Multiclass Classification of Software Vulnerabilities with Deep Learning

Crystal Contreras[*]
Amazon.com, Inc.
East Palo Alto, California, USA
cryacon@amazon.com

Hristina Dokic
DePaul University
Chicago, Illinois, USA
hdjokic@depaul.edu

Zhen Huang
DePaul University
Chicago, Illinois, USA
zhen.huang@depaul.edu

Daniela Stan Raicu
DePaul University
Chicago, Illinois, USA
draicu@cdm.depaul.edu

Jacob Furst
DePaul University
Chicago, Illinois, USA
jfurst@cdm.depaul.edu

Roselyne Tchoua
DePaul University
Chicago, Illinois, USA
rtchoua@depaul.edu

ABSTRACT

Detecting software vulnerabilities has been a challenge for decades. Many techniques have been developed to detect vulnerabilities by reporting whether a vulnerability exists in the code of software. But few of them have the capability to categorize the types of detected vulnerabilities, which is crucial for human developers or other tools to analyze and address vulnerabilities. In this paper, we present our work on identifying the types of vulnerabilities using deep learning. Our data consists of code slices parsed in a manner that captures the syntax and semantics of a vulnerability, sourced from prior work. We train deep neural networks on these features to perform multiclass classification of software vulnerabilities in the dataset. Our experiments show that our models can effectively identify the vulnerability classes of the vulnerable functions in our dataset.

KEYWORDS

Vulnerability classification, software and application security, machine learning, deep learning, neural networks

ACM Reference Format:
Crystal Contreras, Hristina Dokic, Zhen Huang, Daniela Stan Raicu, Jacob Furst, and Roselyne Tchoua. 2023. Multiclass Classification of Software Vulnerabilities with Deep Learning. In *2023 15th International Conference on Machine Learning and Computing (ICMLC 2023), February 17–20, 2023, Zhuhai, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3587716.3587738

1 INTRODUCTION

Software security is severely impacted by vulnerabilities, which are frequently exploited by attackers for various malicious purposes such as compromising computers systems, gaining monetary benefits, or causing damages. Over the years, many real-world attacks took advantage of vulnerabilities. Some prominent incidents include the breach of private data of 533 million Facebook users [1] and the Lockfile ransomware [26] in 2021, Russian's attack on U.S Federal agencies in 2020 [35], and the WannaCry ransomware [2] in 2017.

Despite decades' effort to address vulnerabilities [7, 8, 13, 15–17, 20, 31, 36–39], they are still ubiquitous. From 2019 to 2021, over 17,000 software vulnerabilities have been reported publicly each year. The prevalence of vulnerabilities has motivated the development of many new techniques to automatically detect vulnerabilities. As learning-based approaches have achieved success in many areas of software security and reliability [9, 11, 14, 22, 27–29, 34, 36, 40], they have also demonstrated promising results in detecting vulnerabilities [4, 5, 12, 23, 25, 41].

The vast majority of these techniques aim to detect the existence of vulnerabilities. Unfortunately few of them take a step further to identify the type of detected vulnerabilities, which is crucial for developers to quickly understand the nature of the detected vulnerabilities and develop corresponding strategies to repair them. This kind of information can considerably save the time and effort of developers in fixing vulnerabilities and thus significantly reduce the pre-patch window, the time between the discovery of a vulnerability, and the release of the patch for the vulnerability.

In this paper, we present our approach on automatically categorizing the types of vulnerabilities using deep learning models. The main challenge of our approach boils down to choosing the taxonomy for vulnerability categorization, identifying the features that capture the intrinsic characteristics of different types of vulnerabilities, and determining which deep learning algorithms to use.

First, we need to choose which taxonomy of vulnerability types should be used. There exist various standards for categorizing vulnerabilities. For example, they can be categorized as memory-safety vulnerabilities, authentication vulnerabilities, web security vulnerabilities, etc., based on the subsystems in which the vulnerability exists. The memory-safety vulnerabilities can be further categorized as buffer overflows, double free, use-after-free, and so on. From the perspective of the developers who are fixing vulnerabilities, finer granularity of categorization would be more useful.

Second, we should identify appropriate features that can be used to distinguish different vulnerability types based on our choice of taxonomy. Using a dataset from prior work that has been used to

[*]This work was done when the author was a graduate student at DePaul University.

ICMLC 2023, February 17–20, 2023, Zhuhai, China

ACM ISBN 978-1-4503-9841-1/23/02…$15.00
https://doi.org/10.1145/3587716.3587738

categorize vulnerabilities, i.e. performing multiclass classification, may prove to be useful in this regard. When using prior work that focuses on detecting vulnerabilities, i.e. performing binary categorization of vulnerabilities, it is unclear whether the features used by prior work will meet the needs for our purpose.

