Code2Tensor: Tensor Representation of Code for Vulnerability Detection

Shuo Liu

***Abstract*— With the explosive growth of the amount of code in recent years, more and more vulnerabilities are weakening the overall security of computer systems. However, it is hard to detect the vulnerabilities in large projects, because of the complex project structures, interactive calls of multiple programming languages, and dependence on domain knowledge. An expressive code embedding can facilitate the machine learning model to understand abstract semantics. Code2Vec pioneers research in this field, but it only quantizes the codes into one-dimensional vectors, losing a lot of latent information. In this paper, we put forward a Code2Tensor framework to get a tensor representation of codes. By feeding the tensors of C++ codes to deep learning models for vulnerability detection, we show the huge potential of applying the code tensorization to other deep learning pipelines. Our open-source codes are available at https://github.com/LovelyBuggies/Code2Tensor.**

I. INTRODUCTION

Software vulnerabilities are flaws that degrade the performance, cause the expected functionality and damage the system security. Statically or dynamically detecting vulnerabilities before project deployment can help improve efficiency and reduce losses caused by security issues. Although dynamic analysis tools can help to detect vulnerabilities in an efficient way, most of them suffer from low code coverage. In the static analysis for huge projects, multiple modules with different programming languages and inter-project packages call each other, bringing a lot of noise to this task. In this paper, to detect vulnerabilities, we use concrete syntax trees to comprehensively represent the codes as tensors and feed them to prediction models.

A. Related Work

There are two kinds of vulnerability detection techniques used before deployment, i.e., dynamic and static analysis. Dynamic techniques like [1], [2], [3] often require practitioners to provide a comprehensive set of tests, and sometimes cannot cover all attracting cases [4], [5]. Though more reliable, static analysis suffers from high false positives and has difficulty to identify the type of vulnerabilities. Some work use hybrid approaches [6], [7], [8], but they inherit both their drawbacks and are not efficient in practice [4].

Recently, there is a growing trend of applying machine learning in static vulnerability detection [9], [10], [11]. However, it is pretty challenging because the machine learning detection systems fail to understand the semantic meaning of code vulnerabilities like practitioners do [4]. Human experts often have profound knowledge of the code base and can better read the syntax and semantics of code structure. To address this issue, some works apply text mining techniques to find the patterns in codes [12], [13], [14]. However, the surface text mining works can hardly extract long-term contextual relations from long source codes, which motivates researchers to find structural information of the code.

Codes are special kinds of texts that have a well-defined structure, it is intuitive to take their structural information into consideration. Some works apply static code analysis tools to extract the code structures from graph-based or sequence-based representation, like abstract syntax trees, control flow graphs, program dependency graphs, etc [4], [15], [16], [17], [18]. Though, these kinds of structures can not be fed into the machine learning model directly. Thus, a lot of studies emerge to embed the code. For example, Code2Vec [19] expresses code snippets as abstract syntax tree, and use context path to train deep learning models to make a prediction of function names; SmartEmbed [20] represents the codes as vectors to find bugs in smart contracts on Ethereum; Flow2Vec [21] transforms code data-flows into vectors for code classification and summarization. While, most of these works only get code embedding as vectors, which is not informative enough to get all latent features of those structures. Some deep learning models like convolutional neural networks (CNN), recurrent neural networks (RNN), long-short term memory (LSTM), graph neural networks, etc, are reliable to deal with high-dimensional input data [22], [23], which release the potential to represent codes as more complicated tensors.

B. Contributions

The contributions of this paper include:

1) We propose Code2Tensor framework, which can obtain a strongly expressive tensor representation of codes from the abstract syntax trees.
2) We demonstrate a deep learning pipeline that uses C++ code tensors as the inputs for deep neural networks to detect vulnerabilities.
3) Based on IBM/D2A dataset that uses differential analysis to label the vulnerable codes, we develop some baselines with various common-used deep neural networks and compare them.

The rest of the paper is organized as follows. Sec. II introduces the workflow of our Code2Tensor framework. Sec. IV presents prediction results of various deep neural networks for vulnerability detection. Sec. V concludes this study.

Shuo Liu is with the Department of Computer Science, Columbia University, New York City, NY 10027 USA (e-mail: sl4921@columbia.edu).

II. Code2Tensor Framework

In this section, we introduce the details of the Code2Tensor framework (Fig. 1).

