript, successfully identifying many instances of malicious JavaScript [12, 44, 47]. However, these solutions rely on hand-engineered features (e.g. the location of the flow, the number of functions involved in the flow, the source and sink of the flow). We avoid using such techniques, allowing our solution to generalize to different contexts and to adapt to changing source code idioms. The building blocks for program analysis from data can also be learned by training decision trees [5]. In contrast our work opts for deep learning, which can learn latent representations of complex data.

The most closely related work has used deep learning to analyze the information flow in programs for more efficient taint tracking [37] or vulnerability detection [24] in C. In these projects, ML models must identify key points in the program to analyze the information flow, relying on the highly static nature of C programs. In our work, we focus on DOM XSS vulnerabilities in the browser, which predominantly executes dynamic JavaScript code. This makes it difficult to confidently determine the key points of a program and to extract data dependencies, and thus the above solutions do not apply to our setting.

2.5.1 Vector representations of programs. Representing programs in a form that deep neural network models can analyze is an open issue. Prior work has explored a handful of representations [3, 4, 24, 28] but there is little agreement about what representation is most appropriate for a given task.

Researchers have developed `code2vec`, which translates ASTs into vectors for machine learning by linking start and terminal nodes with a series of movements up and down the AST tree [4]. Tree convolutions analyze AST node information over the tree structure, in a similar way as a convolutional neural network processes images over its pixels. Tree convolutions base the classification of each node on the nodes that are close to it using neural network convolutions and have been used previously to detect algorithm performance bugs from source code [28].

Work has also been done on analyzing graph-structured data for use in program analysis. We also explored an approach using gated graph neural networks [23], which have recently been used to model certain properties of source code, such as idiomatic coding style [3]. However, we experimentally found that these techniques were unable to accurately model JavaScript semantics.

3 DATA COLLECTION METHODOLOGY

We describe our methodology in collecting two ground-truth datasets for training and evaluating our ML classifiers. We use a taint-tracking-enabled browser to collect unconfirmed vulnerabilities (dataset #1) in a large-scale web crawl, described in Sec. 3.1. We then label instances of these unconfirmed vulnerabilities as confirmed vulnerabilities (dataset #2), described in Sec. 3.2. Finally, we discuss attributes (Sec. 3.3) and limitations (Sec. 3.4) of our data collection.

3.1 Ground-truth data collection

We describe our infrastructure for collecting ground-truth data to train and test our vulnerability classifiers, shown in Fig. 1.

3.1.1 Taint tracking browser. We leverage a modified, taint-tracking-enabled version of the Chromium browser from prior work [26] to collect a series of website execution traces, identifying code that is potentially vulnerable to DOM XSS injections. The modified browser is driven by an extension that interacts with a server-side database, directing crawling activities via HTTP interactions and storing records of tainted flows.

The browser's V8 engine and WebKit infrastructure are modified with taint tracking to identify potentially vulnerable flows. During execution of each webpage's JavaScript, the modified browser stores: a record of all browser-executed source code, the parsed V8 representation of that source code, all tainted sink executions, and other bookkeeping information. For each execution of a sink with tainted data, we additionally log: the value of the tainted argument, the specific tainted characters, whether any specific built-in encoding methods were applied (e.g., `escape` or `encodeURI`), and a full trace of the JavaScript call stack.

Since the modified browser is only able to use taint tracking to identity potentially vulnerable code (which we define as *unconfirmed vulnerabilities*), we use proof-of-concept exploits from prior work [26] to further confirm whether these flows are indeed vulnerable to DOM XSS injection and can be labeled as *confirmed vulnerabilities*, a process described in Sec. 3.2.

3.1.2 Crawl methodology. In total, we crawled the Alexa top 10,000 [2] websites and visited 289,392 web pages on those websites. We began by visiting the root webpage of the website and sample 40 sublinks within the same domain for a total of 410,000 attempted webpage visits. Not all pages were loaded during our crawl; we obeyed robots.txt [19] directives and other webpages did not correctly load during our crawl. If an individual webpage did not load successfully, we attempted to load another sampled webpage on the same domain if possible. While loading webpages, the crawler first waits for the page ready event, then waits an additional 90 seconds for page execution. We empirically observed that 90 seconds was sufficient to detect the vast majority of tainted sinks.

Since our crawl is non-deterministic, we aggregate results across multiple executions. Log files from execution for our crawl are 26TB when compressed using GZIP. Many scripts are repeated across multiple crawls, so we remove all source code duplicates, reducing the compressed size of our aggregated database to 382GB.

3.2 Labeling and confirming flows

We next detect which scripts contain tainted arguments to potentially vulnerable sinks and where those sinks are located. If a flow is labeled as vulnerable in any execution, we label it as vulnerable in our aggregation. To locate the specific call to sink functions in source code, we output an annotated stack trace of the function call during execution, which contains the parsed AST of all executed JavaScript. We use the AST node of the JavaScript stack frame closest to the bottom of the call stack as indication of the vulnerability. We do not use other functions in the stack trace, since we do not have information about the location of the tainted flow's source and are unable to label such nodes. An overview of how ground-truth data is transformed and labeled is shown in Fig. 2.

3.2.1 Finding unconfirmed vulnerabilities. To determine whether a sensitive sink should be labeled as an unconfirmed vulnerability, we observe whether the encoding methods applied to tainted data match the context of where the taint is applied, using similar logic as prior work [22, 26, 38]. For example, if taint tracking indicates that the `document.write` function was called with a tainted argument from a webpage's URL without any applied encoding functions, we would mark that flow as an unconfirmed vulnerability. However, if the `encodeURI` function was later applied to the tainted bytes before use in the sink, then we would mark the function as safe, because the `encodeURI` function sanitizes the input and prevents the vulnerability. From the results of our data crawl, we collected approximately 32,000,000 instances of unconfirmed vulnerabilities, occurring in approximately 180,000 distinct, unique functions.

3.2.2 Confirming vulnerabilities. For the remaining unconfirmed vulnerabilities, we generate confirmation test injections by leveraging techniques from prior work [22, 26]. We combine our knowledge of the applied encoding functions to generate a proof-of-concept test injection for each unconfirmed vulnerability, and re-execute the webpage with our test injection to see if the injection succeeds. This step is necessary because developers have the broad ability to do ad-hoc sanitization of flows without using the built-in encoding methods, such as checking if a tainted input matches the regular expression for a number, a technique which would neutralize a unconfirmed vulnerability. After using the proof-of-concept exploits, we collected approximately 4,500,000 instances of confirmed vulnerabilities, occurring in over 2,300 distinct, unique functions.