Lastly, many different deep neural networks exist. Some deep neural networks have been used in prior work for detecting or categorizing vulnerabilities and successfully achieved high accuracy. We need to find out which of them works best for categorizing vulnerabilities.

In order to offer fine-grained categorization, our approach uses the taxonomy of Common Weakness Enumeration (CWE) for software development, which uses a well-defined hierarchy of software weaknesses, i.e. defects, and labels each type of software weakness with a unique CWE ID. We find that the data we use from prior work are rooted from the National Vulnerability Database (NVD) and Software Assurance Reference Dataset (SARD), which come with manually labelled CWE IDs.

Many studies have shown that the features based on the control flow and data flow information on program source code can be used to detect vulnerabilities accurately [24, 25, 41]. Particularly μVulDeePecker is the pioneering work in categorizing vulnerabilities using deep learning, but it only uses the features relevant to syntax characteristics of type API/library function calls and limits the categorization to 40 vulnerability classes. Also, their approach has only been tested using the Recurrent Neural Network (RNN) Bidirectional Long Short-Term Memory (BLSTM).

To extend the work on multiclass classification of vulnerabilities, we use a dataset that consists of four different types of vulnerable syntax characteristics: API/library function calls (API), array usage (AU), pointer usage (PTR), and arithmetic expression (AE). This dataset contains 11 classes found in μVulDeePecker's dataset plus 39 new classes, giving us a total of 50 vulnerability classes. We train two distinct deep learning models, one that uses BLSTM as the deep neural network and another that uses Bidirectional Gated Recurrent Unit (BGRU), on this data and compare their effectiveness at classifying vulnerabilities. Our evaluation focuses on achieving the highest F1-measure to determine the overall effectiveness of our models' ability to classify vulnerabilities.

In summary, this paper makes the following major contributions:

- We develop two different deep learning models in a simple architecture to classify vulnerabilities.
- We use a dataset that contains 50 classes of vulnerabilities covering four different types of syntax characteristics.
- We have made our implementation publicly available at https://gitlab.com/vulnerability_analysis/vulnerability_classification.

The remainder of this paper is organized as follows. We discuss the background of our work and related work in Section 2. We present the design of our approach in Section 3. Section 4 shows the results of our experiments. Finally, we summarize our conclusions in Section 5.

2 RELATED WORK

Over the years many techniques have been proposed to use machine learning or deep learning models to detect vulnerabilities in C/C++ programs [4, 6, 11, 23, 24, 30, 32]. Some techniques extract features from the dynamic behavior of programs, such as the execution traces [11]. Other techniques extract features from the static program code [23, 24]. In this section, we focus on the techniques that extract features from the source code of C/C++ programs.

2.1 Embedding Models

To transform program code into numeric vectors that can be used by machine learning and deep learning models, various embedding models, such as Word2Vec and Doc2Vec, are adopted for vulnerability detection. Jeon and Kim compared five embedding models, Word2Vec, Sent2Vec, Doc2Vec, Glove, and FastText, in their effectiveness at representing C/C++ source code as vectors for deep learning models to detect vulnerabilities [18]. Their results show that FastText and Word2Vec have the best performance. Zhen Li, et al. also find that Word2Vec yields significantly better results than the classic bag-of-words method [24, 42].

Code2Vec is a recent embedding model that is specifically designed to transform code into vectors [3]. It is originally used to predict method names from the code of methods and achieves decent accuracy. Although it may have the potential for other applications, such as vulnerability detection, it has two major limitations. First, Code2Vec has a high dependency on variable names. Its accuracy degrades when obfuscated or adversarial variable names are used. Second, Code2Vec is designed to work with Java and C# programs. In order to use it with C/C++, a new extractor needs to be implemented. Bilgin et al. use astminer, an extractor for the C language[21], with Code2Vec to detect vulnerabilities in C programs [6]. Their evaluation shows that Code2Vec does not perform very well on highly imbalanced datasets.

2.2 Syntax Characteristics

Vulnerability detection techniques usually learn features from the program code mostly relevant to vulnerabilities. Recent work has shown promising results by focusing on program slices that contain likely vulnerable code. The program slices are often obtained by following the dependency to the code that fits certain syntax characteristics, such as calling API functions.

SySeVR [24] distinguish four different types of syntax characteristics: calling API functions, accessing arrays, using pointers, and arithmetic operations. It applies machine learning on these syntax characteristics individually and achieves different accuracy for each of them. μVulDeePecker [41] focuses on only one type of syntax characteristics: calling API functions.

Different from prior work, our work combines program slices obtained using all the four different types of syntax characteristics and extracts features from these program slices.

2.3 Multiclass Vulnerability Classification

The vast majority of prior work detects the existence of vulnerabilities. Few work classifies the types of vulnerabilities.

