

Challenging Machine Learning Algorithms in Predicting Vulnerable JavaScript Functions

Rudolf Ferenc[†], Péter Hegedűs*, Péter Gyimesi[†], Gábor Antal[†], Dénes Bán[†], and Tibor Gyimóthy*[†]

*MTA-SZTE Research Group on Artificial Intelligence, Szeged, Hungary

{hpeter | gyimothy}@inf.u-szeged.hu

[†]Department of Software Engineering, University of Szeged, Szeged, Hungary

{ferenc | pgyimesi | antal | zealot}@inf.u-szeged.hu

Abstract—The rapid rise of cyber-crime activities and the growing number of devices threatened by them place software security issues in the spotlight. As around 90% of all attacks exploit known types of security issues, finding vulnerable components and applying existing mitigation techniques is a viable practical approach for fighting against cyber-crime. In this paper, we investigate how the state-of-the-art machine learning techniques, including a popular deep learning algorithm, perform in predicting functions with possible security vulnerabilities in JavaScript programs.

We applied 8 machine learning algorithms to build prediction models using a new dataset constructed for this research from the vulnerability information in public databases of the Node Security Project and the Snyk platform, and code fixing patches from GitHub. We used static source code metrics as predictors and an extensive grid-search algorithm to find the best performing models. We also examined the effect of various re-sampling strategies to handle the imbalanced nature of the dataset.

The best performing algorithm was KNN, which created a model for the prediction of vulnerable functions with an F-measure of 0.76 (0.91 precision and 0.66 recall). Moreover, deep learning, tree and forest based classifiers, and SVM were competitive with F-measures over 0.70. Although the F-measures did not vary significantly with the re-sampling strategies, the distribution of precision and recall did change. No re-sampling seemed to produce models preferring high precision, while re-sampling strategies balanced the IR measures.

Keywords-vulnerability, JavaScript, machine learning, deep learning, code metrics, dataset

I. INTRODUCTION

JavaScript is getting traction not just in client-side web development but as a desktop and server language (Node.js), mobile app language (React Native), or even as an IoT (e.g. JerryScript or the Espruino framework) implementation language. Therefore, programs written in JavaScript are exposed more and more to various security risks.

Even though the rapid rise of cyber-crime activities and the growing number of devices threatened by them place software security issues in the spotlight, security concerns of programs are still neglected from time to time. According to past studies [1], around 90% of all attacks exploit known types of security issues. Therefore, finding vulnerable components for applying existing mitigation techniques on them might be a viable practical approach for fighting against cyber-crime. In this paper, we investigate how the state-of-the-art machine learning techniques – including a popular

deep learning algorithm – perform in predicting functions with possible security vulnerabilities in JavaScript programs.

Security vulnerabilities are very similar to bugs (i.e. most of them can be seen as special types of bugs, though not functional), however, many studies show that bug prediction models cannot be applied for finding vulnerabilities as is [2], [3]. Although this suggests that specific prediction models are needed for finding vulnerable software components, we can still leverage the abundance of knowledge already accumulated in the area of bug prediction. JavaScript, however, is not well studied in terms of bug prediction, so general conclusions based on other languages might not hold.

Moreover, most of the bug prediction models find fault-prone files or classes [3]–[9], while rarely working at a finer granularity level (e.g. for methods, functions, or statements [10]). These approaches are less effective for JavaScript, as source code is often structured in only several files (even into one single `js` file) and usually there are no higher level logical constructs (like classes) above functions. Prediction models for vulnerable source files would not be really useful in such contexts; we need at least function level vulnerability information and prediction models.

To the best of our knowledge, there are no existing vulnerability datasets specifically for JavaScript programs, which would contain vulnerability information at the level of functions. VulinOSS [11] and VulData7 [4] are very useful proposals with the aim of collecting general vulnerability datasets together with fixing patches. However, they are not specific to JavaScript and do not map the fixed vulnerabilities to individual functions.

For this study, we created a fine-grained, public, JavaScript vulnerability dataset with data extracted from *nsp* (Node Security Platform [12]) and the *Snyk Vulnerability Database* [13] automatically matched with information available on GitHub (i.e. fixing commits and patches). The new function level vulnerability dataset contains 12,125 functions from which 1,496 are vulnerable. It includes static code metrics provided by the OpenStaticAnalyzer [14] and escomplex [15] tools, too.