Thus, as Fig. 1 shows, we create two datasets for training ML models: (1) a dataset of *unconfirmed vulnerabilities* based on the outputs of the taint tracking browser [26], and (2) a dataset of *confirmed vulnerabilities* based on our proof-of-concept test injections. The set of confirmed vulnerabilities is a subset of the unconfirmed vulnerabilities; we train two separate classifiers using these datasets and execute experiments on both.

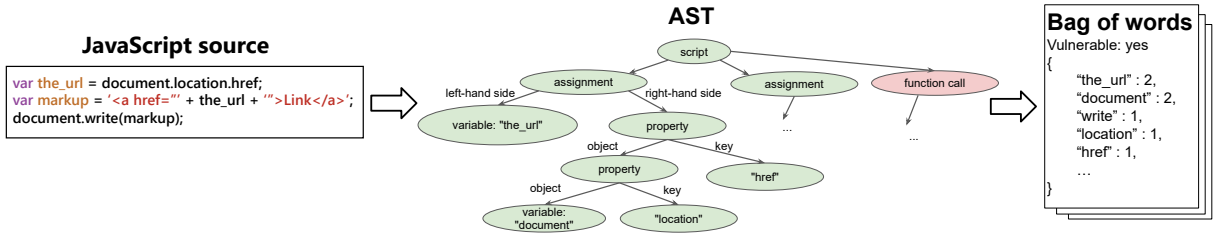


Figure 2: Ground-truth data transformation. The red node represents a vulnerable node. The function is then unitized, and the related context is converted into a bag of words.

Table 1: Summary statistics of our two vulnerability datasets. We present the average of all folds (see Sec. 4.2). We show the total number ("# Total") of functions, and measure the total number ("# vuln") and percentage ("% vuln") of functions that are vulnerable. "Distinct" removes duplicate functions by counting based on hashes of the function content. "Weighted" counts functions by their overall occurrence frequency.

			Training	Testing	Validation	All data
Confirmed	weighted	# total	15B	2.0B	1.8B	19B
		# vuln	3.8M	357K	354K	4.5M
		% vuln	0.025%	0.018%	0.019%	0.024%
	distinct	# total	383M	48M	48M	478M
		# vuln	1,853	235	238	2,326
		% vuln	0.00048%	0.00049%	0.00050%	0.00049%
Unconfirmed	weighted	# total	15B	1.7B	1.8B	19B
		# vuln	27M	2.8M	2.2M	32M
		% vuln	0.18%	0.17%	0.12%	0.17%
	distinct	# total	382M	48M	48M	478M
		# vuln	144K	19K	18K	180K
		% vuln	0.038%	0.039%	0.037%	0.038%

3.3 Properties of ground-truth dataset

After collecting ground-truth data, we wanted to understand the degree to which frequently used scripts (such as jQuery) could impact training and evaluation. If a small set of frequent scripts account for a significant amount of the dataset, then the ML model's performance could be dominated by its ability to recognize those frequent scripts. A summary of our datasets and the distribution of vulnerabilities is shown in Table 1. We found that while our datasets contain some frequently occurring scripts, there is also a significant long tail of unique scripts; our dataset included 240,830,867 observations of 23,013,705 unique scripts. The dataset is significantly one-sided: positive labels (vulnerabilities) are extremely rare compared to negative labels (non-vulnerable functions). Only 0.17% of all functions are unconfirmed vulnerabilities and 0.024% of all functions are confirmed vulnerabilities. Furthermore, these proportions are even smaller when considering unique scripts (0.038% and 0.0005%).

3.4 Limitations

Our browser infrastructure is based on an old version of Chromium, version 57 (a version from August 2016). In principle, the vulnerabilities that we observe may not apply to other browsers. This version of Chromium handles encoding of values after the hash differently than the latest version of Chromium, which may affect whether an unconfirmed vulnerability is exploitable. However, defense mechanisms in newer versions of Chromium are not ubiquitous in other browsers, so relying on newer versions of Chromium may overlook vulnerabilities that still affect many browsers.

Our ground-truth data is also constrained by the limitations of the dynamic analysis used for labeling. Our data only contains labeled AST nodes from actual executions, and we cannot make claims about code that is not executed. However, while our datasets contain false negatives, it does not contain false positives: all of our 2,326 confirmed vulnerabilities were demonstrated to be vulnerable on at least one generated proof-of-concept exploit.

If any instance of a function is labeled as vulnerable, then we label all instances of that function as vulnerable. However, exploiting that vulnerability may require cross-function interactions that are only present on some web pages. Arguably, it may still be appropriate to flag such functions as vulnerable, since they are not safe in all contexts. In either case, analyzing such cross-function interactions is complex and beyond the scope of this work.

4 CLASSIFIER DESIGN

We first describe our assumptions about potential threats to our ML model (Sec. 4.1). We then discuss our feature extraction and data processing techniques (Sec. 4.2). Finally, we describe our implementation details (Sec. 4.3) and evaluation metrics (Sec. 4.4).

4.1 Attacker capabilities: poisoning attacks and evasion attacks

The use of ML for security tasks can expose systems to new attacks. For example, in *poisoning attacks* attackers inject malicious training data into a system [6] and in *evasion attacks* attackers construct inputs that appear benign but evade detection [25, 36]. For the purposes of our design, we do not consider such attacks.

Although an adversary could possibly establish a malicious website that our web crawler then uses to collect poisoned training data, we assume that this is prohibitively expensive for an attacker (since we are crawling the 10,000 most popular websites) and the training data used to train our model is not poisoned by an adversary.

Regarding evasion attacks, our system is designed to detect *accidental* vulnerabilities (e.g., to help protect website developers who have control over the JavaScript on their website). If an attacker is able to manipulate the code on a website, the website is already compromised and an attacker would not need to evade our detection infrastructure. Our attacker model assumes that JavaScript code is benign but potentially vulnerable; attackers provide malicious input to websites to exploit DOM XSS vulnerabilities, but do not control the system otherwise.

4.2 Feature extraction and data preparation

Before being able to train on and classify pieces of source code, we must translate this code into a form that can be consumed by a neural network. In this section, we describe our methodology in translating labeled AST nodes into feature vectors for training.

4.2.1 Segmentation of code. Given a block of labeled source code, we chose to segment the code by its function calls. For the experiments presented in this work, the code located within an individual function call is used as a single unit for training and classification. We also attempted to segment code based on scripts, and using segments that contained the surrounding AST nodes within a fixed semantic distance; however, segmenting the code by functions produced the best results for training our model. Segmenting by entire scripts selects code snippets that are too large, while the fixed-semantic-distance strategy produces code snippets that are too small. Both representations prevent the classifier from learning meaningful features when predicting vulnerabilities.