A. Concrete Syntax Tree

To tensorize code snippets (example shown in Fig. 2), we first need to one-to-one map the code grammar to a tree-form. Unlike abstract syntax trees, the concrete syntax tree maintains a comprehensive representation of codes. Tree-sitter is an efficient parser generator that can build concrete syntax trees from source codes of multiple programming languages. Fig. 3 is the concrete syntax tree of the example, whose nodes include the following features:

```
void BN_init(BIGNUM *a)
{
    memset(a,0,sizeof(BIGNUM));
    bn_check_top(a);
}
```

Fig. 2: A code snippet example (the 79^{th} code snippet in the training set of IBM/D2A).

- Type records the type of the field, e.g., primitive_type, type_identifier, identifier, pointer, etc.
- The value of a field can be obtained from the start and end indices, e.g., 'void', '(', 'static', '*', etc.
- By depth-first traversing the tree and recording the leaf node values, we can get tokens in the original order in the code.

```
translation_unit [0, 0] - [5, 0]
  function_definition [0, 0] - [4, 1]
    type: primitive_type [0, 0] - [0, 4]
    declarator: function_declarator [0, 5] - [0, 23]
      declarator: identifier [0, 5] - [0, 12]
      parameters: parameter_list [0, 12] - [0, 23]
        parameter_declaration [0, 13] - [0, 22]
          type: type_identifier [0, 13] - [0, 19]
          declarator: pointer_declarator [0, 20] - [0, 22]
            declarator: identifier [0, 21] - [0, 22]
    body: compound_statement [1, 0] - [4, 1]
      expression_statement [2, 1] - [2, 28]
        call_expression [2, 1] - [2, 27]
          function: identifier [2, 1] - [2, 7]
          arguments: argument_list [2, 7] - [2, 27]
            identifier [2, 8] - [2, 9]
            number_literal [2, 10] - [2, 11]
            sizeof_expression [2, 12] - [2, 26]
              value: parenthesized_expression [2, 18] - [2, 26]
                identifier [2, 19] - [2, 25]
      expression_statement [3, 1] - [3, 17]
        call_expression [3, 1] - [3, 16]
          function: identifier [3, 1] - [3, 13]
          arguments: argument_list [3, 13] - [3, 16]
            identifier [3, 14] - [3, 15]
```

Fig. 3: The corresponding concrete syntax tree for the code snippet in Fig. 2. For each node, it shows its type, start index, and end index.

In this way, we can split the codes into multi-dimensional tokens by depth-first searching the leaf nodes of the concrete syntax tree.

B. Token2Tensor

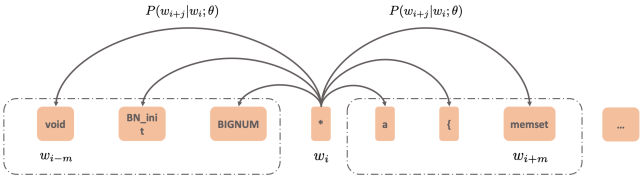


Fig. 4: Skip-gram algorithm for Word2Vec model.

To change the tokens to tensors, we facilitate Word2Vec model to embed the values and types of the code snippet [24]. In Word2Vec model, we randomly initialize the vector for each word in the vocabulary. We go through each position i and select it as the center word c. To identify the context words o, we look at other words with a window size of m around the selected word $[w_{i-m}, w_i) \cup (w_i, w_{i+m}]$ as shown in Fig. 4. Then we can maximize the likelihood of the surrounding words according to Eq. 1.

$$L(\theta) = \prod_{i=1}^{n} \prod_{j=-m \neq 0}^{m} P(w_{i+j}|w_i;\theta), \tag{1}$$

the probability is calculated by

$$P(O=o|C=c) = \frac{\exp(\boldsymbol{u}_o^\intercal \boldsymbol{v}_c)}{\sum_{w_i, i=0,\dots,n} \exp(\boldsymbol{u}_{w_i}^\intercal \boldsymbol{v}_c)}, \tag{2}$$

where $\boldsymbol{u}$ and $\boldsymbol{u}$ are two sets of vectors, which denotes when w_i is the selected word or the context word respectively. Then the model are descending the gradient by adjusting the parameter

$$\theta = \begin{bmatrix} \boldsymbol{v}_{w_1} & \cdots & \boldsymbol{v}_{w_n} & \boldsymbol{u}_{w_1} & \cdots & \boldsymbol{u}_{w_n} \end{bmatrix}$$

to minimize the negative log function of the likelihood

$$\min_{\theta} J(\theta) = -\frac{1}{n} \log L(\theta). \tag{3}$$

Some of the important parameters in Word2Vec are listed below:

- Vocabulary: All words (token values or types) that are input into vocabulary.
- Vector Size: The size of the output vector.
- Window Size: The range of nearby words to predict the word vectors.
- Minimum Count: The minimum number of occurrences of the word being trained.
- Workers: Number of workers for parallel training.

