Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 222 (2023) 35–44

Procedia Computer Science

www.elsevier.com/locate/procedia

International Neural Network Society Workshop on Deep Learning Innovations and Applications (INNS DLIA 2023)

A Novel Deep Multi-head Attentive Vulnerable Line Detector

Miles Q. Li[a], Benjamin C. M. Fung[a], Ashita Diwan[b]

[a] *School of Information Studies, McGill University, Montreal, Canada*
[b] *School of Computer Science, McGill University, Montreal, Canada*

Abstract

Detecting and fixing vulnerabilities in software programs before production is crucial in software engineering. Manual vulnerability detection is labor-intensive, especially for large programs, leading to the proposal of machine learning-based methods for automation. However, existing approaches primarily detect vulnerabilities at the function level, providing non-specific results that require additional developer effort to locate vulnerabilities. Detection at the line-of-code level is an underexplored area. In this paper, we propose a novel deep learning method for line-of-code vulnerability detection. Our hybrid neural network combines a memory network and multi-head attention mechanism. Through comprehensive experiments, we analyze the impact of each modification, demonstrating significant improvements in performance. Our approach outperforms existing methods for comparison, showcasing its effectiveness in vulnerability detection.

Keywords: Deep learning, vulnerability detection, memory networks, multi-head attention

1. Introduction

Software vulnerabilities are flaws in software that may cause security issues. They can be exploited by attackers to cause severe problems, such as system breakdowns, financial losses, and data breaches [16]. In 2019, a vulnerability was exploited by a hacker to gain access to more than 100 million Capital One customers' account information [1]. Vulnerabilities universally exist even in software written by experienced programmers. There are more than thousands of vulnerabilities in widely used software reported to the Common Vulnerabilities and Exposures (CVE) database[2]. To avoid the losses caused by vulnerabilities, it is important to identify vulnerabilities in source code and fix them at the development phase.

Compiler warnings and other rule-based methods are commonly used as the first step for vulnerability detection [21, 20]. The detection recall of these rule-based methods are limited by the variety of rules crafted by human

[*] Corresponding author: Benjamin C.M. Fung (ben.fung@mcgill.ca)
[1] https://www.consumidor.ftc.gov/blog/2019/07/ capital-one-data-breach-time-check-your-credit-report
[2] https://cve.mitre.org/

10.1016/j.procs.2023.08.142

exports. Code similarity-based methods [11, 17, 12] are also proposed to find vulnerabilities from code that is similar to code pieces that are known to be vulnerable. The detection recall of this kind of methods is also limited since it cannot find the same kinds of vulnerabilities in source code that are significantly different from the known vulnerable code in the database. And since this kind of methods is instance-based, when the database is large, the efficiency inevitably drops, yet a large database is required for good recall.

To achieve better generalizability, deep learning-based vulnerability detection methods have been proposed [16, 20]. They are expected to identify patterns that are correlated to the vulnerabilities that can be used to recognize vulnerabilities in unseen samples. Currently, most deep learning-based vulnerability detection methods work at the function level, i.e., they determine whether a function contains a certain vulnerability or not. This is not specific enough in practice because even if the programmers know that a function contains a certain vulnerability, to locate the lines that are relevant to it may still take a lot of time. Thus, a detection method that can tell how each line of code is related to a vulnerability is a more ideal solution.

To train those deep learning models, a large dataset of vulnerable code with labels of the types of vulnerabilities are required. In most publicly available datasets, there are at most tens of thousands positive vulnerable samples for each vulnerability in a certain programming language. This is enough to train traditional machine learning models and some small neural networks. For the datasets that provide the vulnerable line numbers, the number of labeled samples for each vulnerability is just about hundreds. This is too small for training deep learning models.

Sestili et al. [25] proposed s-bABI, a method to randomly generate vulnerable source code functions that resemble real-world source code to a large degree. It can generate unlimited number of pieces of source code with each line labelled. This provides an opportunity to train deep learning models to predict how each line is related to a vulnerability. They also proposed a memory network to determine whether a line of buffer write is vulnerable. The contributions of this paper are that we improved their preprocessing step and propose a more sophisticated deep learning model to achieve better vulnerability detection performance.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 defines the line-of-code level vulnerability detection task. Section 4 describe our the details of our vulnerability detection method. Section 5 presents the evaluation of the proposed method. Section 6 concludes the paper.