4.2.2 Extracting features and code representation. After the labeled source code has been transformed into segments, we extract input features for our ML model. For the experiments shown here, we use a bag of words representation: each function is uniquely identified by a term-frequency dictionary of the parsed AST tokens contained within the function call. We store all of the relevant symbols and operations (variable names, operation names, method names, property names, etc) in this dictionary. Although variable and method names may change, we believe that this representation is robust in the face of small changes in source code, such as differing library versions, because often a significant amount of the variable names and function names are maintained across versions.

We also experimented with methodologies from prior work that extract program slices from C [24], but found that, the highly dynamic nature of Javascript prevents us from confidently identifying the key points of the program required for slicing. We also experimented with prior work that used models based on gated graph neural networks [3], but found that these techniques produced models that were unstable and performed poorly, potentially also due to the dynamsism of Javascript.

4.2.3 Experimental data setup. We divided our dataset into subsets: 80% "training" to train our models, 10% "validation" to evaluate competing models during hyperparameter exploration, and 10% "test" for measuring the final model performance. When dividing, we split by the script that the function originated from (for each script in our collected dataset, there is a 80% chance its function calls would be used for training, 10% chance in testing, and a 10% chance for validation). This split is performed to evaluate our model's performance on complete scripts that it has never seen before and captures a more realistic setting in which the model is presented with complete scripts (which are then segmented by functions).

Additionally, we would like to increase the importance of each function based on its observed frequency in our crawl. Functions that are defined in very common libraries are more important to classify correctly than code that is comparatively uncommon. To do this, we oversample frequent code instances in the training set before shuffling the data. This is preferred over applying a weight during training because, in our experiments, the models would not converge when presented with extremely common functions that massively outweighed other functions. If a common function is observed at the end of the training epoch, the model is drastically changed. However, by repeating instances multiple times, these effects are smoothed out over the training period.

4.2.4 Balancing errors. Another problem in training with our data is the massive class imbalance across labels: there are far more non-vulnerable functions than vulnerable functions (only 0.024% of all functions were confirmed as vulnerable). Therefore, we added a weight to positive labels during training by penalizing the loss function accordingly. We experimented with penalization terms of 1, 10, 100, and 1,000, and found that 100 was optimal—with lower penalizations the classifier would never predict functions as vulnerable, and with a penalization term of 1,000, the classifier would not converge.

4.2.5 Vectorizing features. We used feature hashing [46] to represent our sparse data, which allows our unbounded vocabularies to be represented as vectors by hashing terms to specific buckets. The downside of this technique is that it introduces ambiguity when the hash function has collisions. In order to mitigate the effect of collisions, we use a feature size of 2^{18}, a recommended size that balances memory requirements and collision probability [21]. We use an embedding layer that encodes the sparse bag of words into a dense vector space. This embedding is the first part of our model architecture, acts as the input to the first hidden layer, and is also optimized during training. We experiment with varying sizes of this embedding in Sec. 5.1.

4.3 Implementation

We build our model in TensorFlow [1] and train the model with the Adagrad optimizer (learning rate of 0.05, batch size of 64). For our smallest model (Fig. 5), the training time is 11K functions per second, which translates to approximately 20 hours to train on 5% of our total data, using a 64GB virtual machine with a 16GB NVIDIA Tesla P100 GPU.

4.4 Performance metrics

For any class imbalanced task, accuracy is not a useful metric, because a classifier could achieve near perfect accuracy by predicting that all functions are not vulnerable.

Since we are evaluating whether or not our ML model could be used in combination with other techniques, the precision-recall trade-off is more useful when tuning the trade-off between accuracy and overhead. We define *precision* as the proportion of predicted vulnerabilities that are indeed labeled vulnerabilities, and *recall*

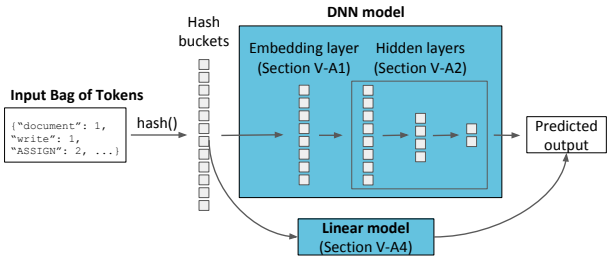


Figure 3: An overview of our ML architecture and its available hyperparameters. We explore with different embedding sizes, hidden layer sizes and model types.

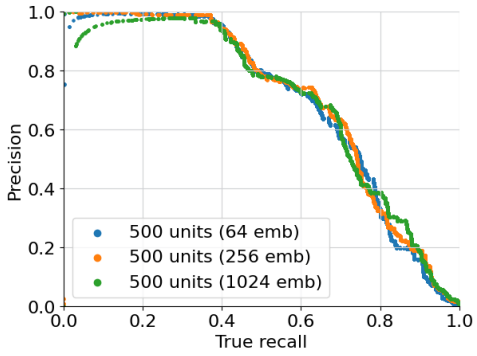
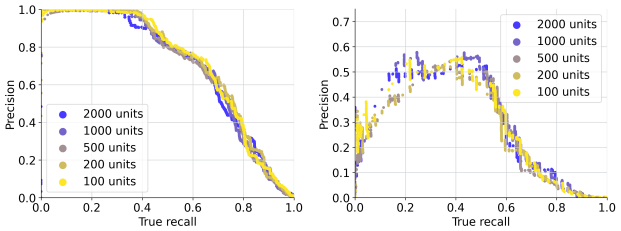


Figure 4: Varying embedding layer sizes in predicting unconfirmed vulnerabilities



(a) Unconfirmed **(b) Confirmed**

Figure 5: Effect of varying the model hidden-layer sizes when predicting unconfirmed (left) and confirmed (right) vulnerabilities. Points are plotted from largest (2000) to smallest (100), and overlap at several recall values.

as the proportion of labeled vulnerabilities that were correctly predicted as vulnerabilities.

Since recall can particularly be influenced by performing well on a frequent vulnerability, we also consider performance on *distinct vulnerabilities*. We define *distinct recall* to be the proportion of distinct labeled vulnerabilities correctly identified by our models, and *true recall* as the proportion of all labeled vulnerabilities that are correctly identified. When computing true recall, each function is weighted by its true observed frequency; so true recall represents recall on real data that the algorithm would encounter if deployed.