C. Prediction Models

We build some prediction models for detecting the vulnerabilities in the code functions, more details will be introduced in Sec. III.

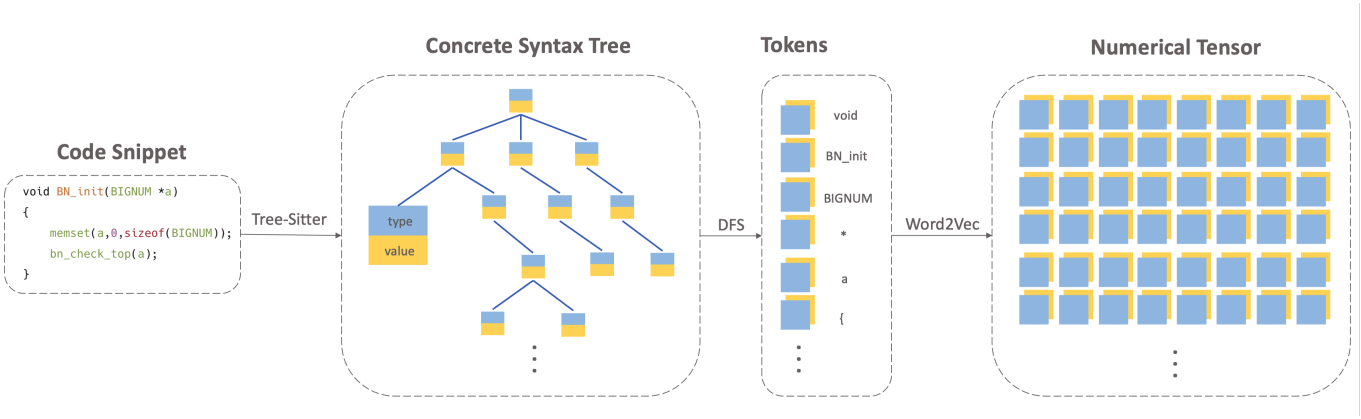


Fig. 1: Code2Tensor framework.

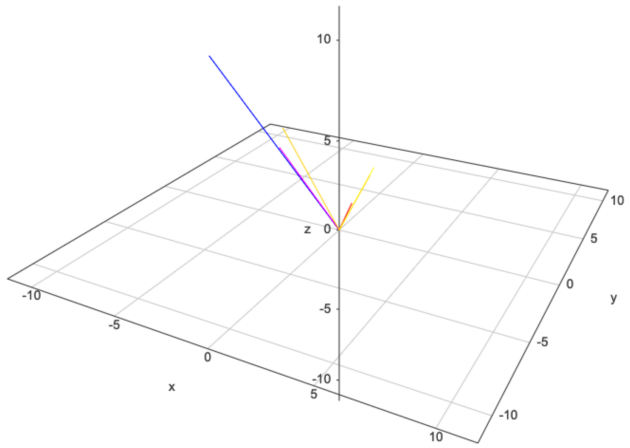


Fig. 5: The first 5 value vectors of the code snippet with length 3 in Fig. 2, after training Word2Vec model using IBM/D2A vocabulary, i.e., 'void', 'BN_init', '(', 'BIGNUM', '*'.

III. Deep Learning Pipeline for Vulnerability Detection

In this section, we introduce the deep learning pipeline to detect vulnerabilities based on the Code2Tensor framework. Deep learning is able to identify and extract features from high-dimensional tensors. [25] provides a detailed review of vulnerability detection using deep learning techniques. The process of using deep neural networks to predict the vulnerability of several neural networks is shown in the following subsections.

A. Multi-Layer Perceptron (MLP)

Considering that in datasets, the input code tensors tend to be very different in size, we need to unify the size of tensors. We employ a trick similar to image resizing to scale tensors of different sizes to the same. Since MLP takes vectors as inputs, we add a flatten layer to transform the tensor into one dimension.

In the model, the hidden layers are fully-connected, and there is an activation layer acts as a gate to determine whether to activate a neuron.

B. CNN

In the field of deep learning, CNN is often used for the classification of image tasks, benefiting from its strong ability to perceive edge information [26]. In addition to the tensor resize and flatten, we apply spatial filters in convolutional layers to remark the dominant features of the code tensors in CNN. And to reduce the dimensions of the features, we also use a pooling layer after each convolutional layer.