With the help of this dataset, we investigate if predicting vulnerable functions is feasible based on the fast and easily calculable static code metrics. We compare the performances of the most widely used machine learning algorithms on this prediction task, including two deep neural network variants,

the K-Nearest Neighbors algorithm (KNN), a decision tree classifier (Tree), the C-Support Vector Classification variant of the Support Vector Machine algorithm (SVM), Random Forest (Forest), Logistic regression (Logistic), Linear regression (Linear) and the Gaussian Naive Bayes algorithm (Bayes). We apply various re-sampling strategies to handle the imbalanced nature of the dataset.

In this paper, we address the following research questions:

RQ1: Is predicting vulnerable JavaScript functions feasible using static source code metrics?

RQ2: How do the various machine learning algorithms perform compared to each other for vulnerability prediction?

Given the highly dynamic nature of JavaScript, we got encouraging results using only static code metrics as predictors. The main contributions of the paper are two-fold:

- We release a new public vulnerability dataset consisting of the static analysis results of 12,125 JavaScript functions complemented with the information whether the functions contain a vulnerability or not.
- We publish a comprehensive comparison of 8 well-known machine learning algorithms on predicting vulnerable functions.

II. RELATED WORK

In their preliminary study, Siavvas et al. [16] investigated if a relationship exists among software metrics and specific vulnerability types. They used 13 metrics and found that software metrics may not be sufficient indicators of specific vulnerability types, but using novel metrics could help. In our study, we used 35 static source code metrics, including various Halstead variants, and found that they can effectively predict vulnerable functions in JavaScript.

In their work, Jimenez et al. [4] proposed an extensible framework (VulData7) and dataset of real vulnerabilities, automatically collected from software archives. Although it is similar to our work, VulData7 is general, i.e., it contains vulnerabilities for various languages at file level. Even though it contains JavaScript vulnerabilities, using them in our study was infeasible, as JavaScript files could contain lots of functions. Our proposed database is more fine-grained, every piece of information is available at the function level, thus enabling more accurate experiments.

Neuhas et al. [5] introduced a new approach (and the corresponding tool) called Vulture, which can predict vulnerable components in the source code, mainly relying on the dependencies between the files. They analyzed the Mozilla code base to evaluate their approach using SVM for classification. Although their results are very promising, we could not locate the proposed tool online. Contrary to this work, we predict vulnerabilities at the level of JavaScript functions and apply multiple machine learning approaches.

In their work, Shin et al. [10] created an empirical model to predict vulnerabilities from source code complexity metrics. Their model was built on the function level similar

to ours, but they consider only the complexity metrics. They concluded that vulnerable functions have distinctive characteristics separating them from “non-vulnerable but faulty” functions. They studied the JavaScript Engine from the Mozilla application framework. In this paper, we use 35 different metrics as predictors and build our prediction models specifically for JavaScript programs.

Shin et al. [6] performed an empirical case study on two large code bases: Mozilla Firefox and Red Hat Enterprise Linux kernel, investigating if software metrics can be used in vulnerability prediction. They considered complexity, code churn, and developer activity metrics. The results showed that the metrics are discriminative and predictive of vulnerabilities. However, their model was also built on file level, while we are predicting vulnerable functions.

Chowdhury et al. [9] created a framework that can predict vulnerabilities mainly relying on the CCC (complexity, coupling, and cohesion) metrics [17]. They also compared four statistical and machine learning techniques (namely C4.5 Decision Tree, Random Forests, Logistic Regression, and Naive Bayes classifier). The authors concluded that decision-tree-based techniques outperformed statistical models in their case. We also found that tree-based classifiers perform well for vulnerable JavaScript function prediction.

Morrison et al. [8] built a model – replicating the vulnerability prediction model by Zimmermann et al [3] – for both binaries and source code at file level. The authors checked several learning algorithms including SVM, Naive Bayes, random forests, and logistic regression. On their dataset, Naive Bayes and random forests performed the best. In our setup, the Naive Bayes algorithm was the worst performer, while random forest achieved good results.

Yu et al. introduced HARMLESS [7], a cost-aware active learner approach to predict vulnerabilities. They used a support vector machine based prediction model with under-sampled training data, and a semi-supervised estimator to estimate the remaining vulnerabilities in a code base. HARMLESS suggests which source code files are most likely to contain vulnerabilities. They also used Mozilla’s code base in their case study, with 3 different feature sets: metrics, text, and the combination of text mining and crash dump stack traces. The same set of source code metrics were used than that of Shin et al. [6].

All the above works target file-level vulnerability prediction, while we address the prediction of vulnerable JavaScript functions in our current work.