5 RESULTS

First, we use our validation dataset to tune parameters such as the model type and model size (Sec. 5.1). Then, we evaluate our best-performing models on our test data set, both for unconfirmed and confirmed vulnerabilities (Sec. 5.2).

In this section, the results shown are the average of 3 folds for unconfirmed vulnerabilities, and the average of 5 folds for confirmed vulnerabilities. Since the number of confirmed vulnerabilities is significantly lower then the number of unconfirmed vulnerabilities, we found that 3 folds were not sufficient for confirmed vulnerabilities, and thus used 5 folds for these experiments. We also found that using the entire training dataset was not required for convergence. We monitor the performance of our models during training, and ultimately decide that, for each fold, using 20% of the available training data (16% of the overall dataset) was sufficient.

5.1 Model size and type

We experimented with different sizes of deep neural network models. For these experiments, we report results for a 3-layer, fully-connected DNN. For each architecture, we conventionally halve the layer size after each layer, resulting in a fully connected architecture with layer sizes of [N, N/2, N/4], where N is the size of the first hidden layer. We also experimented with linear models and compared their performance to our DNNs. Fig. 3 highlights the different components of our ML architecture, and shows the various hyper-parameters that we evaluate.

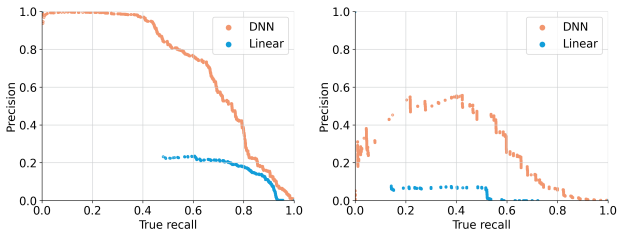
5.1.1 Embedding size. We first experimented with the size of the embedding layer in our neural network, described in Sec. 4.2.5. The embedding layer is a dense, fully-connected layer that translates the sparse tokens in hashed space (2^{18} in our implementation) and outputs a dense vector to the first DNN hidden layer. Fig. 4 shows various embedding sizes of 64, 256 and 1024 for a 3-layer DNN with N=500, trained to predict unconfirmed vulnerabilities. Again, we did not find a significant difference between embedding layer sizes, and chose the smallest embedding size of 64 for all future experiments to minimize size and inference time in our use case.

5.1.2 Model size. We explored the effect of model size by varying the size of the hidden layers in the [N, N/2, N/4] DNN architecture. For both unconfirmed and confirmed vulnerabilities, we trained DNNs where N = 100, 200, 500, 1000, and 2000. The results for unconfirmed vulnerabilities are shown in Fig. 5a and the results for confirmed vulnerabilities are shown in Fig. 5b. As expected, the performance in predicting unconfirmed vulnerabilities is significantly better than when predicting confirmed vulnerabilities. Across both experiments, we found that the model size also did not have a significant impact on the performance of the data. Since decreasing the model size does not adversely affect the prediction performance, we choose to use the smallest evaluated model architecture with (3 hidden layers of size 100, 50, and 25) in further experiments for both confirmed and unconfirmed vulnerabilities.

5.1.3 Model size trade-offs. In our proposed use case, smaller models are preferred due to their low inference time and small storage size. Without any optimization, the size of our chosen model

Table 2: Model sizes and inference times for various architectures. The results for our final selected configuration (100 units, 64 unit embedding) is bolded.

Embedding layer units	First hidden layer units	DNN size on disk	Inference Time on GPU / Desktop / Laptop
256	500	258 MB	12μs / 28μs / 45μs
1024	500	1027 MB	13μs / 50μs / 105μs
64	**100**	**65 MB**	**11μs / 17μs / 34μs**
64	200	65 MB	11μs / 19μs / 36μs
64	500	65 MB	11μs / 23μs / 39μs
64	1000	67 MB	11μs / 27μs / 48μs
64	2000	75 MB	12μs / 46μs / 100μs



(a) Unconfirmed **(b) Confirmed**

Figure 6: Performance of linear (Lin) and deep neural network (NN) models when predicting unconfirmed vulnerabilities (left) and confirmed vulnerabilities (right).

is 65MB. Since most of our models are small enough to be fully processed within our 12GB GPU, the inference time is largely unaffected by model size. To better understand the overhead of our model in other settings, we also measure the inference time on commodity non-GPU hardware and show the results in Table 2. For our chosen model with N = 100 and an embedding size of 64, the average time to classify a function is 11μs on our 24GB RAM, 4 core Intel i5-6400 3.30GHz CPU with a Titan X Pascal 12GB GPU, 17μs with a 32GB RAM, 12 core Intel E-2136 4.50GHz CPU desktop, and 34μs for a 8GB RAM, 8 core Intel i5-8250U 3.30GHz CPU laptop. We ultimately relate these numbers to potential end-to-end overhead savings when combining ML with taint tracking in Sec. 6.1.

The model size and inference time can be further reduced through different model encodings, compression techniques, and quantization. Prior work has shown that these techniques can enable deep neural network sizes to be reduced by two orders of magnitude [27]. Recently, TensorFlow has released libraries that enable DNNs to be compressed for inference on IoT and mobile devices [42], further improving the overhead savings and extending the potential reach of our solution to detect DOM XSS vulnerabilities in other domains.

5.1.4 Model types. We compared our trained DNNs to logistic regression models, which predict vulnerabilities based solely on a weighted linear combination of the observed code tokens (in our 2^{18} hash space). If linear models are able to accurately detect vulnerabilities, it would obviate the need for using a more complex DNN. The results in predicting unconfirmed vulnerabilities and confirmed vulnerabilities are shown in Fig. 6. In both cases, the neural network model far outperforms the linear model.

For example, when the prediction thresholds are set such that 50% of confirmed vulnerabilities are detected, the logistic regression model has a precision of 6.7%, while the DNN has a precision of 44.1%. For unconfirmed vulnerabilities, the precisions are 22.4% and 82.5% respectively. In both cases, the precision of the linear model drops to nearly 0% for recall rates over 90%, indicating a failure to capture higher complexity vulnerabilities; this trend is discussed in more detail in Sec. 6.3. We hence conclude that a linear model is not competitive with a DNN at detecting DOM XSS vulnerabilities.

5.2 Final models

Using the best-performing combination of our parameters, we train two final models, one to detect unconfirmed vulnerabilities (as labeled by taint tracking) and one trained on confirmed vulnerabilities (as labeled by testing with proof-of-concept exploits). The final models use a deep neural network with 3 layers—with 100, 50, and 25 units, respectively—trained on 20% of the available data.