C. LSTM

Codes always have contextual structures, so it is quite intuitive to consider applying recurrent neural networks that have feedback connections to our task. LSTM is a kind of RNN that uses both standard and special units [27]. By adding an LSTM layer, the network is able to maintain knowledge for a long period.

D. Deep Learning Pipeline

Alg. 1 shows the pipeline of vulnerability detection for general deep learning models.

IV. Experiments

In this section, we apply the Code2Tensor framework to IBM/D2A Leaderboard dataset. We first introduce the set-up of our experiment environments and then present numerical results by different deep neural networks.

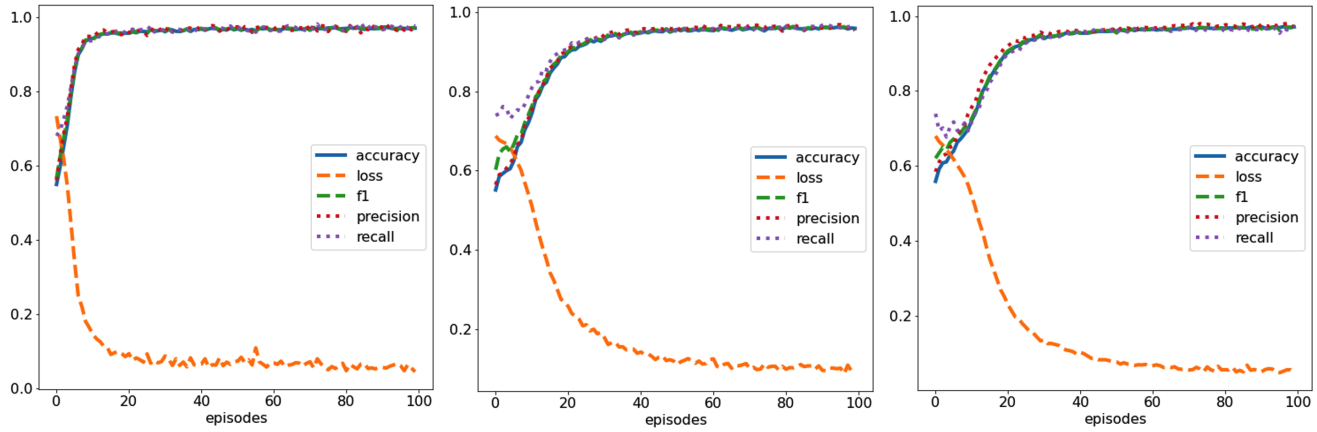
A. Set-up

In this paper, we use IBM/D2A LeaderBoard function dataset, which extracts code snippets from 6 well-known C++ open-source projects [28]. Different from other synthetic vulnerability detection datasets based on predefined patterns like [29], [10], [30], it has two static analyzers and uses differential analysis to generate labels. Besides bug code source codes, IBM/D2A contains comprehensive information about bug reports (trace), bug type, bug URL, labeler source, etc, which may be useful for further research (discussed in Sec.VI).

In our Word2Vec model, we use all tokens extracted from concrete syntax trees in the training set and validation set as

	MLP				CNN				LSTM			
	Accuracy	F1	Precision	Recall	Accuracy	F1	Precision	Recall	Accuracy	F1	Precision	Recall
Training	97.18%	97.21%	99.91%	94.81%	91.64%	91.42%	98.36%	85.84%	97.44%	97.50%	97.20%	97.98%
Validation	53.02%	53.01%	54.61%	53.66%	54.72%	49.66%	59.63%	44.97%	51.01%	53.16%	52.18%	56.28%

TABLE I: Comparison of our baseline deep learning models on the IBM/D2A with respect to 4 metrics.



(a) MLP. (b) CNN. (c) LSTM.

Fig. 6: Training curves of the deep learning models.

vocabulary. The size of the first dimension is the token list length. The size of the second dimension is the Word2Vec vector length, which is set to 64 for all of our deep learning models. The last dimension is the feature number, where we set to 2, representing the values and types of the tokens. Value and type Word2Vec models are trained separately. The window sizes of both models are 4, with the minimum word count 1. Our value Word2Vec is a comprehensive vocabulary, it can go through 86% non-seen snippets in the test set.

The network structures of the deep neural networks we used are listed below.