III. APPROACH

A. Dataset collection method

To build machine learning models, we needed a training dataset with features of JavaScript functions manually labeled as vulnerable or non-vulnerable. The overview of the data mining process we performed is shown in Figure 1.

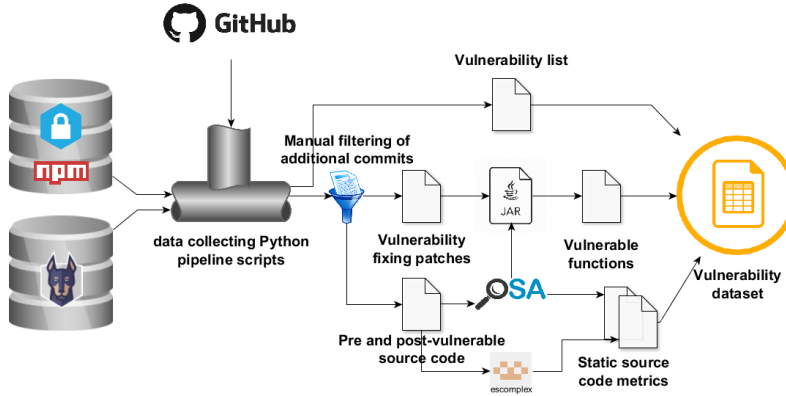


Figure 1. Data processing overview

1) *Processing nsp and Snyk and linking them with GitHub*: We leveraged two publicly available vulnerability databases, nsp (the Node Security Platform, which is now part of npm) [12] and the Snyk Vulnerability Database [13]. Both of these projects aim to analyze programs for vulnerable third party module usages. They have command line and/or web-based interfaces, which can inspect an arbitrary Node.js module to find external dependencies with known vulnerabilities. To achieve this, they utilize a list of known vulnerabilities to look for security issues in the particular version of an external module the programs depend on. We extracted and processed these vulnerability databases.

As for nsp, we used its command line interface to collect vulnerability data. It provides a *gather* command that saves its internal list of vulnerabilities into a JSON file. Snyk has an online repository of known vulnerabilities, but there is no possibility for downloading the entries. Nonetheless, Snyk maintains a GitHub mirror¹ of its vulnerability database with monthly synchronization. We used the content of this GitHub repository in case of Snyk (accessed on 27/05/2018).

The main issue with these extracted raw vulnerability sources is that they contain unstructured data. The entries include human readable description of vulnerabilities with URLs of fixing commits, pull requests or issues in GitHub or other repositories. However, these URLs are somewhat arbitrary, they can appear on multiple places within the entries and any of them might be missing entirely. To handle this, we wrote a set of Python scripts to process these vulnerability entries and create an internal augmented and structured representation of them. The scripts collected all the URLs from each entry $vuln_i$ and kept all those pointing to GitHub. Traversing these URLs, we derived a set of fixing commits (commits pointing to the state of the system where a security vulnerability has been fixed, thus they already contain the mitigation code) for each $vuln_i$ using the GitHub REST API following these steps:

- 1) If the URL pointed to a particular commit, we put the appropriate commit hash to the fixing list (fix_i)

- 2) If the URL pointed to a pull request or merge request, we put all the commit hashes in the request to fix_i
- 3) If the URL pointed to an issue, we traversed through the comments of the issue and collected all mentioned URLs into a separate list for manual validation.

If the separate list for manual validation was not empty, we manually checked all the commit URLs in it and put only those commits into fix_i that were indeed related to the fix of the original vulnerability issue ($vuln_i$). The manual validation was performed by one of the authors, while another author participated in the discussion and resolution of problematic cases. The added commits usually introduced unit tests or some corrections if the first fix was incomplete.

We note that it is possible that a commit which was referenced in the dataset entry (i.e. fix_i) contained tangled code changes (i.e. pull or merge requests). To lower the risk, we performed a random cross-check on several of these large commits, but found no tangled changes in our sample.

Upon finalizing the fixing commit lists for each entry, we collected all their code modifications in the form of a combined patch file ($patch_i$) that contained all the modifications from the fixing commits. We used the GitHub API again to collect this information. Moreover, we identified the parent commit of the first commit in time belonging to the vulnerability fix (sha_{pre}) for each system. Version sha_{pre} was used to assign the labels 1 or 0 to functions indicating if the function contained a vulnerability or not. The final dataset was assembled from all the sha_{pre} versions of the functions in the systems. We marked all functions that were affected by any of the vulnerability fixing modifications (i.e. $patch_i$ changed those functions) as vulnerable. All the other functions of the JavaScript programs were marked as non-vulnerable. We note that all the test functions (i.e. functions contained in files under “test” folders) have been filtered out as these would only distort the prediction models.

