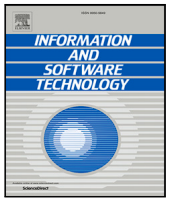
Information and Software Technology 146 (2022) 106844

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Featherweight assisted vulnerability discovery

David Binkley [a,*], Leon Moonen [b], Sibren Isaacman [a]

[a] Loyola University Maryland, 4501 N. Charles St., Baltimore, MD 21210-2699, USA
[b] Simula Research Laboratory, Oslo, Norway

ARTICLE INFO

Keywords:
Model interpretability
Vulnerability prediction
Identifier splitting
Source code vocabulary
Software security

ABSTRACT

Predicting vulnerable source code helps to focus the attention of a developer, or a program analysis technique, on those parts of the code that need to be examined with more scrutiny. Recent work proposed the use of function names as semantic cues that can be learned by a deep neural network (DNN) to aid in the hunt for vulnerability of functions.

Combining identifier splitting, which we use to split each function name into its constituent words, with a novel frequency-based algorithm, we explore the extent to which the words that make up a function's name can be used to predict potentially vulnerable functions. In contrast to the *lightweight* prediction provided by a DNN considering only function names, avoiding the need for a DNN provides *featherweight* prediction. The underlying idea is that function names that contain certain "dangerous" words are more likely to accompany vulnerable functions. Of course, this assumes that the frequency-based algorithm can be properly tuned to focus on truly dangerous words.

Because it is more transparent than a DNN, which behaves as a "black box" and thus provides no insight into the rationalization underlying its decisions, the frequency-based algorithm enables us to investigate the inner workings of the DNN. If successful, this investigation into what the DNN does and does not learn will help us train more effective future models.

We empirically evaluate our approach on a heterogeneous dataset containing over 73 000 functions labeled vulnerable, and over 950 000 functions labeled benign. Our analysis shows that words alone account for a significant portion of the DNN's classification ability. We also find that words are of greatest value in the datasets with a more homogeneous vocabulary. Thus, when working within the scope of a given project, where the vocabulary is unavoidably homogeneous, our approach provides a cheaper, potentially complementary, technique to aid in the hunt for source-code vulnerabilities. Finally, this approach has the advantage that it is viable with orders of magnitude less training data.

1. Introduction

Security vulnerabilities in source code are a key quality concern in software development. Exploitation of vulnerabilities may cause financial damage, decrease users' trust, and, depending on the domain, introduce personal risks. This makes identifying vulnerabilities in the early stages of software development useful. Automated software inspections have proven effective at identifying certain classes of security vulnerabilities in source code [1–4], but at the same time suffer from a considerable number of false positives [5–7]. Manual code reviews, or software inspections [8], have fewer problems with false positives, but suffer from the sheer volume of code that must be inspected [9–11]. Thus, methods that help focus the attention of a developer or program analysis technique on those parts of the code that should be examined with more scrutiny have the potential to lower false positives and overall workload.

Li et al. recently proposed LAVDNN as a lightweight approach that uses function names as semantic cues that can be learned by a Deep Neural Network (DNN) [12]. To be clear, LAVDNN is not intended as a replacement for more involved techniques that use a multitude of code features from various levels of code granularity, both for general defect prediction [13–15], and vulnerability specific methods [16–19]. Instead, it is used to triage the code and thus assist a developer in deciding where to manually inspect the code or apply more sophisticated techniques.

In a similar vein, the goal of our work is *not* to outperform the state of the art in defect prediction. Instead, we have two goals related to a complementary approach. First, we seek to study the viability of a novel word-frequency-based approach, and second we aim to use this approach to provide a level of interpretability to the LAVDNN

* Corresponding author.
E-mail address: binkley@cs.loyola.edu (D. Binkley).

https://doi.org/10.1016/j.infsof.2022.106844
Received 4 August 2021; Received in revised form 4 December 2021; Accepted 9 January 2022
Available online 22 January 2022

model. If, after tuning, the frequency-based approach manages reasonable performance, it then suggests future opportunities to improve vulnerability predictors such as LAVDNN by augmenting them with information gleaned from what we refer to as dangerous words. Though a "dangerous word" is not in and of itself a danger, the word may be a red flag, much in the same way that a code smell is not a problem in and of itself, but suggests a point of potential concern. If successful, the frequency-based approach provides a *featherweight* alternative that avoids the need to construct a DNN, which, among other things, finds the approach viable using orders of magnitude less training data.

The paper also investigates the degree to which LAVDNN is leveraging the presence of dangerous words. If the frequency approach provides similar performance, then it suggests that we have succeeded in interpreting the learning of the DNN as we have evidence that LAVDNN learns to identify dangerous words. On the other hand, a difference in performance indicates that LAVDNN learns something orthogonal to the words. Thus a contribution of our work includes its exploration of model interpretability. For example, although Li et al. claim impressive results (with F_2-scores reaching 0.910 and 0.915 for, respectively, C/C++ and Python programs), our experiments, as well as the data presented in their work [12, Table 11], show a steep drop-off in efficacy on real-world systems. Nevertheless, the impressive published performance of LAVDNN both raises the question of "what makes it tick" and also makes it a prime candidate for further study. Understanding *why* and *when* LAVDNN is successful may have implications for understanding how to better discover vulnerable functions universally. It is therefore instructive to try and understand *what* LAVDNN is actually learning, either to improve LAVDNN itself, or to develop complementary techniques. For example, are certain words, abbreviations, or other language patterns indicative of vulnerabilities?

Contributions: We investigate how the individual words that make up each function name affect vulnerability predictions:

- We present a featherweight approach, FAVD (Featherweight Assisted Vulnerability Discovery), that uses the notion of *dangerous words* as semantic cues. The underlying idea is that when developers choose semantically sensible names, a vulnerable function is more likely to be given a name that contains dangerous words.
- We explore two methods for identifying dangerous word: the first, FAVD_L, uses LAVDNN [12] as classifier to determine if a word should be considered dangerous. FAVD_L's performance provides insights into the role that dangerous words play in the LAVDNN model. The second method for identifying dangerous words, FAVD_F, identifies dangerous words based on the *frequency* of the words in the names of known vulnerable and benign functions.
  Comparing these two with LAVDNN enables us to provide insight into what is learned by the otherwise black-box approach used by LAVDNN. To begin with, the comparison of LAVDNN and FAVD_L tells us about the use of words by LAVDNN, but tells us nothing about the absolute value of those words in the prediction. FAVD_F provides that baseline. For example, if FAVD_F performs worse than FAVD_L then we can assert that LAVDNN is making use of features beyond words.
- We empirically evaluate the predictive ability of FAVD_L and FAVD_F using nine datasets ranging in vocabulary diversity. Our analysis shows that words alone account for a significant portion of the DNN's classification ability especially with more homogeneous vocabularies. Hence, it is feasible to train a featherweight "triage predictor" using the function names associated with past vulnerabilities of a mature project to gain an initial focus. Furthermore, this technique requires orders of magnitude less training data, and can thus easily complement existing techniques for vulnerability prediction.

The paper is organized as follows: Sections 2 and 3 present the background and the approach itself. Sections 4 and 5 introduce our research questions and experimental design, followed by a discussion of results in Section 6. We survey related work in Section 7 and conclude in Section 8.

2. Background

2.1. LAVDNN

The LAVDNN model was trained on an (undisclosed[1]) dataset of 8 525 vulnerable function names extracted from the Common Vulnerabilities and Exposures (CVE) database, and 8 000 benign function names extracted from open-source projects [12]. Each name was one-hot encoded into a matrix of 66 rows (for the allowed alphanumeric characters) and 50 columns (for the allowed maximum function-name length), which together with its label as benign or vulnerable, was used to train a multi-layer Bidirectional Long Short Term Memory (LSTM) network for classification. LAVDNN concludes with two densely connected output nodes whose values are run through a softmax function. Thus, the network outputs the likelihoods that a function is "vulnerable" or "benign" as values in the range 0.0–1.0. The authors experimentally determine that a threshold value of 0.55 for the "vulnerable" class provides the best performance. Thus, function names with a score of 0.55 or greater in the "vulnerable" output node are classified as vulnerable and the rest benign.

Care must be taken when referring to the paper that introduced LAVDNN [12]. While the paper claims very impressive F_2 values, computing F_2 values using data from the paper's Table 11 produces much lower F_2 values for real-world systems (e.g., 0.683 for LibTIFF and 0.746 for FFmpeg). These values cannot be reproduced, because the paper's limited replication package, which only includes the function names from these two systems, does not label them as *vulnerable* or *benign*. Fortunately, Lin et al. [20] independently provide the necessary data, albeit for slightly newer versions of LibTIFF and FFmpeg. However, as shown in the last column of Table 4, discussed later, applying LAVDNN to this data results in the notably lower F_2 scores of 0.292 for LibTIFF and 0.083 for FFmpeg.