μVulDeePecker classifies vulnerabilities into multiple classes. It uses three BLSTM networks to learn global features, local features, and to fuse features, respectively. The global features capture both data dependency and data dependency among multiple statements in program code, while the local features focus on information within one statement, such as function call arguments.

μVulDeePecker uses a dataset of 33,409 programs and labels 40 classes of vulnerabilities.

Comparing to μVulDeePecker, our work has a simpler design and classifies more vulnerability classes. It uses only one BLSTM network. Our dataset has 54,049 programs and contains 50 classes of vulnerabilities.

3 DESIGN

3.1 Vector Representation of Samples

Judging from previous studies [23, 24, 30, 32, 41], implementing a methodical code structure representation of the data is seen to enhance a model's performance during feature learning. Since deep neural networks expect vectors as inputs, we convert our samples into vector representations while still retaining as much of the context required to notice a vulnerability within that sample. We normalize the samples by converting them into tokens and applying a word embedding model to convert them into vectors. While there are many word embedding models (e.g., TF-IDF, LDA, LSA, Ngram), some vulnerability detection studies have found vector representations generated by Word2Vec to be highly effective in capturing the context of a code sample when preprocessed efficiently [24, 32, 41]. Thus, we chose to use Gensim's Word2Vec model to transform our dataset into vectors. We flattened the Word2Vec matrices to shape (samples, average row length, features) for our models to take as input (see Fig. 1).

We apply the same sample tokenization technique as was used in [4]. Comments and white spaces are removed. Elements specific to the C/C++ language such as "calloc" or "sizeof" are left as is, but user-defined elements (such as variable names and function names) are assigned a generic naming convention as their token representation. For example, function name "goodB2G" changes to "func_0", and so on (see Fig. 1). The result is an array of tokens per slice, and each slice is stored in a pickle file and labeled with a unique sample ID in order to avoid duplicate IDs. Each pickle file contains an array of 7 elements including a list of tokens, its vulnerable state generated from SySeVR (0/1), the index of the potentially vulnerable line (used as midpoint when averaging our samples), a list of the functions within the sample, the assigned vulnerability syntax characteristic (API, AU, AE, PTR), their class label (CWE ID), and the sample's unique ID as assigned by us.

Our Word2Vec model was trained on the full dataset with hyper-parameters: alpha of 0.05, min frequency count of 0, max vocab size of *None*, hierarchical softmax of 0, negative sampling of 10, a vector length v of 30, a window size of 3, using the skip-gram training algorithm, and 10 epochs over the corpus. The difference in length between each code sample carried over to the vector representations per sample. That is, the number of vectors representing tokens per sample was imbalanced. Since BGRU and BLSTM expect their input size to be of the same length, we averaged out the lengths by calculating the average length of the samples ($\overline{X}$), then truncating or padding samples to match the average length. If the length was less than $\overline{X}$, then the remaining slots were padded with zero-filled vectors. If the length was above $\overline{X}$, then we calculated the midpoint to be where the index of the vulnerable syntax occurred and kept lines that were $\frac{\overline{X}}{2}$ before and $\frac{\overline{X}}{2}$ - 1 after the midpoint in order to retain as much context as possible around the vulnerable line. The result is a 3-dimensional shape of (N, $\overline{X}$, v), where N represents the total number of samples, $\overline{X}$ is the average length of 266 rows per sample (where each row is a vector that represents a token), and a column size of v. To reduce the training time for our deep neural networks, we flattened the matrices into 2-dimensions (54049, 7980), where each row still represents one sample but the columns are now all the tokens sitting side by side resulting in a feature length of $\overline{X} \times v$ shape. Preliminary tests confirmed that flattening the matrices did not change the outcome, and thus was safe to proceed with this technique.

Table 1: Model Hyper-Parameters

Optimizer	ADAM
Loss Function	categorical_crossentropy
Activation Function	softmax
Learning rate	0.001
Accuracy Metric	CategoricalAccuracy, Recall
Neuron Units	256
Hidden Layers	2
Batch Size	64
Epochs	60
Dropout Rate	0.2