2. Related Works

In this section, we discuss different vulnerability detection methods and publicly available vulnerability datasets.

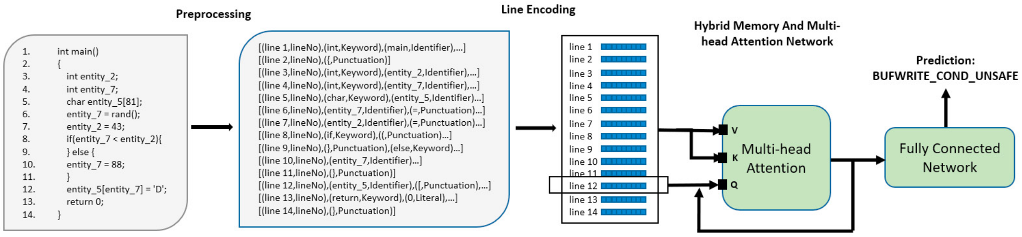
2.1. Rule-based Vulnerability Detection Methods

Rule-based methods rely on human experts to craft rules (a.k.a. templates) that correspond to certain types of vulnerabilities. If a piece of code matches a rule of a vulnerability, it is considered to be vulnerable. These kinds of methods yield either high false positive rate (when the rules are too strong) or high negative rate (when the rules are too weak) [16, 20]. The problem with this kind of methods is that it is hard to craft exhaustive and accurate rules for all vulnerability cases that appear in different forms. Frama-C, Cppcheck, Checkmarx, and Flawfinder[3], are some publicly available rule-based detection tools.

2.2. Deep Learning-based Vulnerability Detection Methods

As deep learning have been thriving as the state-of-the-art data-driven classification methods over the years, the attempts to apply them in vulnerability detection have also emerged. Deep learning-based vulnerability detection can be conducted on source code, intermediate representation, or assembly code. Methods applied on source code or intermediate representation can be used to identify and fix vulnerabilities at the development phase before the software programs are put in use. Those on assembly code are aimed at recognizing software programs developed by others to estimate potential risks.

[3] https://frama-c.com/, https://cppcheck.sourceforge.io/, https://www.checkmarx.com/, https://dwheeler.com/flawfinder/



```
1.      int main()
2.      {
3.        int entity_2;
4.        int entity_7;
5.        char entity_5[81];
6.        entity_7 = rand();
7.        entity_2 = 43;
8.        if(entity_7 < entity_2){
9.        } else {
10.       entity_7 = 88;
11.       }
12.       entity_5[entity_7] = 'D';
13.       return 0;
14.     }
```

```
[(line 1,lineNo),(int,Keyword),(main,Identifier),...]
[(line 2,lineNo),({,Punctuation)]
[(line 3,lineNo),(int,Keyword),(entity_2,Identifier),...]
[(line 4,lineNo),(int,Keyword),(entity_7,Identifier),...]
[(line 5,lineNo),(char,Keyword),(entity_5,Identifier)...]
[(line 6,lineNo),(entity_7,Identifier),(=,Punctuation)...]
[(line 7,lineNo),(entity_2,Identifier),(=,Punctuation)...]
[(line 8,lineNo),(if,Keyword),((,Punctuation)...]
[(line 9,lineNo),(},Punctuation),(else,Keyword)...]
[(line 10,lineNo),(entity_7,Identifier)...]
[(line 11,lineNo),(},Punctuation)]
[(line 12,lineNo),(entity_5,Identifier),([,Punctuation),...]
[(line 13,lineNo),(return,Keyword),(0,Literal),...]
[(line 14,lineNo),(},Punctuation)]
```

Figure 1: The workflow of our vulnerability detection model.

Commonly used features extracted on source code or intermediate representation include abstract syntax trees (ASTs) [23, 30, 22], control flow graphs [8, 30, 18], data dependency graphs [19, 30, 18], use of variables [8], bag-of-word opcodes [8], bag-of-word tokens [8], token sequences [19, 24, 30, 22, 18]. Commonly used features extracted on assembly code include library call N-grams [7], assembly code sequences [14]. Even though the proposed features have been reported good results, some features can only be used to identify some types of vulnerabilities. For example, bag-of-word tokens cannot be used alone to identify buffer overflow vulnerabilities, since the order of the tokens determines whether a piece of code is vulnerable and bag-of-word representation ignores the order of tokens. Some features (e.g., ASTs, control flow graphs, and data dependency graphs) contain graphical structure information, however, the graphs are not always fully used by graph neural networks, but flattened into vectors or sequences for the neural networks [23, 19, 22, 18].