2) *Mapping patches with JavaScript functions*: To perform the mapping of patches to functions, we used the patch files ($patch_i$ for each vulnerability $vuln_i$ collected by the process described in Section III-A1) of the vulnerability-

¹<https://github.com/snyk/vulnerabilitydb>

fixing commits in a unified diff format. Each diff contains a header information specifying the name of the original and the new files. After that, there are one or more change hunks that contain the actual line differences and each hunk begins with range information about the modification. We checked whether any function falls into this range. We achieved this by using the source code positions of the functions – begin and end line numbers, which were produced by the OpenStaticAnalyzer tool – and checked whether these two ranges intersect or not. An example is shown in Listing 1.

```

1 --- /path/to/original.js    timestamp
2 +++ /path/to/new.js       timestamp
3 @@ -4,1 +4,2 @@
4 + var tmp = bar(i);
5 + return tmp;
6 - return bar(i);

```

Listing 1. Example diff file

```

1 function foo(a) {
2   var i = 4 * a;
3   // call bar
4   var tmp = bar(i);
5   return tmp;
6 }

```

Listing 2. Example JavaScript function

The source position of the *foo* function is [1,6] and the range from the diff is [4,5]. They intersect, so our method incorporates the *foo* function into the dataset. With this algorithm, we found all the functions that were changed by each vulnerability fixing commit, which we mapped to their previous versions (in *sha_{pre}*) to mark them vulnerable in the version prior to the first fixing commit.

3) *Static source code metrics*: For predictors (or, features), we used static source code metrics. We calculated the metrics for the functions included in the final dataset using two tools, escomplex [15] and OpenStaticAnalyzer (OSA) [14]. Both OpenStaticAnalyzer [18] and escomplex [19], [20] were used and referenced in related research works, thus we consider them to be reliable. The list of calculated metrics is shown in Table I. Please note that similar metrics are grouped together in one line, so the total number of calculated metrics is 35.

4) *Dataset structure*: The final dataset structure follows a simple CSV format that is easy to feed into many machine learning frameworks. Each line of the CSV file represents a function from a Node.js program. The 1st column is a short name, while the 2nd is the qualified name of the function generated by the algorithm described in Section III-A2. The 3rd column shows the path of the containing JavaScript source file, while the 4th column contains a GitHub URL to the analyzed JavaScript source file (in the *sha_{pre}* version). The 5th and 6th columns contain the starting, while the 7th and 8th the ending line and column information, respectively. Columns 9 to 43 contain the calculated metric values listed in Table I. The last column (column 44) contains the flag indicating whether the function is vulnerable or not.

²Total means that the metric is calculated for the actual code element including all the contained elements recursively.

Table I
CALCULATED STATIC SOURCE CODE METRICS

Metric	Description	Tool
CC	Clone Coverage	OSA
CCL	Clone Classes	OSA
CCO	Clone Complexity	OSA
CI	Clone Instances	OSA
CLC	Clone Line Coverage	OSA
LDC	Lines of Duplicated Code	OSA
McCC, CYCL	Cyclomatic Complexity	OSA, escomplex
NL	Nesting Level	OSA
NLE	Nesting Level without else-if	OSA
CD, TCD	(Total ²) Comment Density	OSA
CLOC, TCLOC	(Total) Comment Lines of Code	OSA
DLOC	Documentation Lines of Code	OSA
LLOC, TLLOC	(Total) Logical Lines of Code	OSA
LOC, TLOC	(Total) Lines of Code	OSA
NOS, TNOS	(Total) Number of Statements	OSA
NUMPAR, PARAMS	Number of Parameters	OSA, escomplex
HOR_D	Nr. of Distinct Halstead Operators	escomplex
HOR_T	Nr. of Total Halstead Operators	escomplex
HON_D	Nr. of Distinct Halstead Operands	escomplex
HON_T	Nr. of Total Halstead Operands	escomplex
HLEN	Halstead Length	escomplex
HVOC	Halstead Vocabulary Size	escomplex
HDIFF	Halstead Difficulty	escomplex
HVOL	Halstead Volume	escomplex
HEFF	Halstead Effort	escomplex
HBUGS	Halstead Bugs	escomplex
HTIME	Halstead Time	escomplex
CYCL_DENS	Cyclomatic Density	escomplex

The created vulnerability dataset³ consists of 12,125 JavaScript functions from which 1,496 are vulnerable.