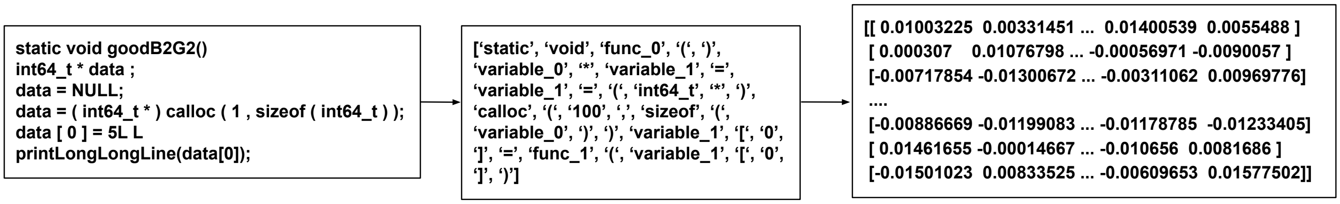
3.2 Training Phase

Previous natural language processing and software vulnerability detection/classification studies have used deep neural networks, such as BGRU or BLSTM, in their experiments because of the network's ability to better grasp the context of more complex data [4, 23, 24, 32, 41]. RNNs are a suitable selection since they deal with sequential data, specifically Gated Recurrent Units (GRU) and Long Short-Term Memory (LSTM) because they do not suffer from the Vanishing Gradient problem like other RNNs [23]. GRU and LSTM contain memory cells that help the models retain important features throughout each iteration of the training process. By making GRU and LSTM bidirectional, the model is better able to capture the context of a statement. Thus, we also chose to use BGRU and BLSTM for our multiclass classification of software vulnerabilities.

We performed a grid search with stratified 3-fold cross validation to find adequate hyper-parameter values to use for our models. For both models we use the sigmoid activation function in the hidden layers, hard sigmoid as our recurrent activation function, categorical cross-entropy loss, and categorical accuracy plus recall as our metrics. We use softmax as our dense layer's activation function since it is best for multiclass classification. The parameters of our model are listed in Table 1. Figure 2 shows the architecture for our neural networks.

3.3 Evaluation Metrics

To evaluate the effectiveness of our models in their ability to classify vulnerabilities, we calculate the mean and weighted F1-measure (aka F1-score), recall (aka true positive rate or TPR), false positive rate (FPR), and false negative rate (FNR). The average of each metric was calculated to see our models' effectiveness at a macro level. To take into consideration the imbalanced data, we also calculated the



```
static void goodB2G2()
int64_t * data ;
data = NULL;
data = ( int64_t * ) calloc ( 1 , sizeof ( int64_t ) );
data [ 0 ] = 5L L
printLongLongLine(data[0]);
```

```
['static', 'void', 'func_0', '(', ')',
'variable_0', '*', 'variable_1', '=',
'variable_1', '=', '(', 'int64_t', '*', ')',
'calloc', '(', '100', ',', 'sizeof', '(',
'variable_0', ')', ')', 'variable_1', '[', '0',
']', '=', 'func_1', '(', 'variable_1', '[', '0',
']', ')']
```

```
[[ 0.01003225  0.00331451 ...  0.01400539  0.0055488 ]
 [ 0.000307    0.01076798 ... -0.00056971 -0.0090057 ]
 [-0.00717854 -0.01300672 ... -0.00311062  0.00969776]
 ....
 [-0.00886669 -0.01199083 ... -0.01178785  -0.01233405]
 [ 0.01461655 -0.00014667 ... -0.010656  0.0081686 ]
 [-0.01501023  0.00833525 ... -0.00609653  0.01577502]]
```

Figure 1: An example of a code slice's conversion into tokens before transforming into vectors via Word2Vec

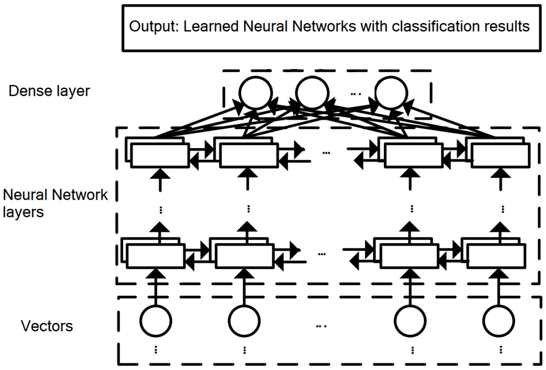


Figure 2: The architecture for our BGRU and BLSTM models.

weighted average by calculating the weight of each vulnerability class and multiplying it with each class's respective metric before calculating the total mean.

F1-score is a better indicator of our model's effectiveness than accuracy because we have an imbalanced data set, which causes our work to achieve higher accuracy for vulnerability classes with fewer positive samples.

Recall is useful for depicting our model effectiveness in correctly classifying vulnerabilities. The more true positives we have, the less false negatives there will be. Recall gives a sense of completeness in each class. We want to decrease the false negatives. Basically, out of the positive examples, how many did our model correctly identify versus how many were missed? False negatives can either represent missed vulnerabilities (labeled non-vulnerable) or misclassified vulnerabilities. The higher the recall, the lower the number of misclassified vulnerabilities. This is important because if the classifier misses some, then the vulnerability can go undetected and will get overlooked by the programmer. Also, if the classifier misclassifies them, then the programmer may not be able to recognize where the real vulnerability lies.