2.2. Identifier splitting

Identifier splitting [21] splits an identifier into its constituent parts, called *terms*. For example, the identifier read_file includes the terms read and file. Splitting algorithms range from conservative (looking for Camel and Snake case) to aggressive (e.g., able to separate maxstrlen into max, str, and len) [22].

3. Approach

This section describes FAVD, our algorithm for *featherweight assisted vulnerability discovery*, the core of which is given as Algorithm 1. The algorithm takes as input three parameters and outputs the set of identifiers predicted to be vulnerable, $\mathcal{V}$. The output is a subset of the first input parameter, $\mathcal{I}$, which is the set of identifiers being tested. The second input parameter is the training data, $\mathcal{T}$, which is a set of identifiers each labeled as *vulnerable* if it is the name of a function with a vulnerability and *benign* otherwise. The final input parameter, min_score, is the minimum score that an identifier must receive to be returned by Rank as a dangerous word. The algorithm first conservatively splits each function name from the training data into its constituent terms. The resulting set of terms is used as the source of potential dangerous words:

[1] We have requested this data from the authors but could not obtain it.

Definition 1. A word is a *dangerous word* if its presence in a function's name correlates with the function being more likely to include a vulnerability.

For example, functions that accept user input are often vulnerable to various stack attacks. The names of such functions often include words such as *read* or *input*. Thus, we may classify *read* and *input* as "dangerous," marking functions using those terms as potentially vulnerable.

FAVD's primary goal is converting a set of terms into a ranked list of dangerous words. This is done by the function Rank, which takes the set of terms and a minimum (dangerousness) score. This function first assigns a dangerousness score to each term and then discards those terms whose score is less than the given minimum, min_score. It returns a list of the remaining terms in decreasing order of dangerousness.

FAVD next calls the function FindBest, which takes the list of dangerous words and the training data, and outputs two values: cutoff and threshold. The cutoff determines how many of the dangerous words are retained when predicting the vulnerability of the test data found in $\mathcal{I}$. If the percentage (see Algorithm 1) of a function-name's terms that are dangerous is greater than threshold, then the function is predicted to be vulnerable.

FindBest searches for a *winning combination* of these two. For example, having a small cutoff means a short list of dangerous words, which often works better with a low threshold, because with few dangerous words, most identifiers will have at most a few dangerous terms. On the other hand, when cutoff is large, there are lots of dangerous words, and a higher threshold often works better.

As mentioned in Section 1, we experiment with two different vulnerability discovery algorithms, FAVD_L and FAVD_F. These two differ only in the implementation of the function Rank. In the ideal case, Rank returns exactly those words that will identify the vulnerable identifiers in the test data, $\mathcal{I}$. We approximate this ideal using two different ranking functions, $\mathcal{R}_L$ and $\mathcal{R}_F$. The first, $\mathcal{R}_L$, uses LAVDNN to determine each term's dangerousness. Here, each term found in a function name is fed into LAVDNN in isolation, thereby producing a score for the term between 0.00 and 1.00.

The second ranking function, $\mathcal{R}_F$, is based on term frequency, and is thus entirely independent of LAVDNN. This enables us to better understand the value that the words that make up a function's name play in LAVDNN's assessment. It produces integer term scores as follows: for each vulnerable identifier in the training data it increments the dangerousness score of all of the identifier's terms by the constant plus, while for each benign identifier it decrements this score by the constant minus. Using different pairs of constants, referred to as *weights*, allows $\mathcal{R}_F$ to place more or less emphasis on terms frequently found in the vulnerable or benign training data.

4. Research questions

RQ1 *What is the result of being excessively conservative and declaring all functions vulnerable?* – While simplistic, this is the lightest-weight of approaches and is guaranteed to have high recall, so it sets a good baseline.

input : test identifiers $\mathcal{I}$, labeled training data $\mathcal{T}$, min_score
output : set of vulnerable identifiers, $\mathcal{V}$
terms ← Unique (Split ($\mathcal{T}$))
dangerous ← Rank (terms, min_score))
cutoff, threshold ← FindBest (dangerous, $\mathcal{T}$)
for *id* ∈ $\mathcal{I}$ **do**
  terms ← Unique (Split (id))
  percentage ← |terms ∩ dangerous[1..cutoff]|/|terms|
  $\mathcal{V}$ ← $\mathcal{V}$ ∪ { id } if percentage > threshold
end

Algorithm 1: FAVD vulnerability prediction.

Table 1
Overview of the nine datasets used in the study. Note that the ninth dataset, within, aggregates the first six.

	Dataset	Vulnerable	Benign	% vuln.	Overlap
within	Asterisk	49	10 102	0.5%	2
	FFmpeg	184	4 379	4.2%	18
	LibPNG	31	491	6.3%	0
	LibTIFF	75	522	13.6%	8
	Pidgin	26	6 722	0.3%	0
	VLC	37	2 699	1.4%	3
	loo	402	24 906	1.6%	33
	VDISC	72 612	932 741	7.2%	11 970

RQ2 *Can LAVDNN be used to construct a list of dangerous words that can effectively predict vulnerable functions?* – In other words, can LAVDNN predict which terms are associated with vulnerable functions?

RQ3 *Does direct construction of a list of dangerous words provide insight to what LAVDNN learns?* – In order to investigate the potential value that terms might bring to the DNN, we use *term frequency* as an alternative method of determining the list of dangerous words.

5. Experimental design

5.1. Datasets and ground truth

We consider the nine datasets shown in Table 1 (note that the ninth dataset, within, aggregates the first six datasets). For each dataset, we work exclusively with lists of benign and vulnerable function names. We clean each list by removing "internal" duplicates (caused when two or more functions share the same name). For each dataset, we make the lists of benign and vulnerable names disjoint, taking the conservative stance that names appearing on both lists are potentially vulnerable. A replication package with our data is available on GitHub and Zenodo.[2]

The first six datasets in Table 1 come from data shared by Lin et al. [20]. They extracted 457 vulnerable functions from six open-source projects based on CVE reports [23] and added 32 531 benign functions from each project's source code repository. Table 1 shows the sizes for each after cleaning. Asterisk is a C++ library for PBX integration and Pidgin is a library for developing chat clients. The other four projects are from more closely related domains, with FFmpeg and VLC being well-known video applications, and LibPNG and LibTIFF providing image manipulation. The main programming language for all projects is C, with small amounts of C++, Python, HTML, Shell, and Assembly. Table 2 summarizes demographic details of the six projects of the within dataset.

We first consider these six in isolation, performing k-fold cross-validation separately, per project. That is, we use the names from each project independent of the other projects. Given the identifiers in these experiments are coming from a single project, they are expected to have the least *diversity* in their vocabulary. By the *diversity* of a vocabulary, we mean the variety of words used. For example, the vocabulary built from the identifiers remove_node and remove_edge show less diversity than the one built from the identifiers remove_node and delete_edge. This gives us an indication of how well a proposed algorithm performs when applied to a mature system, where there exists vulnerability data from exploits found in older versions of the system to train against. When comparing these six to the loo and VDISC datasets, we often aggregate them by taking means. We refer to this aggregate as the within dataset.

The next dataset, loo, makes use of the same six open-source projects. However, this time we perform *leave-one-out* cross-validation

[2] https://github.com/secureIT-project/FAVD DOI: 10.5281/zenodo.5957264

Table 2
Demographic details characterizing the projects in our dataset (src: OpenHub [24]).

Features	Asterisk	FFmpeg	LibPNG	LibTIFF	Pidgin	VLC
number of contributors	302	1 968	58	64	790	924
total LOC	2 529k	1 248k	462k	267k	224k	717k
estimated effort (years)	741	356	122	68	52	197
number of commits	96 265	102 835	10 679	6 447	40 605	88 567
files modified	17 777	10 082	4 724	1 830	12 199	16 012
lines added	14 051k	4 292k	2 378k	1 229k	8 321k	4 989k
lines removed	9 996k	2 316k	1 696k	938k	7 549k	3 991k
security confidence	97.84%	95.14%	85.41%	91.35%	N/A[a]	N/A[a]
vulnerability exposure	1.1‰	4.3‰	27.0‰	21.8‰	N/A[a]	N/A[a]

[a]There were no vulnerabilities reported for Pidgin and VLC in OpenHub.

on the set of six. Leave-one-out cross-validation uses the vulnerability of function names from all but one of the projects to predict the vulnerability of function names from the one left out. The names are not expected to be as similar as in the first six datasets (i.e., they have higher diversity), but because a number of the projects are from similar domains, we expect some similarity.