B. Dataset analysis approach

We employed 8 different types of machine learning algorithms on the vulnerability dataset created with the method described in Section III-A. These algorithms were two deep neural network variants, a simple (DNN_s) and a complex one (DNN_c), the K-Nearest Neighbors algorithm (KNN), a decision tree classifier (Tree), the C-Support Vector Classification variant of Support Vector Machine algorithm (SVM), Random Forest (Forest), Logistic regression (Logistic), Linear regression (Linear) and the Gaussian Naive Bayes algorithm (Bayes). The deep neural network algorithms were implemented in the *TensorFlow* [21] framework⁴, while we used *scikit-learn*⁵ to run all the other algorithms. Both frameworks were used in a Python environment. We could not use only one of them because while TensorFlow has a strong support for deep learning, it does not contain all the classic algorithms. In contrast, scikit-learn is very strong in classic machine learning algorithms but it is not a deep learning framework in itself.

DNN_s stands for the base DNN algorithm implemented in TensorFlow. We used it without any modifications except for changing the parameters it provides (see Section III-B1). DNN_s learning runs for a fixed number of iterations over all the training instances (i.e. epochs). DNN_c is our own modified strategy for training a DNN. It uses an adaptive learning rate method where the learning rate parameter is not constant over the course of training. We start with a relatively

³<http://www.inf.u-szeged.hu/~ferenc/papers/JSVulnerabilityDataSet/>

⁴<https://www.tensorflow.org/>

⁵<http://scikit-learn.org/stable/>

high learning rate parameter and continue the classic back propagation algorithm until there is no improvement in the value of F-measure (we call this a *miss*). Then we reduce the learning rate parameter to half, restore the previous model state, and continue the learning process from there. We repeat these steps until we get 4 misses in succession, then terminate the algorithm and return the last, best performing model. This strategy reduces the likelihood of the algorithm getting “stuck” in a local optimum. Regarding KNN, Tree, SVM, Forest, Logistic and Linear regression, and the Naive Bayes algorithm, we used their scikit-learn implementation.

1) *Grid search for the best parameters*: To find the best performing configuration of each algorithm, we applied a grid search approach [22] on the hyper parameters of the learning algorithms. It means that we defined various values for machine learning algorithm parameters and trained multiple models using various combinations of hyper parameters. After having multiple results for each model, we could select the best performing ones.

For all training sessions we divided the training data into three sets, *train*, *dev*, and *test* in a 80%, 10%, 10% proportion, respectively, and used a 10-fold cross-validation. At the end of the 10 folds, we calculated the precision, recall, and F-measure values. For selecting best performing parameter configurations, we relied only on the results of the dev set. This ensured that we did not use information for selecting the best parameters from our final test set in any way. We used F-measure as our primary performance indicator as in the security domain both precision and recall are important.

2) *Sampling strategies*: In our assembled vulnerability dataset, only slightly more than 10% of the functions were marked as vulnerable. This highly imbalanced nature of the training set is usually unwanted as prediction models might be distorted by these skewed distributions.

A common way of handling such situations is the usage of random under or over-sampling strategies [23]. Random under-sampling means we randomly throw away training instances from the larger set until we reach a pre-defined ratio between the two classes. Random over-sampling is when we randomly repeat training instances from the smaller set until we reach a pre-defined ratio between the two classes.

We repeatedly ran our algorithm parameter grid search (see Section III-B1) with the following re-sampling strategies: *no re-sampling* (None); *over-sampling* (\uparrow) with ratios 25%, 50%, 75% and 100%; *under-sampling* (\downarrow) with ratios 25%, 50%, 75% and 100%.