As part of the F1-score, precision gives us a better indication for our model's performance in correctly classifying a vulnerability. Here the classifier may have missed some vulnerabilities, but out of the predicted vulnerabilities, how many were actually correct? So here we would like to avoid false positives, which means the programmer may have to look at errors that are not actually errors.

3.4 Dataset

Our dataset comprises of 54,049 vulnerable samples obtained from SARD and NVD [24]. We use CWE IDs as the vulnerability types for these samples. In total the samples cover 50 CWE IDs. Each sample is a program slice that encapsulates a syntax characteristics of the vulnerable code. The slices are split into tokens, which are transformed to vectors to feed into deep learning models.

Table 2: The number of samples for each CWE ID.

CWE ID	Count	CWE ID	Count
23	2533	400	594
36	2498	401	296
78	1587	404	107
88	200	415	398
89	660	416	554
90	237	426	151
114	694	427	318
119	1940	457	519
121	5760	476	808
122	8356	506	166
124	3102	590	1556
126	1786	591	120
127	2735	617	143
134	2842	666	400
190	702	675	107
191	205	680	1318
194	1709	690	1037
195	1528	758	320
197	225	761	540
200	103	762	1416
252	118	771	329
253	118	773	201
319	584	785	171
363	128	787	144
369	400	789	1586

4 EXPERIMENTS AND RESULTS

The following Research Questions (RQs) are at the center of our experiments:

- RQ1: How effective are our models for multiclass vulnerability classification using the combined dataset of syntax vulnerability characteristics API, AE, AU, and PTR? Can we explain their (in)effectiveness?

- RQ2: How effective are our models for multiclass vulnerability classification in datasets containing only a single syntax vulnerability characteristic? Can we explain their (in)effectiveness?

We implement two deep neural networks, BLSTM and BGRU, in Python using Keras (which runs on top of TensorFlow) in order to answer these questions [19, 33]. We randomly select 80% of the dataset as the training set and 20% as our test set using stratified split technique so that the proportions of our 50 classes would remain intact. We run our experiments on a server that uses an AMD Ryzen Threadripper 3970X CPU running at 3.7GHz and has a NVIDIA Quadro RTX 5000 GPU.

4.1 Experiments for Answering RQ1

First, we trained and evaluated our two models on the combined dataset. From the results presented in Table 3, we discover that BGRU outperforms BLSTM with a mean F1 (M_F1) of 87.67% versus 84.34%, respectively. We also observe that the F1-scores for the vast majority of vulnerability types are above 80%, as shown in Fig. 3. Only 6 of our 50 vulnerability types have an F1-score below 80%, indicating a reasonable effectiveness of our models' ability to classify vulnerabilities.

To understand the F1-scores for different vulnerability classes, we take a deeper look on some of the best and worst performing classes. The two classes with the lowest F1 across both models are CWE-191 "Integer Underflow" (66.67% BGRU, 63.01% BLSTM) and CWE-617 "Reachable Assertion" (58.33% BGRU, 71.43% BLSTM).

CWE-191 has a total of 204 samples. Our model tends to misidentify this class with CWE-190 "Integer Overflow or Wraparound", CWE-121 "Stack-based Buffer Overflow", or CWE-122 "Heap-based Buffer Overflow". This is because these classes are identified with various yet similar arithmetic operations, making it hard for our models to precisely differentiate between them.

CWE-617 has a total of 143 samples. Our model often misidentifies this class with CWE-369 "Divide by zero", CWE-252 "Unchecked Return Value", and CWE-126 "Buffer Over-read". Because assertion statements often exist in code related with vulnerabilities in these classes, it is challenging to identify vulnerabilities belonging to CWE-617.

The two classes with the highest F1 scores are CWE-789 "Memory Allocation with Excessive Size Value" and CWE-761 "Free of Pointer not at Start of Buffer". Most of the samples for CWE-789 have a common pattern of dynamically allocating memory using the `malloc` function with the allocation size set in a variable, which misses the code to check the maximum allocation size. The unique pattern contributes to the successful classification of this class. The few samples in CWE-761 that were misclassified typically omit the call to `malloc` function or use the `new` operator instead.

Similarly, CWE-761 samples consistently contain the code that changes the value of a pointer before frees it. The majority of the samples use a `for` loop to change the value of a pointer and is typically followed by an `if` statement.