The largest dataset, VDISC, was extracted from the data published by Russell et al. [25]. It contains 1.27 million functions mined from open-source software, labeled for potential vulnerability by static analysis tools. After cleaning, we end up with 1 005 353 function names, including 72 612 marked as vulnerable and 932 741 marked as benign. Because some of this data is likely included in the within and loo datasets, we never combine it with either of the two, however studying them separately improves the external validity of our analysis. For example, one interesting difference is that the percentage of vulnerable identifiers in the loo dataset is only 1.6%, which is a notably smaller than the 7.2% of the VDISC dataset.

5.2. Performance measures

Our goal is to classify function names as either vulnerable or benign. We define true positives, TP, as the correctly identified vulnerable functions, true negatives, TN, as the correctly identified benign functions, false positives, FP, as any benign function identified as vulnerable, and false negatives, FN, as any vulnerable function identified as benign.

The evaluation of the quality of this classification is based on a combination of precision and recall. *Precision* is the fraction of function names determined to be vulnerable that actually are. In other words, $TP/(TP + FP)$. $TP/(TP + FP)$ *Recall* is the fraction of all vulnerable functions correctly determined to be vulnerable. In other words, $TP/(TP + FN)$.

Precision and recall often oppose each other. For example, high precision is often possible by choosing only those cases that you are very sure of, but this necessarily lowers recall. The balanced F-score, F_1, is the mean of precision and recall, and thus provides a balanced combination of the two.

However, as Li et al. observe, "in vulnerability detecting systems, it is first necessary to detect as many vulnerabilities as possible. When analyzing the source code, the false reporting may increase the workload, but failing to identify a vulnerable function is costly and unacceptable" [12]. To support this position, they use the F_2-score, which values recall over precision. In general

$$F_\beta = \frac{(1+\beta^2)\ Precision \times Recall}{\beta^2\ Precision + Recall} = \frac{(1+\beta^2)TP}{(1+\beta^2)TP + \beta^2 FN + FP}$$

For consistency, we follow Li et al. and use $\beta = 2$, and thus focus the evaluation on the F_2 score, $5TP/(5TP + 4FN + FP)$.

5.3. Procedure

The surface goal of both $\mathcal{R}_L$ and $\mathcal{R}_F$ is to retain only high-impact dangerous words. However, too high a minimum (too high a value of min_score in Algorithm 1) can starve the algorithm of sufficient vocabulary. Therefore, we consider a range of minimum scores in the experiments. For $\mathcal{R}_L$, we initially consider values between 0.00 and 1.00 stepping by 0.05. We later add a few additional values to zoom in on points of interest. For $\mathcal{R}_F$, we consider only two minimums: none, where all terms are considered dangerous, and zero, which eliminates terms of low dangerousness. This is sufficient for $\mathcal{R}_F$, because we can use the relative values of plus and minus to impact the number of terms that receive a positive score. Note that when including all words on the list of dangerous words, their relative position is still impacted by the values of plus and minus.

Beneath the surface, we are interested in forming a better understanding of the role that the terms found in function names play in LAVDNN's ability to identify dangerous functions. Thus, while the results include some direct comparisons, we are more interested in the deeper understanding that the relative performance and the relative value of dangerous words bring to the prediction.

To provide some intuition for the words ranked as the most dangerous and the least dangerous, Table 3 lists the top ten examples from each category as identified by $\mathcal{R}_F$. Over 90% of the top ten most dangerous words occur in more vulnerable names than benign names. The LibPNG name handle is a classic example of the expected behavior. Of the fifteen function names that include the word handle, ten are on the vulnerable list. In extreme cases all the names are vulnerable. For example, in Pidgin the word mxit is found in six names, all vulnerable, while the VLC word AVI is found in two names, both vulnerable. In LibPNG the word read occurs in five vulnerable names and 19 benign names. In all five vulnerable names (e.g., png_push_read_chunk) the top scoring word png also appears. None of the ten least dangerous words from LibPNG occur in a vulnerable name.

Considering the least dangerous words, most occurrences are found in benign names. With the exception of FFmpeg there are only zero to six occurrences of all ten words in the vulnerable names for each program. While for FFmpeg the least vulnerable names occur in 97 names, each is counter balanced by numerous benign occurrences. For example, frame occurs in 33 vulnerable names, but 293 benign ones.

6. Results

6.1. RQ1

Our first research question considers the results of being excessively conservative by predicting that all functions are vulnerable. This approach provides intuition for the meaningful range of F_2 scores. A perfect predictor attains the highest possible F_2 score of 1.0, while the worst-case F_2 score is 0.0. The F_2 score for a random predictor depends on the percentage of vulnerable functions. For example, the VDISC dataset, with its 72 612 vulnerable functions and 932 741 benign ones, is 7.2% vulnerable, meaning a random predictor will generate $TP =$

Table 3
Examples of the most and least dangerous words.

Asterisk	FFmpeg	LibPNG	LibTIFF	Pidgin	VLC
Most Dangerous					
invite	avi	png	JPEG	mxit	MP
retrans	264	handle	pdf	msn	AVI
pkt	vp	CCP	Checked	httpconn	Html
unpacksms	old	CAL	readwrite	slp	Strip
aocmessage	avcodec	PLT	LZWDecode	emoticon	Tags
milliwatt	ivi	do	Into	silc	ASF
astman	tile	PLTE	Entry	slplink	vcd
sipsock	mjpeg	read	Strips	yahoo	skcr
action	hdr	filter	cvt	idn	LOADSparse
ha	gif	chunks	readgitimage	untar	Recieve
Least Dangerous					
asn	ff	image	Samples	purple	Get
ast	get	transform	Handler	cb	vlc
254	write	store	Fax	get	Callback
PD	init	init	Error	pidgin	vlclua
PE	frame	standard	Check	set	Control
225	read	gpc	Image	jabber	Set
get	parse	16	Swab	add	Add
channel	mov	gp	Set	account	Out
handel	tag	gamma	Proc	blist	test
to	mxf	display	Warning	media	rtp

Table 4
Performance of the *all-vulnerable* predictor with performance of the LAVDNN model for comparison.

Dataset	Vulnerable	Benign	Empirical F_2 score	Theoretical F_2 score	Fold	LAVDNN F_2 score
Asterisk	9.8	2 020.4	0.024	0.024	mean[a]	0.013
FFmpeg	36.8	875.8	0.173	0.174	mean[a]	0.083
LibPNG	6.2	98.2	0.234	0.240	mean[a]	0.137
LibTIFF	15.0	104.4	0.410	0.418	mean[a]	0.292
Pidgin	5.2	1 344.4	0.019	0.019	mean[a]	0.019
VLC	7.4	539.8	0.064	0.064	mean[a]	0.089
loo	49.0	10 102.0	0.042	0.075	Asterisk	0.013
loo	184.0	4 379.0	0.274	0.075	FFmpeg	0.083
loo	31.0	491.0	0.362	0.075	LibPNG	0.137
loo	75.0	522.0	0.564	0.075	LibTIFF	0.292
loo	26.0	6 722.0	0.034	0.075	Pidgin	0.019
loo	37.0	2 699.0	0.110	0.075	VLC	0.089
VDISC	14 522.4	186 548.2	0.280	0.280	mean[a]	0.148

[a]For brevity, we present the mean values for 5-fold cross-validation. 10-fold cross-validation showed no statistically significant differences.

$FN = 3.6\%$ and $FP = TN = 46.4\%$, yielding an F_2 score of 0.228. When only 1.6% of the functions are vulnerable, the resulting F_2 score drops to 0.071.

Assuming that all functions are vulnerable, there are no false negatives or true negatives, and thus FN and TN are both zero. In part because F_2 favors recall, the *all-vulnerable* assumption will yield predicted F_2 scores slightly higher than random. For example, the VDISC dataset with its 7.2% vulnerable functions, results in an *all-vulnerable* predicted F_2 score of 0.280.

Conversely, given our imbalanced dataset, a high-accuracy strategy is to classify all functions as being in the dominant class. In our case, most functions are benign. Predicting that all functions are benign produces very high accuracies (e.g., 0.928 for the VDISC dataset). However, other than highlighting a potential misinterpretation, classifying all functions as benign makes little sense in the context of vulnerability prediction; thus, we consider it no further.

In support of RQ1, Table 4 shows the empirical and theoretical F_2 values for each dataset. These two scores provide some intuition for the performance of the simple *all-vulnerable* predictor. The empirical F_2 value for each fold is based on the number of vulnerable and benign functions in that fold. For the k-fold validations this leads to some minor variation that we summarize in the table using the mean. For the loo dataset the empirical values deviate from the theoretical values because the distribution of vulnerable names is far less uniform. Table 4 also shows performance of LAVDNN for comparison.