IV. RESULTS

We trained 9 different prediction models on the created vulnerability dataset (8 different algorithms, but two variants of DNN) on a desktop PC⁶ using both CPU and GPU. The running times varied between 6-12 hours for a complete hyper-parameter grid-search of all algorithms. We repeated these grid-search sessions for all the separate over and

⁶8 core 2.4GHz CPU, NVIDIA Titan Xp GPU, 8GB RAM

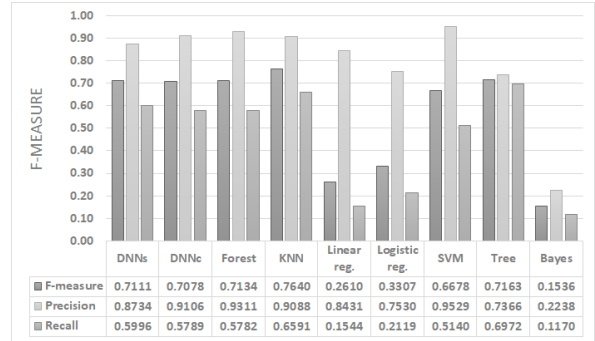


Figure 2. Results on the imbalanced dataset

under-sampling strategies (described in Section III-B2), thus building all the models took a considerable amount of time and computing resources.

A. Results on the imbalanced dataset

First, we ran our grid-search without applying any re-sampling on the vulnerability dataset, which is highly imbalanced (out of 12,125 functions only 1,496 are vulnerable). The performances of the 9 models with their best parameter combinations is displayed in Figure 2.

The overall results are surprisingly good given the fact that JavaScript is a highly dynamic language and we used only static source code metrics as predictors. Five out of the 9 models (DNN_s, DNN_c, Forest, KNN, and Tree) achieved an F-measure of over 0.70 and SVM was also very close with 0.67. It is interesting to note that for all algorithms, precision values were significantly higher than recall, except for the decision tree classifier, which had a precision of 0.74, a recall of 0.7 and an F-measure of 0.72.

Only the Naive Bayes algorithm was clearly incapable of producing a viable prediction model using the original, imbalanced vulnerability dataset. Logistic and linear regression achieved a precision of 0.75 and 0.84, respectively, which are relatively high values, however, they had a very low recall (0.21 and 0.15, respectively) that decreased the F-measure values.

As a simple baseline, we also ran the ZeroR algorithm with the setup that it predicted all instances to be vulnerable (and not vice versa as the default setup would do, because if we predicted every instance to be non-vulnerable, all our IR metrics would have been 0). ZeroR achieved a precision of 0.12 and perfect recall of 1 (as it found all the vulnerable instances), which adds up to an F-measure of 0.21. This result is much worse than those of the other algorithms’ except for the Naive Bayes. Therefore, we can already answer RQ1 based on these results.

RQ1: Choosing a suitable algorithm with proper parameters, it is possible to create efficient function level vulnerability prediction models using only static source code metrics as predictors. The DNN, KNN, Forest, and Tree algorithms all achieved F-measures above 0.70 without any re-sampling on the dataset.

Table II
F-MEASURES ACHIEVED BY THE MACHINE LEARNING ALGORITHMS

Alg.	None	↑25%	↑50%	↑75%	↑100%	↓25%	↓50%	↓75%	↓100%	Rand
DNN _s	0.71	0.71*	0.71	0.65	0.68	0.70	0.71	0.69	0.59	0.05
DNN _c	0.71	0.70	0.71	0.68	0.65	0.71*	0.71	0.68	0.66	0.01
Forest	0.71	0.74*	0.74	0.73	0.72	0.72	0.72	0.72	0.65	0.05
KNN	0.76*	0.75	0.72	0.6935	0.6817	0.76	0.75	0.74	0.64	0.14
Lin. reg.	0.26	0.48	0.55*	0.49	0.45	0.30	0.37	0.51	0.44	0.02
Log. reg.	0.33	0.50	0.57*	0.55	0.49	0.38	0.45	0.53	0.49	0.01
SVM	0.67	0.70	0.72*	0.70	0.68	0.67	0.67	0.67	0.65	0.16
Tree	0.72*	0.71	0.71	0.71	0.70	0.70	0.69	0.67	0.59	0.15
Bayes	0.15	0.16	0.16	0.21*	0.20	0.16	0.16	0.18	0.17	0.07
Median	0.71	0.70	0.71*	0.68	0.68	0.70	0.69	0.67	0.59	0.05

B. Comparison of the models based on the complete results

The best performing model results based on the complete grid-search using various re-sampling strategies are summarized in Table II. Each column of the table contains model results (in terms of F-measure⁷) using a particular re-sampling strategy (see Section III-B2) with the best parameters found by the grid-search method. The first column shows the results on the original imbalanced dataset without re-sampling (in line with Figure 2). The next four columns display the results on the over-sampled, while the following four on the under-sampled dataset. The last column presents results on a random sanity check. To make sure that having these strong prediction results is not coincidental, we created a new training dataset by reassigning the 1,496 vulnerable labels randomly. The training results on this randomly labeled dataset shows that models cannot learn to distinguish arbitrary set of functions based on their static source code metrics, thus our prediction results are unlikely to be the consequences of random factors.