4.2 Experiments for Answering RQ2

Next, we trained and evaluated our two models over each individual vulnerability type. As we can see from Table 4, BGRU would score roughly the same if not slightly better than BLSTM across all types. Both models performed better among the AU and PTR types than with the API and AE types. The highest M_F1 over a single vulnerability type was in the BGRU + AU type with 86.68%. Our lowest scoring type was for BLSTM + API, which was unexpected since it had the 2nd highest number of samples after PTR. A Pearson Correlation Coefficient of 0.2531 between the F1 and counts confirms that there is not strong enough evidence to support a correlation between number of samples and correct predictions, indicating the problem lies elsewhere. We also observe that the M_F1 of BGRU + the combined dataset outperforms all the individually trained models, indicating that using the combined dataset improves the accuracy of our models.

The AE vulnerability type contains 9 CWE IDs. The worst performing one was CWE-23 "Relative Path Traversal" with an F1 of 0.0513. This weakness arises when a path sequence, such as '..', given by an external source does not get neutralized, which could result in the access of a restricted directory. This class was most commonly misclassified as CWE-36, the 2nd worst performing class of this type with an F1 of 0.6422. This weakness arises upon the failure of neutralizing an external input to a path containing an absolute path sequences, such as 'abs/path', leading to the same consequences as the prior example. We note that these weaknesses are better suited for the API type, but had many samples in the AE dataset we obtained from SySeVR. Nevertheless, the samples for these 2 classes shared many common patterns, such as their vulnerabilities containing the open function and omitting to neutralize or validate the external path passed in that is being opened. This causes their vectors to overlap in space, making it clear how CWE-23 and CWE-36 can be misidentified with each other.

CWE-124 "Buffer Underwrite (aka Buffer Underflow)" is found amongst the API, AU, and PTR types, and had low F1 scores of 0.5387, 0.6012, and 0.6667, respectively. CWE-124 is defined as the software writing "to a buffer using an index or pointer that references a memory location prior to the beginning of the buffer"[10]. To catch this, our model would need to know if the data is being written to an invalid (or not within scope) pointer address. In the AU realm, this CWE would often get predicted as CWE-127, CWE-126, CWE-121, and sometimes CWE-400. Likewise, CWE-127 "Buffer Under-Read" would commonly get misclassified as CWE-124. CWE-127 occurs when the software tries to read a buffer using an index or pointer that references a memory location prior to the beginning of the buffer. The distinction between these two classes in our samples is the presence of a read method existing in the latter. However, since many of the other lines contain similar tokens and pointer arithmetic, these two would get cross-wired in the vector space.

Thus, it seems as though the combination of the datasets introduces more variance, allowing our models to better discover the minute distinctions between classes.

5 CONCLUSION

We present a simple approach to categorizing software vulnerabilities using deep learning models and CWE IDs as labels. Specifically, we train two different deep neural networks on 50 classes of vulnerabilities across four types of syntax characteristics, then compare

Table 3: Experimental results of our models against the SySeVR dataset

Model	M_TPR	M_FPR	M_FNR	M_F1	W_TPR	W_FPR	W_FNR	W_F1
BLSTM + SySeVR	0.8484	0.0024	0.1516	0.8434	0.8848	0.0059	0.0023	0.8844
BGRU + SySeVR	0.8680	0.0020	0.1320	0.8767	0.9052	0.0055	0.0019	0.9046
SySeVR's BGRU	N/A	0.0020	0.0147	0.8580	N/A	N/A	N/A	N/A

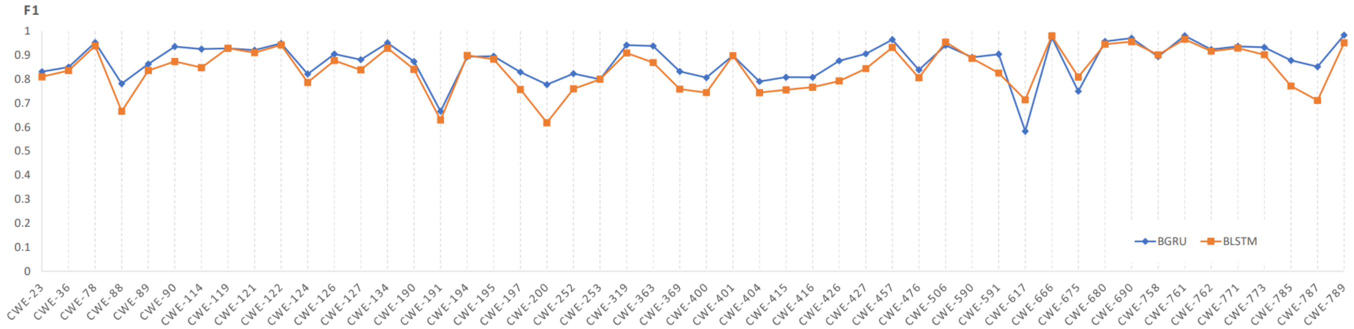


Figure 3: F1-scores for both models using SySeVR's dataset for classes that have a minimum of 100 samples.