5.2.1 Detecting vulnerabilities. The final results are shown in Fig. 7. For the model trained on unconfirmed vulnerabilities, when the threshold is set such that the true recall is 95%, the resulting precision is 26.7%. For confirmed vulnerabilities, a true recall of 95% of confirmed vulnerabilities exhibits an ineffective precision of 0.4%; the performance is poor likely because confirmed vulnerabilities are far less common than unconfirmed vulnerabilities.

Since we are more interested in the trade-off of the models' false-positive and false-negative rates than in high accuracy, we show the trade-off between the raw false-positive and true-positive rates as an ROC curve in Fig. 7. This is more meaningful when considering using such a model in practice, since a browser vendor would tune the model based on their tolerance towards false negatives, trading a higher recall for a lower precision.

As we show in Sec. 6.1, the performance of the model trained to predict unconfirmed vulnerabilities is sufficient such that it can be combined with taint tracking for a more efficient defense than taint tracking alone. Further, we show in Sec. 6.2 that some of the apparent false positives in our models are actually correct predictions (i.e., true positives), and are mislabeled in our dataset by the ground truth data collection methodology.

5.2.2 Previously unseen functions. Because our evaluation involves splitting our data into training, validation, and test datasets by unique scripts, functions may be duplicated across the training data and the test data. To understand the potential effect of duplication, we tested our models' performance on functions that did not appear in training data. For both our unconfirmed and confirmed vulnerability test datasets, we removed any function that was an exact match for a function that existed in any other script, forcing our models to only classify previously unseen functions. Our test data overall contains 48 million distinct functions. Once duplicated functions are removed, we test on the remaining 12 million previously unseen functions (average across all folds).

Fig. 7 shows the results when predicting on previously unseen functions. For confirmed vulnerabilities, the performance of the

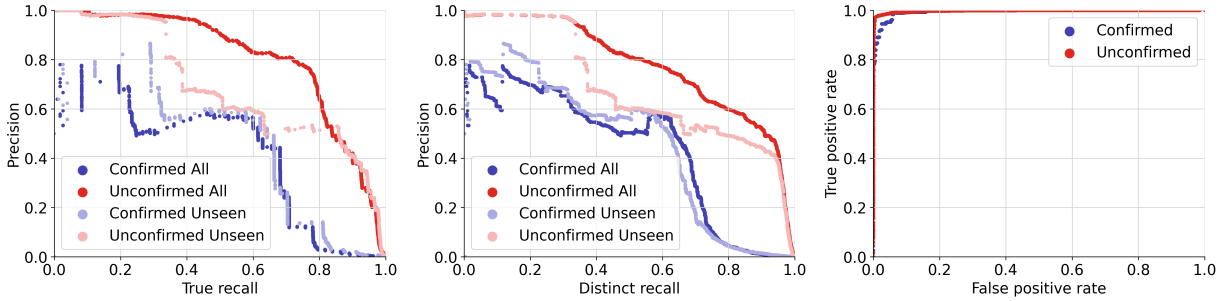


Figure 7: Model performance for test data.

model is only slightly different when predicting on previously unseen data, as shown by the similar performance across all true recall values beyond 40%. For example, when predicting confirmed vulnerabilities at 75% recall, our model has 12.1% precision across all vulnerabilities, and 13.7% precision when limited to previously unseen vulnerabilities. However, for unconfirmed vulnerabilities, there is a more pronounced effect, as shown in the large difference in performance between 40% and 80% recall when considering unseen functions. For example, at 75% recall the model has 77.6% precision across all vulnerabilities and 52.6% precision across previously unseen vulnerabilities, a difference of 25%.

6 DISCUSSION

We first revisit our research questions and discuss what our results imply when using our classifiers to detect DOM XSS in practice (Sec. 6.1). We then examine some seemingly incorrect predictions made by our classifier and show that many predictions were marked as incorrect due to noise in ground truth data, suggesting that our results may be better than reported (Sec. 6.2). Finally, we compare the behavior of our linear model with our DNNs and present potential characteristics of DOM XSS vulnerabilities that are easy or difficult to capture with simple models (Sec. 6.3).

6.1 Using ML classifiers to detect DOM XSS

Our research questions asked whether ML classifiers could help in effectively detecting DOM XSS vulnerabilities, either in combination with taint tracking (RQ1) or as a sole defense (RQ2).

6.1.1 RQ1: A classifier as a filter for taint tracking. To examine the potential utility of an ML classifer that selectively enables taint tracking when an unconfirmed vulnerability is predicted, we compute how many *real*, *confirmed*, vulnerabilities would be successfully detected by the combination of ML and taint tracking. We use the classifier trained on *unconfirmed* vulnerabilities (dataset #1) and measure the proportion of the resulting predictions that are later confirmed by the proof-of-concept exploit.

The recall in our method is *tunable*: if desired, the model can be tuned to capture a higher fraction of vulnerabilities, at the cost of additional taint tracking overhead. Fig. 8 shows the recall of confirmed vulnerabilities as we vary the proportion of functions passed to taint tracking for examination. We consider use cases where the classifier is used on all functions (weighted recall) and when the classifier is only used on distinct functions (distinct recall).

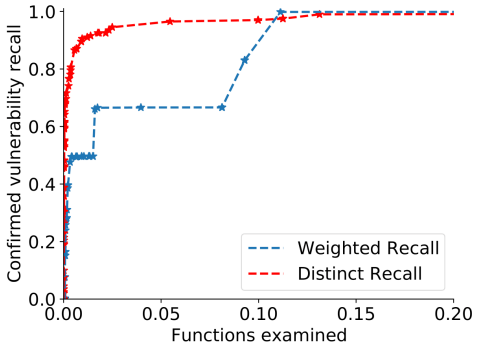


Figure 8: The trade-off between the recall of confirmed vulnerabilities and the fraction of all functions (weighted or distinct) examined by taint tracking.

When our classifier is tuned such that 11.1% of total functions are further examined by taint tracking, 99.8% of confirmed vulnerabilities are ultimately captured. When considering only distinct functions, the classifier can pass 2.5% of distinct functions to taint tracking and capture 94.5% of distinct confirmed vulnerabilities.

A challenge when selectively executing taint tracking in a run-time setting is that the source of the tainted flow needs to be identified. Prior work has explored the automatic detection of tainted sources based on sensitive sinks [10, 48] and we leave the combination of such techniques with our classifier as future work.