Thus, in summary for RQ1, Table 4 shows that the F_2 scores for this ultra-conservative approach range from 0.034 to 0.564 in line with the relative number of vulnerable and benign functions in each fold of each dataset. While its 100% recall may be the most redeeming quality, the *all-vulnerable* predictor sets a baseline for the minimum performance expected of a more sophisticated predictor. In RQ2 and RQ3, our goal is to provide an interpretability viewpoint for the LAVDNN model. We accomplish this by replacing all words being dangerous with more sophisticated techniques that we then compared to *all-vulnerable* and to each other.

6.2. RQ2

RQ2 investigates using $\mathcal{R}_L$ as the Rank function in Algorithm 1, that is, it studies how well LAVDNN can predict *dangerous words*. We begin our analysis with the two graphs shown in Fig. 1. The upper graph shows the minimum score on the x-axis and the F_2 score on the y-axis. Note that the x-axis is not to scale because values between 0.90 and 1.00 are more interesting. The line colors represent the three datasets within, loo, and VDISC. For comparison, the three horizontal lines show the F_2 scores from the ultra-conservative *all-vulnerable* predictor considered in RQ1 when applied to the three datasets.

The lower chart in Fig. 1 shows how increasing the required minimum score decreases the number of words considered dangerous (variable dangerous in Algorithm 1). With this decrease, the expectation is that the remaining, higher scoring, words will be better predictors, but apply to fewer vulnerable functions. This pattern is just evident in the very slight increase in the F_2 values for the loo and within datasets in the top chart. However the pattern is not strong and the trend for VDISC actually declines (reflecting the dataset being starved for useful vocabulary). However, the fact that the F_2 scores are relatively flat suggests that the reduction in the number of dangerous words is not providing greater discrimination, and thus that words given higher scores by $\mathcal{R}_L$ are not necessarily better predictors of vulnerable functions.

Objectively, for VDISC, $\mathcal{R}_L$ always underperforms the *all-vulnerable* predictor (top two lines in the top chart of Fig. 1). However, $\mathcal{R}_L$ outperforms the *all-vulnerable* predictor on the other two datasets. Omitting the minimum score of 1.00, where all three F_2 scores plummet to near zero, all three differences are statistically significant (t-test $p < 0.0001$ for VDISC and within, and $p = 0.0011$ for loo, using the data shown in Fig. 1).

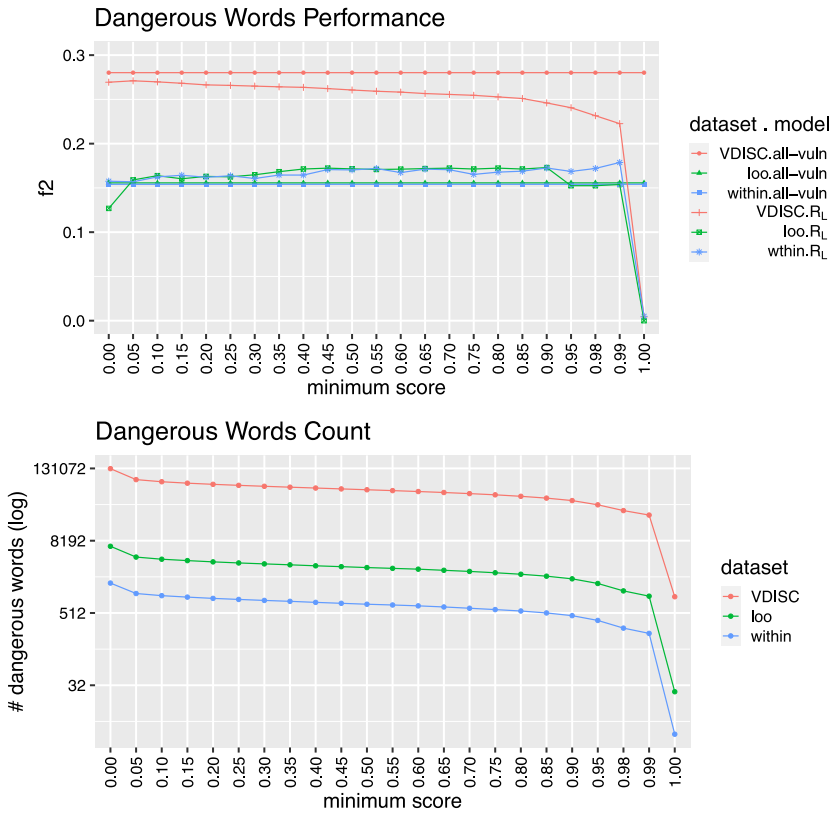


Fig. 1. Using $\mathcal{R}_L$ to select dangerous words.

In summary, LAVDNN finds limited success at identifying dangerous words. $\mathcal{R}_L$ works better with the smaller, more focused, datasets of within and loo. However, we note two caveats: first, the data clearly show that using too high a minimum score leads to too few dangerous words, which dramatically lowers the F_2 score, and second, it must be pointed out that on an absolute scale, the resulting F_2 scores are all on the low side.

Looking ahead to the comparison with $\mathcal{R}_F$, we note that numerically the best performance for VDISC is with a minimum score of 0.05, while for loo it is 0.90, and for within 0.99. These values, which are all evident in Fig. 1, reflect artifacts of the vocabularies. For example, for VDISC, finding the best performance with such a low minimum score indicates that the search is starved for high-quality vocabulary, while at the other end of the spectrum, for within, the very high minimum excludes all but the most suspect words.

6.3. RQ3

Our third research question explores replacing $\mathcal{R}_L$'s use of LAVDNN with $\mathcal{R}_F$'s frequency-based approach. If the performance of $\mathcal{R}_F$ is similar, it suggests that LAVDNN also captures notions related to word frequencies. Interestingly, if the alternative shows better performance, then it suggests the use of dangerous words to augment the training of next generation of predictors. While the range of alternatives is virtually limitless, because we are interested in the contribution of the terms, we consider a family of algorithms that use *term frequencies* to determine potentially dangerous words. For example, a straightforward algorithm would assert that all terms found in vulnerable function names in the training data are dangerous. More sophisticated approaches would consider as negative evidence terms (frequently) occurring in benign function names.

The family we consider increases a term's dangerousness when the term appears in function names from the vulnerable training data and decreases it when the term appears in names from the benign training data. Thus, the algorithm assigns higher scores to terms that have high frequency in the vulnerable training data and low frequency in the benign training data. We refer to this as a *family* because we consider $\mathcal{R}_F$ with a range of different *weights* (a pair of the amount added to, and the amount subtracted from, a term's score).

Algorithm 1's performance is impacted by the cutoff and threshold values returned by FindBest. This section first explores the cutoff/threshold search space. It then considers a range of fixed weights, and finally considers an algorithm that determines the best weight based on the training data.

6.3.1. Exploration of cutoff and threshold

The exploration starts by considering the collection of ROC graphs shown in Figs. 2–4. These graphs are for the optimal weights, which provide greater discrimination and thus make visual patterns easier to observe. A ROC graph plots the true-positive rate against the false-positive rate at various settings of a parameter (in our case threshold). They are useful for comparing classifiers with each other and with a "no skill" classifier. An ideal model produces a ROC curve that goes straight up and then straight over to the right, while the no-skill classifier produces a 45-degree line from the origin to the upper right.

When producing these graphs, we increment cutoff in steps of 100 to reduce visual clutter. The cutoff search landscape is reasonably smooth because, for example, in the step from 4100 to 4200, the first 4100 words are the same. Increments of 100 help speed up analysis and limit the size of the charts, while having no meaningful impact on their interpretation.

The graphs help us understand the interplay between cutoff and threshold. When threshold is 0.00, all function names are predicted to be vulnerable, and thus, performance degenerates to that of the *all-vulnerable* classifier. At this point, both the true-positive rate and the false-negative rate are 1.00 and the ROC curve ends at the upper right of the chart. With the exception of Fig. 2 we suppress this threshold

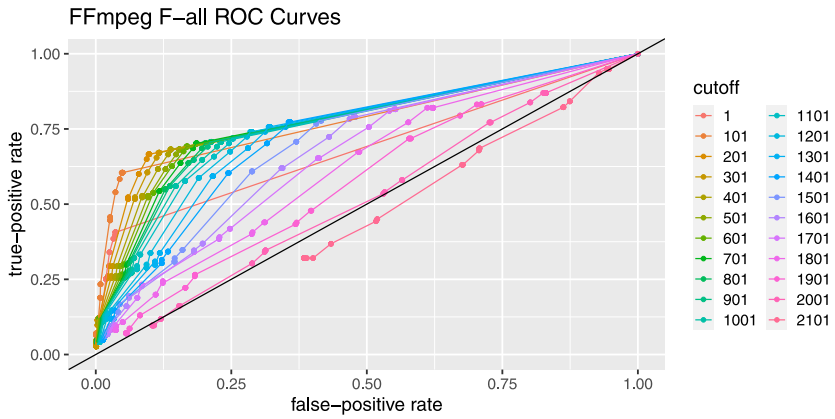


Fig. 2. ROC curves exploring the search space for cutoff and threshold.

because it causes considerable visual clutter (explaining the absence of lines to (1,1) in the other ROC charts).