Table 4: Experimental results for each vulnerable syntax characteristic

Model	M_TPR	M_FPR	M_FNR	M_F1	W_TPR	W_FPR	W_FNR	W_F1
BLSTM + API	0.7755	0.0085	0.2245	0.7839	0.8050	0.0163	0.0078	0.8021
BLSTM + AE	0.8407	0.0142	0.1593	0.8092	0.8830	0.0112	0.0130	0.8654
BLSTM + AU	0.8600	0.0120	0.1400	0.8599	0.8410	0.0208	0.0106	0.8397
BLSTM + PTR	0.8219	0.0048	0.1781	0.8212	0.8421	0.0098	0.0045	0.8416
BGRU + API	0.8046	0.0076	0.1954	0.8153	0.8240	0.0150	0.0070	0.8229
BGRU + AE	0.8417	0.0143	0.1583	0.8087	0.8830	0.0114	0.0130	0.8645
BGRU + AU	0.8649	0.0121	0.1351	0.8668	0.8394	0.0207	0.0107	0.8385
BGRU + PTR	0.8440	0.0041	0.1560	0.8562	0.8662	0.0096	0.0038	0.8660

them against each other and against other similar models. Our evaluation demonstrates a reasonable performance in our models' ability to categorize vulnerabilities with a mean F1 of 87.67%.

ACKNOWLEDGMENTS

This work was supported in part by US National Science Foundation (NSF) grant CNS-2153474.

REFERENCES

[1] 533 million Facebook users' phone numbers and personal data have been leaked online 2021. https://www.businessinsider.com/stolen-data-of-533-million-facebook-users-leaked-online-2021-4.

[2] All You Need to Know About WannaCry Ransomware 2021. https://www.mimecast.com/blog/all-you-need-to-know-about-wannacry-ransomware/.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290353

[4] Amy Aumpansub and Zhen Huang. 2021. Detecting Software Vulnerabilities Using Neural Networks. In *2021 13th International Conference on Machine Learning and Computing* (Shenzhen, China) *(ICMLC 2021)*. Association for Computing Machinery, New York, NY, USA, 166–171. https://doi.org/10.1145/3457682.3457707

[5] Amy Aumpansub and Zhen Huang. 2022. Learning-Based Vulnerability Detection in Binary Code. In *2022 14th International Conference on Machine Learning and Computing (ICMLC)* (Guangzhou, China) *(ICMLC 2022)*. Association for Computing Machinery, New York, NY, USA, 266â€“271. https://doi.org/10.1145/3529836.3529926

[6] Zeki Bilgin, Mehmet Akif Ersoy, Elif Ustundag Soykan, Emrah Tomur, Pinar Çomak, and Leyli Karaçay. 2020. Vulnerability Prediction From Source Code Using Machine Learning. *IEEE Access* 8 (2020), 150672–150684. https://doi.org/10.1109/ACCESS.2020.3016774

[7] David Brumley, T Chiueh, R Johnson, and H Lin. 2007. RICH: Automatically protecting against integer-based vulnerabilities. In *Proceedings of Ndss '07*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.73.7344&rep=rep1&type=pdf%5Cnpapers3://publication/uuid/C0320481-2B40-4264-B778-CBB64ECEFAA4

[8] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. 2003. {PointGuard}: Protecting Pointers from Buffer Overflow Vulnerabilities. In *In Proceedings of the 12th Usenix Security Symposium.*

[9] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. *ISSTA 2018 - Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2018), 95–105. https://doi.org/10.1145/3213846.3213848

[10] CWE. 2021 [Online]. Common Weakness Enumeration. https://cwe.mitre.org/data/index.html

[11] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (New Orleans, Louisiana, USA) *(CODASPY '16)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2857705.2857720

[12] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (New Orleans, Louisiana, USA) *(CODASPY '16)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2857705.2857720

[13] Zhen Huang, Trent Jaeger, and Gang Tan. 2021. Fine-Grained Program Partitioning for Security. In *Proceedings of the 14th European Workshop on Systems Security* (Online, United Kingdom) *(EuroSec '21)*. Association for Computing Machinery, New York, NY, USA, 21–26. https://doi.org/10.1145/3447852.3458717

[14] Zhen Huang and David Lie. 2014. Ocasta: Clustering Configuration Settings for Error Recovery. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*. IEEE Computer Society, Washington, DC, USA, 479–490. https://doi.org/10.1109/DSN.2014.51

[15] Zhen Huang and Gang Tan. 2019. Rapid Vulnerability Mitigation with Security Workarounds. In *Proceedings of the 2nd NDSS Workshop on Binary Analysis Research (BAR '19)*. https://doi.org/10.14722/bar.2019.23052

[16] Zhen Huang and Marc White. 2022. Semantic-Aware Vulnerability Detection. In *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*. 68–75. https://doi.org/10.1109/CSR54599.2022.9850330