Based on these results, we envision two use cases in which our ML classifier could be deployed to reduce the overhead of taint tracking: run-time detection and analysis of large codebases.

Run-time detection of unconfirmed vulnerabilities. We consider the time saved when using a classifier in combination with taint tracking as a run-time defense in a web browser, compared to using just taint tracking.

To practically measure the performance in a run-time setting, we consider the performance based on observed *scripts*. Let n_{func} represent the number of functions in a script, o_{taint} represent the added overhead from taint tracking, t_{func} represent the average time taken to execute a single function, and t_{conf} represent the time taken to perform a proof-of-concept exploit on a single function. A fraction of the functions executed with taint tracking enabled,

p_{conf}, will be marked as containing unconfirmed vulnerabilities. Because many of these are false positives, we must determine whether each function is actually vulnerable through proof-of-concept exploits [26, 38]. In prior work [26], $p_{conf} = 0.0133$ and $t_{conf} = 2 * t_{func}$. When considering an individual script, the added time from taint tracking can be modeled as follows:

$$t_{added} = o_{taint} \cdot t_{func} \cdot n_{func} + (2 \cdot p_{conf} \cdot t_{func} \cdot n_{func}) \quad (1)$$

If ML is combined with taint tracking, each executed function incurs t_{ML} overhead for a classifier prediction, and only a fraction of functions, p_{taint}, is predicted to have unconfirmed vulnerabilities and is further analyzed as above with taint tracking and proof-of-concept exploits. If any script is predicted to contain at least one unconfirmed vulnerability, we assume that taint tracking will be enabled for the entire script. Thus, we estimate the fraction of scripts passed to taint tracking to be $1 - (1 - p_{taint})^{n_{func}}$, making the added execution time of the combination of ML with taint tracking:

$$t'_{added} = (t_{ML} \cdot n_{func}) + (1 - (1 - p_{taint})^{n_{func}}) * t_{added} \quad (2)$$

For the overhead from the combination of ML with taint tracking (t'_{added}) to be lower than the overhead from taint tracking alone (t_{added}), a large majority of scripts must not need to be analyzed with taint tracking, yet the recall of the classifier should be high enough to capture most vulnerabilities.

To estimate o_{taint} and t_{func}, we manually load the top 50 websites from the Alexa 10K [2] and observe that the average slowdown from the taint-tracking-enabled-browser (which looks at all executed scripts) is 16.8%. When aggregating across all scripts, the average time taken to execute a single function is 0.213ms. A proof-of-concept confirmation results in a function being executed at least one additional time, with additional overhead for customizing the exploit [26], so we estimate $t_{conf} = 0.416$ms, twice the original execution time. When considering the analysis in Fig. 8, $p_{taint} = 0.111$ for a recall of 99.8% of confirmed vulnerabilities. We also consider that classifier results could be cached, preventing the need to analyze duplicated functions. This would result in $p_{taint} = 0.025$ for a recall of 94.5% of distinct confirmed vulnerabilities.

In Sec. 5.1 we showed that t_{ML} varies by the hardware used. We calculate the difference between t_{added} and t'_{added} for each of these scenarios and report the reduction in overhead in Table 3. For a single function, this ranges from 1.07× on our laptop to 3.43× with caching on a desktop with a GPU machine.

As the number of functions in a script increases, the probability that at least one function in the script will require taint tracking also increases, decreasing the estimated savings in overhead; we show this trend in Fig. 9. In our datasets, scripts contained 161 functions on average, with a median of 2: a small number of scripts contain many functions, but most contain few. Our proposed solution performs better for the majority of scripts, which have few functions.

In practice, the overhead reduction is likely to be higher than what we report. First, we estimated the in-browser taint tracking overhead based on differences in load time, which includes fixed costs beyond JavaScript execution; the true overhead is likely higher. Second, the ML prediction is not dependent on JavaScript execution and could be run in parallel with other tasks.

Table 3: Per-function reduction in browser overhead when using our classifier as a pre-filter to taint tracking compared to taint tracking alone.

Device	Inference Time (t_{ML})	Savings	Cached Savings
Laptop (CPU)	34μs	1.07×	1.18×
Desktop (CPU)	17μs	1.91×	2.29×
Desktop (GPU)	11μs	2.66×	3.43×

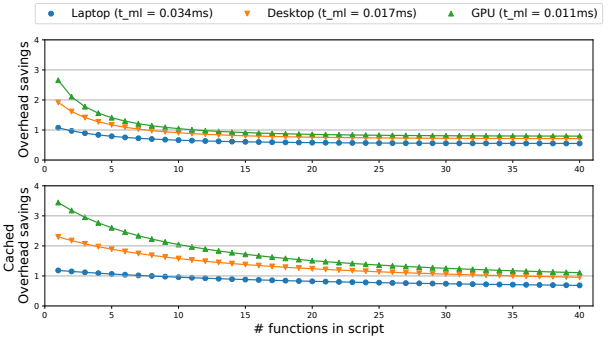


Figure 9: The changes in overhead savings without caching (top) and with caching (bottom) when considering the expected number of functions in a given script. As the number increases, a script is more likely to require taint tracking, so the overhead savings is reduced.

Analyzing large codebases. Another potential use of our classifier is to enable analysis of large codebases, such as software repositories, for which dynamic analysis would be prohibitively expensive. If, in a similar scenario as above, we analyzed all the functions in our dataset with taint tracking, the analysis would take over 7.8 days (0.0358ms per function × 19 billion functions). In contrast, if ML is used to discard predicted true negatives and taint tracking is applied only to the remaining functions, the whole process would require less than 1 day, while maintaining a 99.8% recall of vulnerabilities as described above.

6.1.2 RQ2: A classifier as the sole defense. If we were to use our classifier as the sole method to detecting DOM XSS vulnerabilities, the classifier would need a high precision and a high recall, since false positives would likely hinder practical use. Unfortunately, tuning our classifier trained on *confirmed* vulnerabilities for a high recall produces a high false positive rate, and tuning our model for high precision causes a large false negative rate.

For example, to capture 95% of *confirmed* vulnerabilities with our classifier, the corresponding precision is 0.4%, which results in far too many false positives for practical use. Conversely, the classifier can be tuned to achieve a precision of 75%, but this only captures 19.4% of confirmed vulnerabilities. For a compromise of both precision and recall, a "good" tuning of this model could exhibit 57.8% precision at 50% recall.

For all the tunings we considered, either the precision or the recall are insufficient for most practical uses. Hence, while we

Table 4: Tokens corresponding to top ten most influential features of the three linear models trained on unconfirmed vulnerabilities. Higher weights indicate tokens that are more influential towards predicting a vulnerability.