The gray cells in the table mark the best performing algorithm with the given re-sampling strategy. KNN is the best in five different re-sampling configurations, Forest in three, while DNN_c in one. The values indicated in bold and with an asterisk are the best F-measure values for a given machine learning algorithm (i.e. the highest value in the row). The most important thing to note here compared to the results on the imbalanced training set is that even SVM achieved a result above 0.70 with an appropriate over-sampling strategy (↑50%, ↑75%). Seven out of the nine models achieved better performances in some of the re-sampling configurations than on the original, imbalanced dataset. The exact composition of precision and recall values leading to this F-measures are visualized in Figure 3. Based on the data in Table II and Figure 3, we can answer RQ2 as follows.

RQ2: The best performing algorithm for predicting vulnerable JavaScript functions in terms of F-measure was KNN with an F-measure of 0.76 (0.91 precision and 0.66 recall). The best precision (0.95) was achieved by SVM, while the best recall (0.80) by KNN. In overall, KNN, DNN, SVM, Tree, and Forest are equally well-suited for the task, while the regressions as well as the Naive Bayes algorithm perform much worse.

⁷Matthews correlation coefficients (MCC) were slightly smaller in general, but they showed the same tendency, see the shared dataset for details.

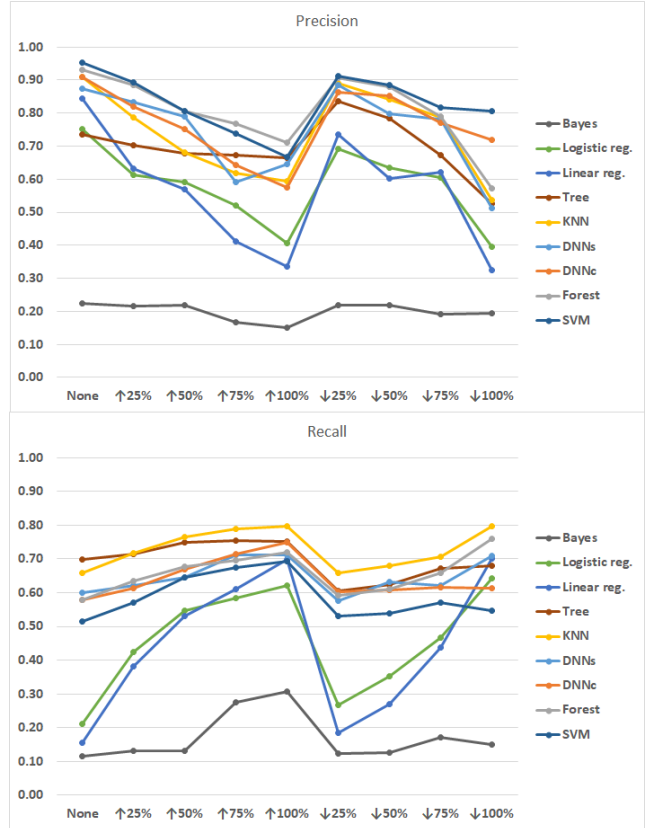


Figure 3. Impact of re-sampling on the learning precision and recall

V. THREATS TO VALIDITY

Our data collection process might not be 100% accurate as only the additional candidate commits collected from issue comments were validated manually. The original data sources might contain errors as well as our automatic patch collection and patch-to-function mapping algorithms might introduce inconsistencies. We tried to mitigate this problem by thorough code review of our scripts and programs.

We mapped static source code analysis results of various tools and functions identified in patches by line information. This is another source of possible errors, but we performed a small evaluation on 20 randomly selected JavaScript functions from the dataset and found no multiple functions in the same line. Based on this and our past experience, we believe it is a safe assumption that multiple functions in the same code line are very rare in a non-minified JavaScript program. As we used line information only within the same version of the programs, the likelihood of mismatching functions is even more negligible.

The extraction of features (i.e. static source code metrics) is heavily dependent on the accuracy of the tools used, which may threaten the extraction process. However, there are numerous related works using the same tools, thus they can be considered stable. Moreover, we manually double-checked some of the calculated metric values and found no problems in their calculation.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we published a novel JavaScript vulnerability dataset to be used for building prediction models. The dataset contains various JavaScript functions together with their static source code metrics and a flag indicating whether the function contains a vulnerability or not. This information was assembled by mining public vulnerability data sources of nsp and Snyk and collecting fixing patches from GitHub.