To begin with, we consider the ROC graph for FFmpeg shown in Fig. 2 where each ROC curve shows threshold going from 1.00 to 0.00, while curve color shows cutoff going from small (red shift) to large (blue shift). Increasing cutoff can be seen to have two effects: first, it tends to flatten the curve, and second early on when cutoff goes from 1 to 101 the true-positive rate increases dramatically (from about 0.40 to about 0.65). Here the true-positive rate increases much more than the false-positive rate. However, by the time cutoff includes most of the words, the performance has degraded below that of a no-skill classifier. Therefore, in practice, a user might choose to increase cutoff until the false-positive rate reaches some tolerance (i.e., patience for wrong answers). Taken together, these two effects imply that the list of dangerous words has its most dangerous words first. Hence, we have our first interesting difference between $\mathcal{R}_F$ and $\mathcal{R}_L$, because $\mathcal{R}_L$ failed to effectively rank the dangerous words.

Fig. 3 illustrates minimum score's impact. These graphs show ROC curves for LibPNG, which is one of the datasets where $\mathcal{R}_F$ excels. Hence, the ROC curves are all closer to the ideal "up and over" curve. The ROC curves in these two charts show one of 10-sample cutoff's. The upper chart in Fig. 3 shows F-all, which uses no minimum score, and thus includes all words as potentially dangerous words. The lower chart shows F-0, where a minimum score of zero is used. F-0's restriction to only high-scoring words clearly causes the ROC curves to more closely resemble the ideal "up and over" curve than the flatter (inferior) ROC curves of F-all. Finally, the comparison again indicates that the high-scoring words are more likely to be associated with vulnerable functions.

Fig. 4 shows the VDISC dataset (the loo curves are similar). The curves reinforce the pattern seen above where larger values of cutoff degrade performance. Because this dataset has the least useful vocabulary all cutoff's except for the smallest, 1, and the largest, 130 001, show very similar overlapping performance. However, the patterns observed above are still evident. For example, performance improves when using more of the vocabulary up until the very end where, for the final cutoff value, performance dips below that of the no-skill classifier.

To summarize, there is a sweet-spot at rather small values of cutoff and threshold. Smaller cutoff values include only the highest scoring words: as seen in the ROC curves as cutoff increases there is a ubiquitous flattening of the curve. Having a limited number of dangerous words works best when combined with a low threshold because this combination requires only a few of a function name's terms to be on the dangerous words list.

6.3.2. Exploring the impact of weight

Initially, we explore the impact of a range of fixed weights, and later we apply an algorithm that dynamically determines the best weight based on the training data. Fig. 5 shows the performance of $\mathcal{R}_F$ for the three datasets within, loo, and VDISC. In this graph, the x-axis shows a range of *weights*. Each pair, plus--minus, shows the amount added to the score for each term in a vulnerable function name followed by the amount subtracted for each term in a benign function name. For reference, the graph also includes the performance of $\mathcal{R}_L$, which appears as a horizontal line because it is not affected by the weights.

Each line in the graph shows the performance of a specific ranking algorithm, which we refer to using a *tag* including three things: 'L' for $\mathcal{R}_L$ or 'F' for $\mathcal{R}_F$, the dataset involved: 'within ', 'loo', or 'VDISC', and the minimum score, where 'all' denotes that there is no minimum, and thus all words are included (in which case the ranking affects only the order of the words). For example, the tag F-within-0 is the top line, which applies $\mathcal{R}_F$ to the within dataset using a minimum score of 0.

Minimum score can help focus the analysis on high-scoring words. This effect is evident in the two lines for $\mathcal{R}_F$ applied to the within data set, F-within-0 and F-within-all. It can also limit the available vocabulary, which can hurt performance. This can be seen clearly in the case of VDISC where, on the left of the figure, F-VDISC-0 performs dramatically worse than F-VDISC-all, then, moving to the right as the weights favor inclusion, the performance gap disappears. Thus, enforcing a minimum can help focus the algorithm on high-scoring words, but it does so at the expense of limiting the available vocabulary.

Big picture, $\mathcal{R}_F$ performs best with the more focused vocabulary of the within dataset, which shows the impact of having the right vocabulary. In this case cross-validation *within each project* clearly provides relevant vocabulary. The top two curves also illustrate the impact of applying a minimum score to the list of dangerous words, which accounts for the gap between them.

For the loo dataset, $\mathcal{R}_F$ struggles to outperform $\mathcal{R}_L$. When including all words, the performance clearly degrades moving from left to right, which indicates that the list is loosing focus. The implication here is that it is more important to reduce the importance of terms from benign function names than to reinforce terms from vulnerable function names. For example, with a weight of 1–10, a word has to be very rare in the benign data to maintain a high score. When using a minimum score of 0, this pattern is eventually seen at the far right. However, at the far left the F_2 score suffers because the minimum limits the number of dangerous words to the detriment of the algorithm. This improves dramatically from 3-4 to 3-2 and levels off until it plummets at the far right where unwanted words do not get enough negative weight.

Finally, for the VDISC dataset $\mathcal{R}_F$ is rarely able to outperform $\mathcal{R}_L$. It only truly does so at the far right where the weight finally concentrates the truly dangerous vocabulary. Note that despite the positive slope on the right side of the graph, running the weights out to 1000-1 shows no further improvement. The left of the graph parallels that of the loo dataset with F-VDISC-0 being starved for vocabulary. One interesting feature of F-VDISC-all is that to the left of 3-2, the training data leads

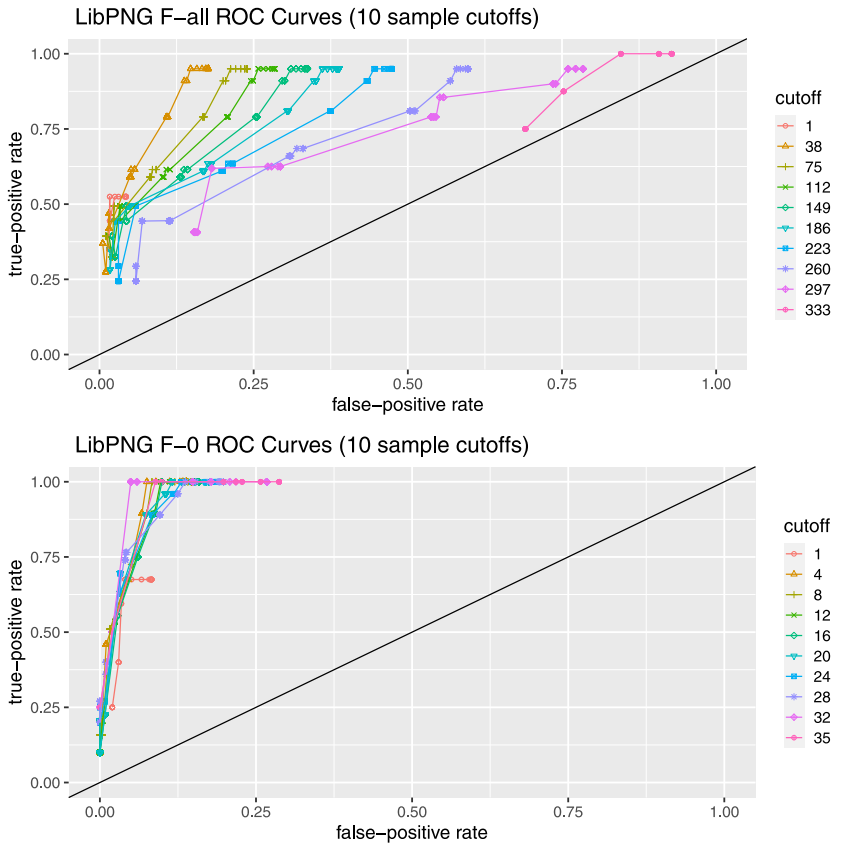


Fig. 3. Impact of minimum score.