[17] Zhen Huang and Xiaowei Yu. 2021. Integer Overflow Detection with Delayed Runtime Test. In *Proceedings of the 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021 (ARES 2021)*. ACM, 28:1–28:6. https://doi.org/10.1145/3465481.3465771

[18] Sanghoon Jeon and Huy Kang Kim. 2021. AutoVAS: An automated vulnerability analysis system with a deep learning approach. *Computers and Security* 106 (2021), 102308. https://doi.org/10.1016/j.cose.2021.102308

[19] Keras. [n. d.]. About Keras. Accessed Oct. 18, 2021 [Online]. https://keras.io/about/

[20] S. Kim, S. Woo, H. Lee, and H. Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. 595–614. https://doi.org/10.1109/SP.2017.62

[21] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. 2019. PathMiner: a library for mining of path-based representations of code. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 13–17.

[22] Zhen Li, Deqing Zou, Jing Tang, Zhihao Zhang, Mingqian Sun, and Hai Jin. 2019. A Comparative Study of Deep Learning-Based Vulnerability Detection System. *IEEE Access* 7 (2019), 103184–103197. https://doi.org/10.1109/ACCESS.2019.2930578

[23] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. Association for Computing Machinery, New York, NY, USA, 201–213. https://doi.org/10.1145/2991079.2991102

[24] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021), 1–1. https://doi.org/10.1109/TDSC.2021.3051525

[25] Z. Li, D. Zou, Shouhuai Xu, Xinyu Ou, H. Jin, S. Wang, Zhijun Deng, and Y. Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, Vol. abs/1801.01681.

[26] LockFile: Ransomware Uses PetitPotam Exploit to Compromise Windows Domain Controllers 2021. https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/lockfile-ransomware-new-petitpotam-windows.

[27] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. *SIGPLAN Not.* 51, 1 (jan 2016), 298–312. https://doi.org/10.1145/2914770.2837617

[28] James Mickens, Martin Szummer, and Dushyanth Narayanan. 2007. Snitch: interactive decision trees for troubleshooting misconfigurations. In *SYSML'07: Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques* (Berkeley, CA, USA). USENIX Association, 1–6.

[29] Dhaval Miyani, Zhen Huang, and David Lie. 2017. BinPro: A Tool for Binary Source Code Provenance. arXiv:1711.00830. *arXiv* (2017).

[30] Samuel Ndichu, Sangwook Kim, Seiichi Ozawa, Takeshi Misu, and Kazuo Makishima. 2019. A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors. *Applied Soft Computing* 84 (2019), 105721. https://doi.org/10.1016/j.asoc.2019.105721

[31] Hovafv Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-Space Randomization. In *Proceedings of the11th {ACM} Conference on Computer and Communications Security ({CCS})*. 298–307.

[32] Gaigai Tang, Lianxiao Meng, Huiqiang Wang, Shuangyin Ren, Qiang Wang, Lin Yang, and Weipeng Cao. 2020. A Comparative Study of Neural Network Techniques for Automatic Software Vulnerability Detection. In *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 1–8. https://doi.org/10.1109/TASE49443.2020.00010

[33] TensorFlow. 2021 [Online]. KerasClassifier. https://www.tensorflow.org/api_docs/python/tf/keras/wrappers/scikit_learn/KerasClassifier

[34] Chin-Wei Tien, Shang-Wen Chen, Tao Ban, and Sy-Yen Kuo. 2020. Machine Learning Framework to Analyze IoT Malware Using ELF and Opcode Features. *Digital Threats: Research and Practice* 1 (2020), 1–19. Issue 1. https://doi.org/10.1145/3378448

[35] VMware Flaw a Vector in SolarWinds Breach? 2020. https://krebsonsecurity.com/2020/12/vmware-flaw-a-vector-in-solarwinds-breach/.

[36] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically Learning Semantic Features for Defect Prediction. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 297–308. https://doi.org/10.1145/2884781.2884804

[37] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.

[38] F. Wu, J. Wang, J. Liu, and W. Wang. 2017. Vulnerability detection with deep learning. In *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*. 1298–1302. https://doi.org/10.1109/CompComm.2017.8322752

[39] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. 2011. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies* (San Francisco, CA) *(WOOT'11)*. USENIX Association, USA, 13.

[40] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. 2011. Context-based online configuration-error detection. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*. 28–28.

[41] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2021. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing* 18, 5 (2021), 2224–2236. https://doi.org/10.1109/TDSC.2019.2942930

[42] R. Řehůřek. 2019 [Online]. models.word2vec – Word2vec embeddings. Gensim. https://radimrehurek.com/gensim_3.8.3/models/word2vec.html