	Token	Average Weight
1	*"write"*	18.45
2	*"eval"*	10.43
3	*"<iframe src='{src}' width='0' height='0' style='display:none;'></iframe>"*	8.91
4	*"class="student-receiver" type="group" hex="*	7.70
5	*"innerHTML"*	6.93
6	*"/home-page"*	6.71
7	*"<!–[if gt IE"*	6.51
8	*"focusin"*	6.09
9	*"]><![endif]–> "*	5.75
10	*"text/html"*	5.52

answer RQ1 positively, we come to the opposite conclusion for RQ2: the classifier designs we investigated are *by themselves* not yet a *practical* method of detecting DOM XSS vulnerabilities.

6.2 Accuracy in the face of noisy ground truth

A challenge in detecting DOM XSS vulnerabilities was the absence of ground truth data that reliably labels vulnerable JavaScript functions. We nevertheless showed (RQ1) that an ML classifier could be trained to be an effective and efficient defense, when used in combination with dynamic taint tracking. Here we revisit the sources of inaccuracy from ground truth data and manually examine a subset of the classifier's false positives and false negatives.

One source of noise in our ground truth data comes from dynamic taint tracking; vulnerabilities that are on unexecuted paths during data collection are mislabeled as safe, since taint tracking would have had no opportunity to detect that they are vulnerable. Another source of noise from our data collection is that we use separate phases to detect unconfirmed vulnerabilities and to confirm them. For highly dynamic web sites, content may change between the two phases. In these false positive cases, a function that was initially labeled as a unconfirmed vulnerability may fail to be labeled as a confirmed vulnerability, simply because it was no longer available on the site during the confirmation phase.

We surprisingly observed that the precision of confirmed vulnerabilities is low even at low recall values (as seen in Fig. 5b and Fig. 7), where one would expect high precision at the cost of low coverage. This indicates that even some of the model's most confident positive predictions were incorrect. We manually investigated ten of the model's most confident false positives and found that seven of the ten errors were incorrectly labeled because of the data collection issues mentioned above; these seven "errors" were in fact true positives. This suggests that our classifier's performance may be much better than what we report.

6.3 Uncovering properties of vulnerabilities

Our results in Sec. 5.1 demonstrated the ineffectiveness of linear models in predicting DOM XSS vulnerabilities. However, we still observed that, at a 50% recall rate, over 1 million unconfirmed vulnerabilities were detected by the linear model at a reasonable (21%) precision. This suggests that some vulnerabilities have properties that make them easier to detect than others, and in this section we report on a manual analysis that explored such properties.

Poor performance aside, a benefit of linear models is that their model weights correspond directly to the influence that particular features have on the prediction output. Thus, for the three linear models trained on unconfirmed vulnerabilities, we analyzed the *most influential* features for predicting a vulnerability. Each feature corresponds to one of the 2^{18} hash buckets over the bag-of-words representation, so we further analyze our test data to find the most frequent tokens that map to these hash buckets, shown in Table 4. Eight of the top ten tokens are shared among all three models.

We make two observations based on these findings. First, *"write"* and *"eval"* are the most significant two tokens for all models by a large margin. We re-compared the outputs of our linear model and our neural network (from Fig. 6a) at their 50% recall rates, and consider only the vulnerabilities that contain *"write"* or *"eval"*. Although the total number of true positives identified by both models is approximately the same, the linear model has a much larger false positive rate; the linear model's precision is 22.4%, compared to the DNN precision of 82.5%. At these operating points, 67% of the linear model's unique true positives contain *"write"*, while only 34.7% of the neural networks' do. We repeated this exercise for "eval"; the difference between the two models was minimal.

Linear models may be biased toward identifying any functions that invoke *document.write* as vulnerable. This is common in practice, as we observed that 60% of our unique unconfirmed vulnerabilities contain *"write"*. Linear models are thus much more prone to identify functions as vulnerable even when their uses of *document.write* are safe, leading to high false positive rates. In contrast, the DNN models appear to learn a more nuanced relationship for cases with *"write"*.

Second, we noticed that for linear models many of the most influential tokens contain long HTML strings that appear in the JavaScript code as string constants (tokens 3, 4, 7, and 9 in Table 4). We searched for these tokens across all public JavaScript repositories on GitHub and found that these tokens occur in frequently copied and imported JavaScript libraries. In all cases, a JavaScript variable is appended to these HTML strings and the result is directly written to the document, exposing a clear DOM XSS vulnerability. Linear models are also well suited to identify and capture these cases with ease, even though they do not directly encode any problematic code semantics that would generally be indicative of vulnerabilities.

DNN models were also able to capture these pathological cases, but additionally achieved much higher precision than linear models. The precision of linear models was particularly poor outside of a small subset of vulnerabilities. This suggests that a large majority of DOM XSS vulnerabilities are still complex enough that a DNN is required to precisely model their characteristics.

7 CONCLUSION

We examined two approaches for ML classifiers to detect DOM XSS vulnerabilities in source code: (1) using ML as a filter for scripts before using taint tracking (RQ1); and (2) using just an ML classifier to detect DOM XSS vulnerabilities directly (RQ2). We collected and labeled 18 billion JavaScript functions in a large-scale web crawl and trained ML models on representations of their source code.

We found that classifiers could be trained to detect DOM XSS vulnerabilities with sufficient recall and precision that using them as a pre-filter for a taint-tracking-based defense substantially reduces the overhead of DOM XSS detection. For example, the overhead of DOM XSS detection in a web browser context could be reduced by 3.43× compared to using taint tracking alone. We argue that this enables new uses for taint-tracking-based DOM XSS detection in contexts with strict performance requirements.

ACKNOWLEDGMENTS

We thank Michael Stroucken and Yoshiki Takashima for help with the web crawl and experiments. This work was supported in part by gifts from John & Claire Bertucci, by CyLab at Carnegie Mellon University via a CyLab Presidential Fellowship, and by the National Science Foundation via grant CNS1704542.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. USENIX Conference on Operating Systems Design and Implementation.*

[2] Alexa. 2017. Top sites in United States. alexa.com/topsites/countries/US.

[3] M. Allamanis, M. Brockschmidt, and M. Khademi. 2018. Learning to Represent Programs with Graphs. In *Proc. Int'l. Conference on Learning Representations.*

[4] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. 2019. code2vec: Learning distributed representations of code. *Proc. ACM on Programming Languages* (2019).