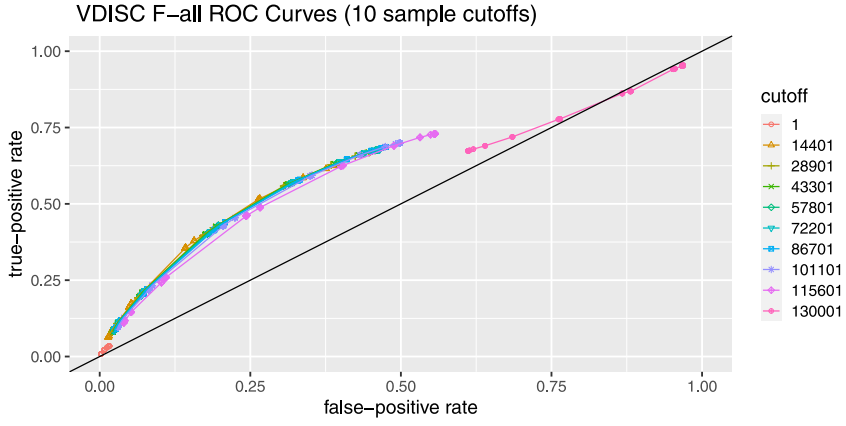


Fig. 4. ROC curves for VDISC.

the algorithm to include all of the terms as potential dangerous words. Here the performance is similar to L-VDISC-0.05, where the minimum score of 0.05 includes all but the lowest-scoring words.

For the F-VDISC-all data, it is interesting that the weights 1-10 and 10-1 have similar performance (one might see 1-10 as all the uninteresting words "taking a step back" while 10-1 as all the interesting words "taking a step forward"). More formally, on the left for F-VDISC-0 only words that are absent from the benign list are included because of the large minus value in the weights. Using V for vulnerable and B for benign this is the set $V - B$. At the far right the influence of B is negligible and the resulting set is effectively V. Digging deeper and comparing the values of cutoff used, for 1-10 each fold uses all of the 130 thousand unique words available. In contrast, for 10-1, only 14% are used. Thus, the important vocabulary is effectively concentrated by the weight 10-1 better than the weight 1-10. That this concentration does not give notably better performance is an indication of the lack of useful vocabulary in the VDISC dataset. Therefore, we can percolate truly dangerous words to the beginning of the list, but lack a sufficient number of them to improve the F_2 score.

To objectively consider the vocabulary patterns, we statistically compare the models used to construct Fig. 5. We also consider the impact of vocabulary size. Table 5 summarizes the results of Tukey's Honestly Significant Difference (HSD) applied to each dataset. This test identifies specific treatment means that differ from each other (those that do not share a letter). For the within dataset, $\mathcal{R}_F$ is clearly the more successful. Of particular interest is that for F-within-0, an average of only 49 words are deemed dangerous, which is a mere 3% of the 1613 words used by F-within-all. This is the frequency algorithm at its best. The same pattern is seen with the loo dataset

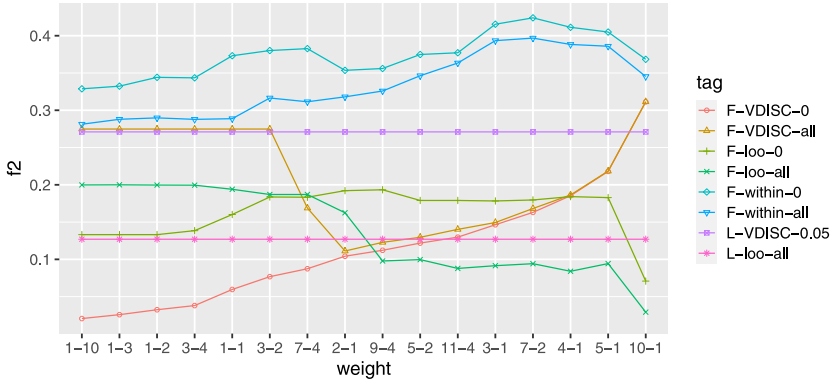


Fig. 5. Comparison of $\mathcal{R}_F$ performance with fixed weights.

Table 5
Tukey's HSD for performance.

model-dataset-filter	F_2	HSD group	DWC average[a]
F-within-0	0.372	a	49
F-within-all	0.333	a	1 613
L-within-0.99	0.179	b	234
L-within-0.90	0.172	b	462
L-within-0.05	0.157	b	1 081
		($p < 0.0001$)	
L-loo-0.90	0.173	a	1 898
F-loo-0	0.163	a	202
L-loo-0.05	0.159	a	4 374
L-loo-0.99	0.154	a	976
F-loo-all	0.138	a	6 618
		($p = 0.9446$)	
L-VDISC-0.05	0.271	a	85 383
L-VDISC-0.90	0.246	a	38 199
L-VDISC-0.99	0.223	a	21 905
F-VDISC-all	0.218	a	129 966
F-VDISC-0	0.115	b	10 782
		($p < 0.0001$)	

[a]DWC = Dangerous Word Count.

Table 6
Comparing $\mathcal{R}_F$ with the maximum $\mathcal{R}_L$ performance (**bold** shows statistically significant improvement).

	Dataset	$\mathcal{R}_F$ F_2	max($\mathcal{R}_L$ F_2)	p-value	limit($\mathcal{R}_F$ F_2)
all words (F-all)	overall	**0.355**	0.182	<0.0001	0.442
	Asterisk	0.000	**0.038**	0.0264	0.060
	FFmpeg	**0.375**	0.230	<0.0001	0.386
	LibPNG	**0.651**	0.275	0.0003	0.850
	LibTIFF	0.434	0.450	0.7046	0.542
	Pidgin	**0.629**	0.011	<0.0001	0.667
	VLC	**0.247**	0.082	0.0111	0.410
	loo	0.196	0.173	0.9767	0.314
	VDISC	**0.311**	0.271	<0.0001	0.330
minimum score 0 (F-0)	overall	**0.345**	0.182	<0.0001	0.416
	Asterisk	0.000	**0.038**	0.0264	0.000
	FFmpeg	**0.361**	0.230	<0.0001	0.381
	LibPNG	**0.639**	0.275	0.0004	0.868
	LibTIFF	0.434	0.450	0.7051	0.503
	Pidgin	**0.601**	0.011	<0.0001	0.678
	VLC	**0.247**	0.082	0.0419	0.388
	loo	0.193	0.173	0.9814	0.221
	VDISC	**0.311**	0.271	<0.0001	0.330

where F-loo-0 selects only 3% of the dangerous words used by F-loo-all (202 versus 6618). However, none of the numeric differences for the loo dataset are statistically significant. Still, it is interesting that F-loo-0 has better numeric performance while using only 3% of the dangerous words. Finally, for VDISC, the opposite is true. For example, comparing F-VDISC-all and F-VDISC-0, the use of a larger vocabulary is accompanied by better performance. The data in Table 5 reinforces the general pattern where $\mathcal{R}_F$ shines when given a more focused vocabulary.

Finally, in the table, higher standard deviation is the reason that no statistically significant differences are seen with the loo dataset despite some of the numeric differences in the F_2 scores being on par with those attained using the VDISC dataset. This is not unexpected as the VDISC dataset is a large uniform dataset where, as shown in Table 1, the projects of loo have very different characteristics. Summarizing the data shown in Table 5: with the right well-focused vocabulary, $\mathcal{R}_F$ performs quite well indicating that words valuable to the prediction exist. The challenge can be finding this vocabulary.

6.3.3. Determining the best weights using training data

In production, $\mathcal{R}_F$ determines the best weight based on the training data. Our current algorithm uses a simple brute-force search through the list of weights used to create in Fig. 5, and selects the weight providing the best performance. The result is shown in Fig. 6, which compares $\mathcal{R}_F$ using all words and a minimum score of zero with the best performing minimum scores for $\mathcal{R}_L$ for the within, loo, and VDISC datasets (0.99, 0.90, and 0.05, respectively). The x-axis shows the dataset while the y-axis shows the F_2 score. Note that in the following, we look at the individual datasets of within separately.

From an interpretability viewpoint, there is clear evidence in Fig. 6 that LAVDNN is not exploiting terms to the extent possible. Specifically, Pidgin, and to a lesser extent LibPNG, showcase $\mathcal{R}_F$'s advantage over $\mathcal{R}_L$ at exploiting a focused vocabulary. While less pronounced, the same is true of FFmpeg and VLC. Of the remaining two projects from the within dataset Asterisk proves universally hard to predict while LibTIFF is comparatively easy to predicted for both $\mathcal{R}_L$ and $\mathcal{R}_F$.

Table 6 shows the results of ANOVAs separately comparing the two $\mathcal{R}_F$ models (F-all and F-0) with the three top-performing $\mathcal{R}_L$ models. $\mathcal{R}_F$'s performance shows that terms have unexploited value in six of the nine datasets where $\mathcal{R}_F$ performs better than $\mathcal{R}_L$. Furthermore, its performance is inferior on only one (where the p-value of 0.0264 is not a strong endorsement). These results reinforce the general pattern seen in the previous analysis, where the frequency models excel when using smaller and less diverse vocabularies. The only within dataset that $\mathcal{R}_F$ truly struggles with is Asterisk, which, as seen in Table 1, has the largest vocabulary and lowest percentage of vulnerable functions.