We presented an assessment of existing machine learning algorithms for building function level vulnerability prediction models using this dataset. We analyzed the performances of 8 different types of algorithms using the training set as is, and also by applying various re-sampling strategies.

Our results show that even for such a highly dynamic language as JavaScript, static source code metrics are suitable predictors of vulnerabilities. However, we experienced large variances in prediction performances depending on the applied sampling strategy and hyper-parameters. Using the appropriate machine learning algorithm (DNN, KNN, Tree, Forest, or SVM) and suitable hyper-parameters, a prediction with F-measure of 0.7 and above can be achieved. Nonetheless, there is a clear trade-off between precision and recall; over-sampling tends to improve recall, but decreases precision, while intensive under-sampling improves precision, but reduces recall significantly.

We plan to extend the set of predictors with history and textual metrics in order to further improve vulnerability prediction at the level of JavaScript functions.

ACKNOWLEDGMENT

The research has been supported by the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme. Ministry of Human Capacities, Hungary grant 20391-3/2018/FEKUSTRAT is acknowledged. The Titan Xp used for this research was donated by the NVIDIA Corporation.

REFERENCES

- [1] N. R. Mead, J. H. Allen, M. Ardis, T. B. Hilburn, A. J. Kornecki, R. Linger, and J. McDonald, "Software assurance curriculum project volume 1: Master of software assurance reference curriculum," CARNEGIE-MELLON UNIV. PITTSBURGH PA SOFTW. ENG. INST., Tech. Rep., 2010.
- [2] Y. Shin and L. A. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, pp. 25–59, 2011.
- [3] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *2010 Third International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2010, pp. 421–428.
- [4] M. Jimenez, Y. Le Traon, and M. Papadakis, "Enabling the Continous Analysis of Security Vulnerabilities with Vul-Data7," in *IEEE International Working Conference on Source Code Analysis and Manipulation*, 2018, pp. 56–61.
- [5] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the ACM Conference on Computer and Communications Security*, 01 2007, pp. 529–540.
- [6] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov. 2011.
- [7] Z. Yu, C. Theisen, H. Sohn, L. Williams, and T. Menzies, "Cost-aware vulnerability prediction: the HARMLESS approach," *CoRR*, vol. abs/1803.06545, 2018.
- [8] P. Morrison, K. Herzig, B. Murphy, and L. A. Williams, "Challenges with applying vulnerability prediction models," in *HotSoS*, 2015.
- [9] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011.
- [10] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, pp. 315–317.
- [11] A. Gkortzis, D. Mitropoulos, and D. Spinellis, "VulinOSS: a dataset of security vulnerabilities in open-source systems," in *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 2018, pp. 18–21.
- [12] "Node Security Platform - GitHub," <https://github.com/nodesecurity/nsp>, Accessed: 2018-10-16.
- [13] "Vulnerability DB — Snyk," <https://snyk.io/vuln>, Accessed: 2018-10-16.
- [14] "OpenStaticAnalyzer - GitHub," <https://github.com/sed-inf-uszeged/OpenStaticAnalyzer>, Accessed: 2018-10-16.
- [15] "escomplex - GitHub," <https://github.com/escomplex/escomplex>, Accessed: 2018-10-16.
- [16] M. Siavvas, D. Kehagias, and D. Tzovaras, "A preliminary study on the relationship among software metrics and specific vulnerability types," in *2017 International Conference on Computational Science and Computational Intelligence – Symposium on Software Engineering (CSCI-ISSE)*, 12 2017.
- [17] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.
- [18] E. Pengő and P. Gál, "Grasping primitive enthusiasm - approaching primitive obsession in steps," in *Proceedings of the 13th International Conference on Software Technologies (ICSOFT)*, 2018, pp. 423–430.
- [19] K. C. Chatzidimitriou, M. D. Papamichail, T. Diamantopoulos, M. Tsapanos, and A. L. Symeonidis, "Npm-miner: An infrastructure for measuring the quality of the npm registry," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 42–45.
- [20] C. L. Mariano, "Benchmarking javascript frameworks," Ph.D. dissertation, Dublin Institute of Technology, 2017.
- [21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.
- [22] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in neural information processing systems*, 2011, pp. 2546–2554.
- [23] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *SIGKDD Explor. Newsl.*, vol. 6, no. 1, pp. 20–29, Jun. 2004.