1) MLP: We construct MLP with 8 fully connected layers after flattening the tensor, $(128 \times 64 \times 32 \times 16 \times 8 \times 1)$, each of them is activated using ReLU and we get the prediction results after the sigmoid function. We use binary cross-entropy as loss function and RMSprop as optimizer. The learning rate is 0.001.
2) CNN: In our CNN model, we have 7 convolutional layers and pooling layers followed by a fully-connected layer after flattening. The number of neurons of the convolutional layers are $(16, 32, 64, 128, 64, 32, 16)$, with all pooling windows (2×2). The fully-connected layer has 64 neurons. We use the same loss function and optimizer with MLP. The learning rate is 0.0015.
3) LSTM: Our LSTM model is relatively shallow, with only one LSTM layer with 128 neurons followed by a fully-connected layer with 128. We use the same loss function and optimizer with MLP. The learning rate is 0.001.

We use the accuracy, loss, F1-score, precision, and recall as evaluation metrics. The models are all trained on a CPU device with a single worker for 100 episodes. The results are shown in the following subsection.

B. Results and Discussion

We train different deep learning models and compare them with respect to 4 metrics in Tab. I. As shown in Fig. 6, the training process of MLP converges the fastest and no more than 20 episodes; CNN converges in 40 episodes; and LSTM converges in the 30 episodes. CNN performs the best among the three, achieving 54.72. While LSTM costs the most time and has the lowest accuracy 51.01. We find that the prediction accuracy of all models is always better than 50, and always has a high recall rate, demonstrating that our models are taking effect. Although all models converge with high accuracy for the training set, we find the general performance on the validation set is far lower than that on the training set, 52% versus 98%, this implies there may have other important information in codes that needs to be expressed by other methods.

C. Ablation Study

In this part, we conduct an ablation study to see how the hyperparameters can affect the general performance of Word2Tensor framework.

From the Tab. II, we can see that when tuning the hyperparameters in a proper range, the LSTM model suffers the most fluctuation in accuracy. Among those hyperparameters, removing the type feature, changing vector length, applying different loss functions, and using other optimizers during

Algorithm 1 Pipeline of Vulnerability Detection

Input: Code snippets with vulnerability label. Word2Vec model $\mathcal{W}$ and its parameters $\theta_{\mathcal{W}}$ window size m, vector length l; classification model $\mathcal{C}$ and its parameters $\theta_{\mathcal{C}}$, training episodes τ, batch size b, learning rate α, loss function l, optimizer σ.
Parse the codes to build concrete syntax trees.
Traverse the syntax trees and obtain the tokens.
Aggregate vocabulary from tokens.
for Each word w_i **do**
    Obtain its context words $[w_{i-m}, w_i) \cup (w_i, w_{i+m}]$.
    Calculate the likelihood according to Eq. 1.
    Minimize its negative log by updating $\theta_{\mathcal{W}}$.
end for
Get the tensor representation of each snippet.
Train classification model $\mathcal{C}$.
for $episode \leftarrow 1$ *to* τ **do**
    for every b samples **do**
        Calculate the total losses according to l.
        Descend gradient by σ with α by updating $\theta_{\mathcal{C}}$.
    end for
end for
Given a test code snippet, get code tensor by $\theta_{\mathcal{W}}$ and make prediction by $\theta_{\mathcal{C}}$.

training have a huge influence on the general performance. This reveals the following processes of Code2Tensor framework are especially important to vulnerability detection, i.e., how to interpret the concrete syntax trees, how much can Word2Vec represent codes, how to measure the difference between prediction and ground-truth, and which method is used to minimize the loss function.

V. CONCLUSIONS

In this paper, we develop a Code2Tensor framework to represent code snippets as tensors. We first use the static code analysis tool to build the concrete syntax trees and then embed the "words" in the leaf nodes as tensors. By feeding the tensors to deep learning models, we demonstrate the pipeline for code vulnerability detection. Our experiment results show that our code embedding is taking effect and can achieve 55% accuracy. We also conduct an ablation study to observe how different deep learning models are affected by various hyperparameters. We find LSTM fluctuates the most, and the code embedding length and whether use type in the concrete syntax trees is important to the general performance of deep learning models.

Though tensorizing the code release more potential than simply embedding them to vectors [19], future work could make this idea more general. In the future, we can collect the paths of the syntax trees to mine more latent information. Besides, we will better express the codes by taking other representations into account with more sophisticated deep learning pipelines.