While it is easy to get drawn into the relative comparison of the F_2 scores, we are also interested in the relative contribution that terms make to the discrimination of vulnerable functions. To provide an indication of how much room there is for improvement, the last column of Table 6 shows the best F_2 score that $\mathcal{R}_F$ attains on the training data. While not a hard limit on its performance with the test data, typically, for the given list of dangerous words, the F_2 score on the training data provides an upper bound for a value seen using the test data. Thus, this column provides an indication of the best that one might expect to

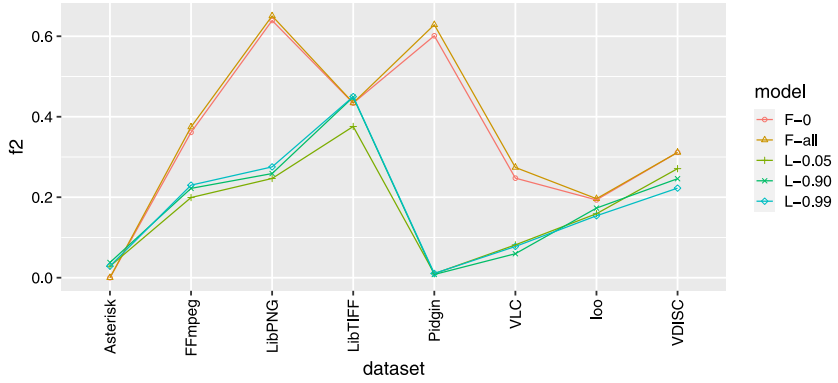


Fig. 6. Dataset comparison (formally a bar graph is appropriate, but lines make the values visually easier to compare).

Table 7
Comparison of dangerous words counts.

Dataset	F_2 score		dangerous words count		
	min 0	all	min 0	all	percent
Asterisk	0.000	0.000	22	3184	0.70%
FFmpeg	0.361	0.375	126	2111	5.96%
LibPNG	0.639	0.651	31	327	9.42%
LibTIFF	0.434	0.434	61	434	14.11%
Pidgin	0.601	0.629	28	1886	1.47%
VLC	0.247	0.274	38	1738	2.16%
loo	0.193	0.196	213	6618	3.22%
VDISC	0.311	0.311	18478	129966	14.22%

attain using only the terms found in the function names. While some projects such as LibPNG and LibTIFF show room for improvement, projects such as Pidgin and FFmpeg are within 0.038 and 0.011 of their maximum F_2 score.

Finally, Table 7 shows the number of dangerous words used by $\mathcal{R}_F$ for each dataset found on the x-axis of Fig. 6. What is quite striking in the table is how well the small vocabularies perform when using a minimum score of zero (e.g., with LibPNG and LibTIFF). While the smaller vocabularies never produce a numerically higher F_2 score, *none* of the differences is statistical significant (the smallest p-value is 0.729).

6.3.4. Summary

Returning to RQ3's "Does direct construction of the dangerous words list provide insight into LAVDNN?" The answer is a resounding "yes." As seen in Table 6, $\mathcal{R}_F$ outperforms $\mathcal{R}_L$ for most datasets. While on an absolute scale the F_2 values are not high, the important take home message here is that there is utility in the terms that LAVDNN is failing to exploit. Hence, the training of future Deep Neural Networks aimed at vulnerability prediction should include features based on the terms in the hope of better exploiting their potential. Furthermore, the relative performance on within where the vocabulary is more focused, suggests two additional things. First, considered in the context of an evolving system where past data from the same system can be used in the prediction, $\mathcal{R}_F$ alone provides a lightweight pre-filter to activities such as manual code review. The second interesting implication comes from $\mathcal{R}_F$ attaining respectable performance on some of the datasets (i.e., Pidgin's F_2 score of 0.651 and LibPNG of 0.629). Specifically, these results are achieved using vanishingly little training data from the neural net training perspective (e.g., only 26 and 31 vulnerable function names exist in the data for Pidgin and LibPNG, respectively).

6.4. Threats to validity

We identify potential threats to the validity of our experimental design and evaluation. One such threat is that the source of our VDISC dataset is a collection of functions extracted from open-source projects [25] that are unknown to us, both in scope, demographics, and domain. This may have resulted in an unknown bias effect regarding our results for VDISC. Furthermore, we evaluate on only six project-based datasets. It would mitigate this threat to external validity if the number of such datasets was increased. We continue to look for additional datasets to use in our experiments. The external validity of our diversity observations could also be improved by access to more data with a known heterogeneity.

Moreover, the greedy search used by FindBest trades precision (finding a global maximum) for speed. Thus, we prioritize performance, but internal validity could improve using techniques with a lower chance of getting stuck in local maxima. While this search is an approximation because local maxima exist, exhaustive searches using the smaller datasets found the error was only a few percent.

A further threat arises from the tools used in our study. There may be defects in the implementation that escaped our testing, thereby affecting our results. The same is true of LAVDNN, which was provided by its authors.

Finally, the statistical tests used are all well established and their implementations publicly available in R, and thus well vetted. However, it is possible that more appropriate tests unknown to us might provide more appropriate evidence. We also endeavored to follow the most up-to-date information from the statistics community when interpreting the models [26].

7. Related work

A source code vulnerability is a weakness in the source code that can be exploited into a security issue. Publicly known vulnerabilities are organized by common identifiers in the Common Vulnerabilities and Exposures (CVE) database [23], where they are classified using the Common Weakness Enumeration (CWE) [27], and ranked using the Common Vulnerability Scoring System (CVSS) [28].

Over the last two decades, various methods have been presented to identify potential security vulnerabilities in code based on static program analysis [3,4]. The recent advances and successes in machine learning (ML) have resulted in an increased interest in adapting these techniques to the vulnerability prediction problem [29–33]. However, the choice of feature types, classifiers, and data balancing techniques has a large impact on the prediction's performance [34]

The *naturalness hypothesis* [35] states that source code exhibits similar statistical properties as other forms of human communication. This means that corpus-based statistical learning can capture the local regularity in source code, i.e., such models can predict with high accuracy what code to expect in a given context, or what properties such code should have. One application of this idea is the use of deep feature representation learning on lexed C and C++ source code for automatic function-level vulnerability detection [25].

Recurrent Neural Networks (RNN) and Long Short Term Memory networks (LSTM) have been successfully applied for code reviews and vulnerability detection [36–38], for example by training a Bidirectional LSTM on so-called code gadgets, which are collections of semantically related lines [19]. Semantic properties of code can be predicted using code2vec [39], which represents code snippets as a fixed-length code vector, very similar to how word2vec [40] represents sentences. Harer et al. find that ML-based vulnerability prediction trained directly on C and C++ source code performs better than alternative approaches that were trained using semantic (build-time) information for the same source code (such as control flow information and def-use relations) [41].

One of the main challenges to all of these approaches is that they are computationally expensive to develop, as well as to keep up to date with newly discovered vulnerability patterns. The LAVDNN research [12] that inspired this paper is an example of a more lightweight approach with more modest goals. By using function names as semantic cues for training a Deep Neural Network (DNN), the model aims to predict potential vulnerability of a function based on its name, with the goal of helping a developer (or analysis technique) focus on those functions that should be scrutinized more carefully.

8. Concluding remarks

LAVDNN [12] attempts lightweight function vulnerability prediction based solely on the function's name. This paper takes that idea a step further and explores featherweight prediction based solely on the terms that make up function names. In doing so, this paper aims to provide an interpretability viewpoint for better understanding of an otherwise "black box" DNN. We find that LAVDNN has limited ability to identify dangerous words, and generally cannot provide an effective ranking of those words.

Comparing the relative performance of FAVD_L and FAVD_F allows us to probe whether LAVDNN learns "dangerous words" or attempts classification in other ways. Generally, FAVD_L provides much more consistent (though not very good) performance. When faced with a diverse dataset such as VDISC, the performance is generally better than FAVD_F. However, with the more focused vocabulary of a single system, FAVD_F dramatically outperforms FAVD_L. From this, we can conclude that LAVDNN has some ability to identify dangerous words, but does *not* generally provide an effective ranking of those words.

The performance difference suggests that there is room to augment DNN's with networks that directly learn dangerous words, particularly in more mature projects with more stable vocabularies. It is also worth noting that there are many cases in which FAVD_F outperforms the publicly available implementation of LAVDNN. This improvement suggests two things. First, augmentation with a dangerous words predictor can improve predictions if appropriate context is detected. The second implication is that for a mature project, FAVD_F might replace LAVDNN as an even lighter weight predictor. In this case, FAVD_F has the advantage that it needs orders of magnitude less training data.