[5] P. Bielik, V. Raychev, and M. Vechev. 2017. Learning a static analyzer from data. In *Proc. International Conference on Computer Aided Verification.*

[6] B. Biggio, B. Nelson, and P. Laskov. 2012. Poisoning attacks against support vector machines. In *Proc. International Conference on Machine Learning.*

[7] K. Bijjou. 2015. Web application firewall bypassing—how to defeat the blue team. OWASP open web application security project.

[8] S. Calzavara, A. Rabitti, and M. Bugliesi. 2016. Content security problems?: Evaluating the effectiveness of content security policy in the wild. In *Proc. ACM SIGSAC Conference on Computer and Communications Security.*

[9] Cenzic, Inc. 2014. Application vulnerability trends report. https://www.info-point-security.com/sites/default/files/cenzic-vulnerability-report-2014.pdf.

[10] V. Chibotaru, B. Bichsel, V. Raychev, and M. Vechev. 2019. Scalable taint specification inference with big code. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[11] Chromium. 2010. The Chromium projects: XSS auditor. https://www.chromium.org/developers/design-documents/xss-auditor.

[12] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. 2011. ZOZZLE: Fast and precise in-browser JavaScript malware detection. In *Proc. USENIX Security Symposium.*

[13] Foundeo, Inc. 2018. Content Security Policy reference. https://content-security-policy.com/.

[14] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. 2011. Saving the World Wide Web from vulnerable JavaScript. In *Proc. International Symposium on Software Testing and Analysis.*

[15] Hackerone. 2017. The Hacker-powered security report 2017. https://www.hackerone.com/sites/default/files/2017-06/The%20Hacker-Powered%20Security%20Report.pdf.

[16] T. Hunt. 2013. Understanding XSS – input sanitisation semantics and output encoding contexts. www.troyhunt.com/understanding-xss-input-sanitisation.

[17] V. Ivanov. 2016. Web application firewalls: Attacking detection logic mechanisms. Blackhat USA.

[18] A. Janc, M. Spagnuolo, L. Weichselbaum, and D. Ross. 2016. Reshaping web defenses with strict Content Security Policy. https://security.googleblog.com/2016/09/reshaping-web-defenses-with-strict.html.

[19] M. Koster. 2017. The Web Robots pages. https://www.robotstxt.org/

[20] K. Kotowicz. 2019. Trusted types help prevent cross-site scripting. https://developers.google.com/web/updates/2019/02/trusted-types.

[21] Scikit Learn. 2019. Feature Extraction. https://scikit-learn.org/stable/modules/feature_extraction.html.

[22] S. Lekies, B. Stock, and M. Johns. 2013. 25 million flows later: Large-scale detection of DOM-based XSS. In *Proc. ACM SIGSAC Conference on Computer and Communications Security.*

[23] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow. 2016. Gated graph sequence neural networks. In *Proc. International Conference on Learning Representations.*

[24] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proc. Network and Distributed System Security Symposium.*

[25] B. Liang, M. Su, W. You, W. Shi, and G. Yang. 2016. Cracking classifiers for evasion: A case study on the Google's phishing pages filter. In *Proc. International World Wide Web Conference.*

[26] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia. 2018. Riding out DOMsday: Toward detecting and preventing DOM cross-site scripting. In *Proc. Network and Distributed System Security Symposium.*

[27] W. Melicher, B. Ur, S.M. Segreti, S. Komanduri, L. Bauer, N. Christin, and L.F. Cranor. 2016. Fast, lean, and accurate: Modeling password guessability using neural networks. In *Proc. USENIX Security Symposium.*

[28] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proc. AAAI Conference on Artificial Intelligence.*

[29] Open Web Application Security Project. 2016. Web application firewall. https://www.owasp.org/index.php/Web_Application_Firewall.

[30] S. Di Paola. 2011. DOMinator. https://github.com/wisec/DOMinator.

[31] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena. 2015. DexterJS: Robust testing platform for DOM-based XSS vulnerabilities. In *Proc. Joint Meeting on Foundations of Software Engineering.*

[32] G. Podjarny. 2017. Snyk blog: XSS attacks: The next wave. https://snyk.io/blog/xss-attacks-the-next-wave/.

[33] pythech's Blog. 2017. Yet another Chrome XSS auditor bypass. https://turkmeno g.lu/blog/2017/11/06/yet-another-chrome-xss-auditor-bypass/.

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

[35] G. Richards, S. Lebresne, B. Burg, and J. Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *ACM Sigplan Notices*, Vol. 45. 1–12.

[36] N. Rndic and P. Laskov. 2014. Practical evasion of a learning-based classifier: A case study. In *Proc. IEEE Symposium on Security and Privacy.*

[37] D. She, Y. Chen, A. Shah, B. Ray, and S. Jana. 2020. Neutaint: Efficient dynamic taint analysis with neural networks. In *Proc. IEEE Symposium on Security and Privacy.*

[38] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns. 2014. Precise client-side protection against DOM-based cross-site scripting. In *Proc. USENIX Security Symposium.*

[39] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns. 2015. From facepalm to brain bender: exploring client-side cross-site scripting. In *Proc. ACM SIGSAC Conference on Computer and Communications Security.*

[40] L. Suto. 2013. Analyzing the accuracy and time costs of web application security scanners. https://www.beyondtrust.com/assets/documents/bt/Analyzing-the-Accuracy-and-Time-Costs-of-Web-Application-Security-Scanners.pdf.

[41] A. Taly, Ú. Erlingsson, J.C. Mitchell, M.S. Miller, and J. Nagra. 2011. Automated Analysis of Security-critical JavaScript APIs. In *Proc. IEEE Symposium on Security and Privacy.*

[42] Tensorflow. 2020. Tensorflow Lite—ML for mobile and edge devices. https://www.tensorflow.org/lite.

[43] O. Tripp, P. Ferrara, and M. Pistoia. 2014. Hybrid security analysis of web JavaScript code via dynamic partial evaluation. In *Proc. International Symposium on Software Testing and Analysis.*

[44] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin. 2014. ALETHEIA: Improving the usability of static security analysis. In *Proc. ACM SIGSAC Conference on Computer and Communications Security.*

[45] O. Tripp, M. Pistoia, S.J. Fink, M. Sridharan, and O. Weisman. 2009. TAJ: Effective taint analysis of web applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[46] K. Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. (2009).

[47] F. Yamaguchi, F. Lindner, and K. Rieck. 2011. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proc. USENIX Workshop on Offensive Technologies.*

[48] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *Proc. IEEE Symposium on Security and Privacy.*