	MLP	**CNN**	**LSTM**
Removing Type Feature	$+0.43\%$	-0.71%	-0.02%
Window size 2	-0.18%	-0.06%	-0.24%
Window Size 8	-0.21%	-0.41%	$+0.04\%$
Vector Length 32	-1.23%	-0.72%	-0.49%
Vector Length 128	$+0.16\%$	-0.07%	-0.09%
Learning Rate 0.01	-0.13%	-0.12%	-0.28%
Learning Rate 10^{-4}	$+0.15\%$	$+0.08\%$	-0.29%
MSE Loss Function	-0.31%	-0.14%	$+0.22\%$
MAE Loss Function	$+0.01\%$	-0.04%	-0.17%
SGD Optimizer	-0.17%	-0.06%	$+0.02\%$
Adam Optimizer	$+0.18\%$	-0.37%	$+0.26\%$

TABLE II: The accuracy increment or decrement of the models on the validation set after changing some hyperparameters.

REFERENCES

[1] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[2] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 406–415, 2014.

[3] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *ACM SIGOPS Operating Systems Review*, 40(4):15–27, 2006.

[4] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020.

[5] Zhen Li, Deqing Zou, Jing Tang, Zhihao Zhang, Mingqian Sun, and Hai Jin. A comparative study of deep learning-based vulnerability detection system. *IEEE Access*, 7:103184–103197, 2019.

[6] Ruoyu Zhang, Shiqiu Huang, Zhengwei Qi, and Haibin Guan. Combining static and dynamic analysis to discover software vulnerabilities. In *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 175–181. IEEE, 2011.

[7] Roopak Surendran, Tony Thomas, and Sabu Emmanuel. A tan based hybrid model for android malware detection. *Journal of Information Security and Applications*, 54:102483, 2020.

[8] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[9] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):1–36, 2017.

[10] Nan Sun, Jun Zhang, Paul Rimba, Shang Gao, Leo Yu Zhang, and Yang Xiang. Data-driven cybersecurity incident prediction: A survey. *IEEE communications surveys & tutorials*, 21(2):1744–1772, 2018.

[11] Liu Liu, Olivier De Vel, Qing-Long Han, Jun Zhang, and Yang Xiang. Detecting and preventing cyber insider threats: A survey. *IEEE Communications Surveys & Tutorials*, 20(2):1397–1417, 2018.

[12] Yulei Pang, Xiaozhen Xue, and Akbar Siami Namin. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In *2015 IEEE 14th International Conference on*

Machine Learning and Applications (ICMLA), pages 543–548. IEEE, 2015.

[13] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.

[14] Dumidu Wijayasekara, Milos Manic, and Miles McQueen. Vulnerability identification and classification via text mining bug databases. In *IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society*, pages 3612–3618. IEEE, 2014.

[15] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 359–368, 2012.

[16] Hantao Feng, Xiaotong Fu, Hongyu Sun, He Wang, and Yuqing Zhang. Efficient vulnerability detection based on abstract syntax tree and deep learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 722–727. IEEE, 2020.

[17] Minmin Zhou, Jinfu Chen, Yisong Liu, Hilary Ackah-Arthur, Shujie Chen, Qingchen Zhang, and Zhifeng Zeng. A method for software vulnerability detection based on improved control flow graph. *Wuhan University Journal of Natural Sciences*, 24(2):149–160, 2019.

[18] Binbin Qu, Beihai Liang, Sheng Jiang, and Chutian Ye. Design of automatic vulnerability detection system for web application program. In *2013 IEEE 4th International Conference on Software Engineering and Service Science*, pages 89–92. IEEE, 2013.

[19] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[20] Zhipeng Gao, Vinoj Jayasundara, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 394–397. IEEE, 2019.

[21] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. Flow2vec: value-flow-based precise code embedding. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, 2020.

[22] Mingyang Li and Zequn Wang. Deep learning for high-dimensional reliability analysis. *Mechanical Systems and Signal Processing*, 139:106399, 2020.

[23] Runpu Chen, Le Yang, Steve Goodison, and Yijun Sun. Deep-learning approach to identifying cancer subtypes using high-dimensional genomic data. *Bioinformatics*, 36(5):1476–1483, 2020.

[24] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[25] Peng Zeng, Guanjun Lin, Lei Pan, Yonghang Tai, and Jun Zhang. Software vulnerability analysis and discovery using deep learning techniques: A survey. *IEEE Access*, 8:197158–197172, 2020.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[27] Klaus Greff, Rupesh K. Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232, 2017.

[28] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. D2a: A dataset built for ai-based vulnerability detection methods using differential analysis. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018.