Future work includes qualitative analysis that uses the ground truth to assess to what extent the vulnerability predictions by the two approaches overlap, are in conflict, or are complementary. When LAVDNN is to be augmented by other word finding algorithms, it is critical to understand what words LAVDNN already finds. Another possibility is to try to determine if the DNN picks up on patterns involving various n-grams (e.g., 3-grams composed of three consecutive letters from an identifier) and also on non-adjacent letter patterns in the identifiers. With respect to the technique itself, one area for future work is to consider alternative sources of dangerous words. These need not all be code-based. For example, issues noted during requirements solicitation might provide a source of additional dangerous words.

As a possible LAVDNN augmentation, we plan to investigate heavier-weight vocabulary possibilities. For example, we plan to train a similar model to LAVDNN on our own data, and investigate the value of splitting identifiers during the DNN model training phase. We also plan to consider sources of vocabulary beyond function names to improve the prediction, such as, formal parameters, called functions, and alike.

CRediT authorship contribution statement

David Binkley: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration. **Leon Moonen:** Conceptualization, Methodology, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Visualization, Supervision, Project administration, Funding acquisition. **Sibren Isaacman:** Methodology, Validation, Formal analysis, Investigation, Resources, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

Dr. Moonen's work is supported by the Research Council of Norway through the secureIT project (RCN contract #288787).

References

[1] P. Anderson, T. Reps, T. Teitelbaum, M. Zarins, Tool support for fine-grained software inspection, IEEE Softw. 20 (4) (2003) 42–50.
[2] A. Bessey, D. Engler, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, A few billion lines of code later, Commun. ACM 53 (2) (2010) 66–75.
[3] M. Pistoia, S. Chandra, S.J. Fink, E. Yahav, A survey of static analysis methods for identifying security vulnerabilities in software systems, IBM Syst. J. 46 (2) (2007) 265–288.
[4] M. Kulenovic, D. Donko, A survey of static code analysis methods for security vulnerabilities detection, in: International Convention On Information And Communication Technology, Electronics And Microelectronics, MIPRO, 2014, pp. 1381–1386.
[5] P. Anderson, 90% Perspiration: engineering static analysis techniques for industrial applications, in: International Working Conference On Source Code Analysis And Manipulation, SCAM, 2008, pp. 3–12.
[6] A. Austin, L. Williams, One technique is not enough: a comparison of vulnerability discovery techniques, in: International Symposium On Empirical Software Engineering And Measurement, IEEE, 2011, pp. 97–106.
[7] D. Baca, B. Carlsson, K. Petersen, L. Lundberg, Improving software security with static automated code analysis in an industry setting, Softw. Pract. Exp. 43 (3) (2013) 259–279.
[8] M. Fagan, Design and code inspections to reduce errors in program development, IBM Syst. J. 15 (3) (1976) 182–211.
[9] B. Freimut, L.C. Briand, F. Vollei, Determining inspection cost-effectiveness by combining project data and expert opinion, IEEE Trans. Softw. Eng. 31 (12) (2005) 1074–1092.
[10] C. Wohlin, A. Aurum, H. Petersson, F. Shull, M. Ciolkowski, Software inspection benchmarking - a qualitative and quantitative comparative opportunity, in: International Software Metrics Symposium, METRICS 2002, IEEE, 2002.
[11] A.A. Porter, H. Siy, A. Mockus, L.G. Votta, Understanding the sources of variation in software inspections, ACM Trans. Softw. Eng. Meth. 7 (1) (1998) 41–79.
[12] R. Li, C. Feng, X. Zhang, C. Tang, A lightweight assisted vulnerability discovery method using deep neural networks, IEEE Access 7 (2019) 80079–80092.
[13] J. Nam, W. Fu, S. Kim, T. Menzies, L. Tan, Heterogeneous defect prediction, IEEE Trans. Softw. Eng. 44 (9) (2018) 874–896.
[14] X.-Y. Jing, F. Wu, X. Dong, B. Xu, An improved SDA based defect prediction framework for both within-project and cross-project class-imbalance problems, IEEE Trans. Softw. Eng. 43 (4) (2017) 321–339.
[15] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, in: International Conference On Software Engineering, ICSE, 2016, pp. 297–308.
[16] G. Lin, S. Wen, Q.-L. Han, J. Zhang, Y. Xiang, Software vulnerability detection using deep neural networks: a survey, Proc. IEEE (2020) 1–24.
[17] K.Z. Sultana, V. Anu, T.-Y. Chong, Using software metrics for predicting vulnerable classes and methods in java projects: A machine learning approach, 33 (3) (2020) e2303.
[18] H.K. Dam, T. Tran, T.T.M. Pham, S.W. Ng, J. Grundy, A. Ghose, Automatic feature learning for predicting Vulnerable software components, IEEE Trans. Softw. Eng. 47 (1) (2018) 67–85.

[19] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, VulDeePecker: a deep learning-based system for vulnerability detection, in: Network And Distributed System Security Symposium, 2018, arXiv:1801.01681.
[20] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, P. Montague, Cross-project transfer representation learning for vulnerable function discovery, IEEE Trans. Ind. Inform. 14 (7) (2018) 3289–3297.
[21] C. Caprile, P. Tonella, Nomen est omen: Analyzing the language of function identifiers, in: Working Conference On Reverse Engineering, WCRE, 1999, pp. 112–122.
[22] E. Hill, D. Binkley, D. Lawrie, L. Pollock, K. Vijay-Shanker, An empirical study of identifier splitting techniques, Emp. Softw. Eng. 19 (6) (2014) 1754–1780.
[23] MITRE, Common Vulnerabilities and Exposures (CVE), https://cve.mitre.org/.
[24] SYNOPSYS, Open Hub, the Open Source Network, https://www.openhub.net/.
[25] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, M. McConley, Automated vulnerability detection in source code using deep representation learning, in: International Conference On Machine Learning And Applications, ICMLA, IEEE, 2018, pp. 757–762.
[26] R.L. Wasserstein, A.L. Schirm, N.A. Lazar, Moving to a world beyond "p < 0.05", Am. Stat. 73 (sup1) (2019) 1–19.
[27] MITRE, Common Weakness Enumeration (CWE), https://cwe.mitre.org/.
[28] Forum of Incident Response and Security Teams (FIRST), Common Vulnerability Scoring System (CVSS), https://www.first.org/cvss.
[29] Y. Pang, X. Xue, H. Wang, Predicting vulnerable software components through deep neural network, in: International Conference On Deep Learning Technologies, ICDLT, ACM, 2017, pp. 6–10.
[30] S.M. Ghaffarian, H.R. Shahriari, Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey, ACM Comput. Surv. 50 (4) (2017) 1–36.
[31] A. Handa, A. Sharma, S.K. Shukla, Machine learning in cybersecurity : A review, Data Min. Knowl. Discov. 9 (4) (2019) e1306.
[32] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, H. Jin, A comparative study of deep learning-based vulnerability detection system, IEEE Access 7 (2019) 103184–103197.
[33] J. Jiang, X. Yu, Y. Sun, H. Zeng, A survey of the software vulnerability discovery using machine learning techniques, in: Artificial Intelligence And Security, in: Lecture Notes in Computer Science, 11635, Springer, 2019, pp. 308–317.
[34] A. Kaya, A.S. Keceli, C. Catal, B. Tekinerdogan, The impact of feature types, classifiers, and data balancing techniques on software vulnerability prediction models, J. Softw. Evol. Process 31 (9) (2019) e2164.
[35] M. Allamanis, E.T. Barr, P. Devanbu, C. Sutton, A survey of machine learning for big code and naturalness, ACM Comput. Surv. 51 (4) (2018) 81:1–81:37.
[36] A. Gupta, N. Sundaresan, Intelligent code reviews using deep learning, in: KDD Deep Learning Day, 2018, p. 9.
[37] G. Fan, X. Diao, H. Yu, K. Yang, L. Chen, Software defect prediction via attention-based recurrent neural network, Sci. Program. (2019).
[38] A. Xu, T. Dai, H. Chen, Z. Ming, W. Li, Vulnerability detection for source code using contextual LSTM, in: International Conference On Systems And Informatics, ICSAI, 2019, pp. 1225–1230.
[39] U. Alon, M. Zilberstein, O. Levy, E. Yahav, Code2vec: Learning distributed representations of code, in: Principles Of Programming Languages, POPL, ACM, 2019, pp. 1–29.
[40] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: International Conference On Neural Information Processing Systems, NIPS, Curran Associates Inc. 2013, pp. 3111–3119.
[41] J.A. Harer, L.Y. Kim, R.L. Russell, O. Ozdemir, L.R. Kosta, A. Rangamani, L.H. Hamilton, G.I. Centeno, J.R. Key, P.M. Ellingwood, E. Antelman, A. Mackay, M.W. McConley, J.M. Opper, P. Chin, T. Lazovich, Automated Software Vulnerability Detection with Machine Learning, Tech. rep., 2018, CoRR e-print, arXiv:1803.04497.