[30] Young-Su Jang and Jin-Young Choi. Detecting sql injection attacks using query result size. *Computers & Security*, 44:104–118, 2014.

VI. Appendix

A. Link of Tools We Used

1) IBM/D2A: https://github.com/IBM/D2A
2) Learderboard Dataset Description: https://dax-cdn.cdn.appdomain.cloud/dax-d2a/1.0.0/data-preview/index.html
3) Tree-Sitter: https://github.com/tree-sitter/tree-sitter
4) GenSim Word2Vec: https://radimrehurek.com/gensim/models/word2vec.html
5) Keras: https://keras.io/

B. IBM/D2A Function Dataset Leaderboard

Model	Team	Organization	Date	Accuracy
VulBERTa-MLP	ACS Group	-	11/10/2021	62.3
VulBERTa-CNN	ACS Group	-	11/10/2021	60.7
C-BERT	AI4VA	IBM Research	03/26/2021	60.2
AugSA-S	AI4VA	IBM Research	03/26/2021	55.2
AugSA-V	AI4VA	IBM Research	03/26/2021	45.6

Fig. 7: IBM/D2A top-5 models with only function data.

As shown in Fig. 7, these models can reach around 62 accuracy in the test set, reflecting that the 54.7 accuracy that our best CNN model can achieve on the validation set is not bad.

C. Future Work 1: Multi-Modal Deep Learning

As we only extract information from the leaf nodes of the concrete syntax tree, we essentially use it as a tokenizer. However, there may contain far more information in this data structure that can help us make the vulnerability prediction. Besides, we can also obtain latent information from other representations of codes, i.e., other modalities, and take them all as inputs to the deep neural networks as shown in Fig. 8. With more input information, we are expected to use more complicated networks correspondingly, like multi-modal learning networks.

Besides, another way to integrate prediction results in terms of different representations is ensemble learning as shown in Fig. 9, where each model makes its own prediction and models ensemble together to vote for the final result.

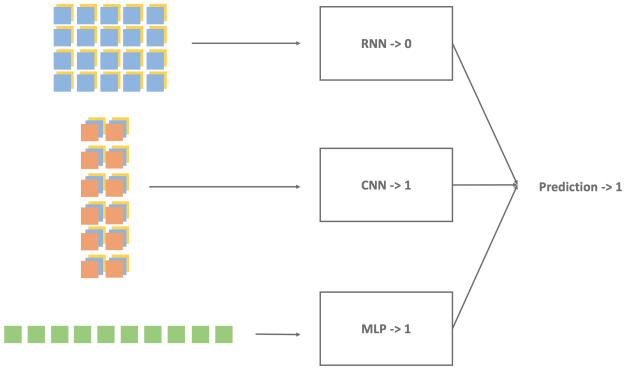


Fig. 9: Brief illustration of ensemble learning.

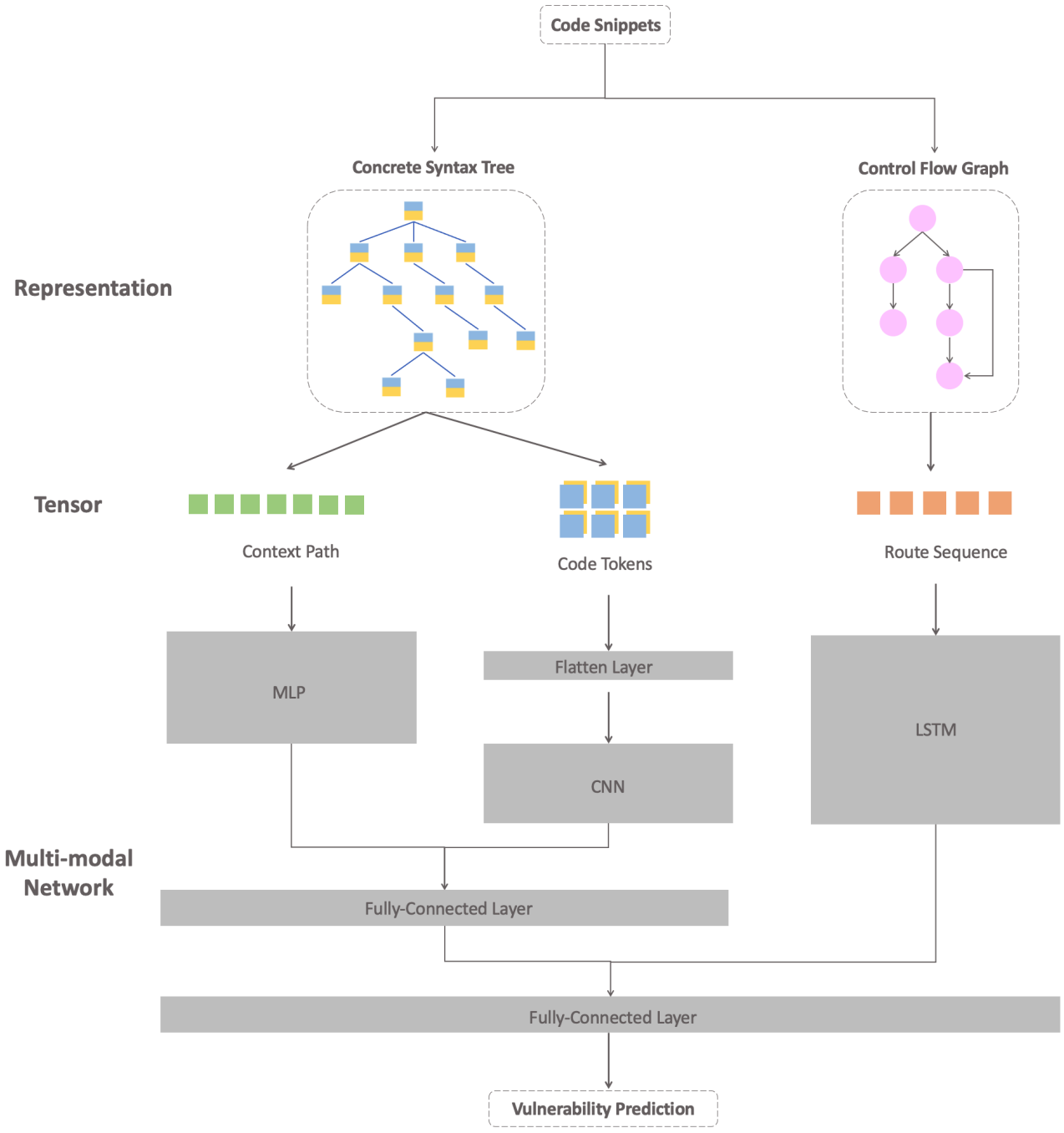


Fig. 8: Brief illustration of multi-modal learning.

D. Future Work 2: Code + Trace as Inputs

In addition to the function source code, IBM/D2A dataset also provides researchers with traces. This kind of bug report reveals more information about how vulnerabilities are detected. By utilizing the attention model, we can analyze the keywords in the trace messages and integrate them to our prediction model.

VII. Clear Description of Final Report Q&A

The final report is in the paper form above, here is a clear description of our work.

A. Synopsis

See the Abstract and the first paragraph of Sec. I.

B. Research Questions

1) To find a way of code embedding that can comprehensively represent the code snippets.
2) To use deep learning techniques to predict the vulnerability, given the function source code.
3) Comparison between Code2Tensor with respect to accuracy is in Sec. VI (our model achieves 54.70% accuracy).

C. Deliverables

Our open-source codes are available at https://github.com/LovelyBuggies/Code2Tensor (*including Python scripts, dependencies, README.md, video link, and milestone files*).

D. Self-Evaluation

This is a one-person project, so I do all the work alone. In this project, it is challenging to face difficulties independently. But I addressed it by constantly consulting Robin and referring to a lot of literature. By completing this project, I have a deeper understanding of software vulnerability, code embedding, and even tensor computing for different neural networks. I found that it is not that richer expressions or more complex neural networks are more helpful for detecting the bugs.

E. What I learned

During the experimentation, the most valuable thing I learned is that I keep trying to think of various code representations and deep learning methods to distinguish what operations are beneficial. Through this process, I needed to open the black box of code embedding and deep learning, which greatly enriched my knowledge base.

F. What Audience can Learn

You can learn vulnerability detection studies, code embedding skills, and their limitations in the related works in Sec. I. In addition, you can also know how to embed the code in Sec. II and how different deep learning networks work in Sec. III.

G. Course Advice

1) Presentation: In our course, students can freely choose speech topics according to their preferences. Although they have expanded their knowledge, their ideas may be too divergent. Rather than include all articles on computer software, perhaps a range could be selected and limited to finding articles in that range. In addition, presentations that link multiple works are highly encouraged.
2) Project: For different projects, it is difficult to have a unified indicator to measure and compare the results. Similarly, several topics can be specified for students to choose from.