As the most basic form of neural networks, fully connected networks (FCNs) have been applied to detecting vulnerabilities [7, 23]. The feature vectors have to have fixed dimension because FCNs cannot handle input of variable length. However, the lengths of source code of software programs are always variant, thus the features should also have variable length by nature. As a consequence, the modelling abilities of FCNs are limited for source code.

To overcome the aforementioned limitation of fully connected networks, convolutional neural networks (CNN) [14, 8, 24], recurrent neural networks (RNN) [24, 19, 22, 18], memory networks [5, 25], or the combination of different neural networks, such as CNN-LSTM [28], are applied on sequential features representing source code. To fully use the information of the structures of a function, graph neural networks, such as gated graph neural networks [30] are also applied on the graphs parsed from the source code.

2.3. Vulnerable Code Datasets

There are different vulnerable code datasets available to the public for data-driven vulnerability detection research. Common Weakness Enumeration (CWE) is a widely used categorization system for vulnerabilities. There are around 922 different types of vulnerabilities defined in the system. In most of the datasets, vulnerabilities are labeled as CWE entries.

The most popular vulnerable code dataset is called NIST Software Assurance Reference Dataset Project (SARD)[4]. It contains wild, synthetic, and academic vulnerable cases of different languages. Each vulnerability is labels with a CWE entry. The cases in SARD are combined to multiple Juliet Test Suites (JTS) for evaluating vulnerability detectors in different programming languages.

Draper VDISC Dataset [24] is another large dataset collected by for research purposes. It contains 1.27 million C/C++ functions that are vulnerable or not. The vulnerable ones belong to CWE-119, CWE-120, CWE-469, CWE-476, or any other vulnerabilities.

bABI [5] and s-bABI [25] are two buffer-overflow-vulnerable C/C++ program generators. The generated datasets are labeled at line-of-code to indicate their relations to the vulnerability. With the generators, unlimited number of cases can be generated. s-bABI is the improved version of bABI to generate syntactically valid C with non-trivial control flows. The code generated by s-bABI can be compiled and contains loop and condition statements so that it is closer to real life source code.

[4] https://samate.nist.gov/SARD/

3. Problem Formulation

The vulnerability detection problem we handle is at the line-of-source-code level, rather than function level. Our setting is better than the function level vulnerability detection because the methods aimed for this task can save developers' time in locating the vulnerabilities.

The code of a C/C++ function is a sequence of lines:

$$func = \langle line_1, line_2, ..., line_n \rangle \tag{1}$$

The sequence of lines also correspond to a sequence of labels:

$$l = \langle label_1, label_2, ..., label_n \rangle \tag{2}$$

Each label represents the relation of a line to the target vulnerability.

Each line of code is a sequence of tokens:

$$line = \langle token_1, token_2, ... \rangle \tag{3}$$

Definition 1 (Vulnerability Detection)**.** Consider a collection of code of functions F and a collection of sequences of labels L that show the relations of the lines of the code of the functions in F to a certain vulnerability. Let $func$ be an unknown function that $func \notin F$. The *vulnerability detection* problem is to build a classification model M based on F and L such that M can be used to determine the labels of lines of $func$. ■

4. Methodology

The workflow of our vulnerability detection model is shown in Figure 1. The first step is to preprocess the source code of a C/C++ function. It generates the tokens and their types in the source code. Then, based on the tokens and their types, we encode a line of code to a vector representation. The hybrid memory and multi-head attention network uses the vectors representing the query line and all the lines of the function as the context to determine the label of the query line.

4.1. Preprocessing

We perform the following preprocessing steps for each function:

1. We use Clang tokenizer to tokenize the function to get the tokens and their types. We use the (token, type) tuple to represent each token in the source code.
2. We prepend a token representing the line number of each line to its beginning. This could provide positional information of each line to the neural network.
3. We represent each line as a sequence of token and type tuples.

This preprocessing procedure is improved from [25]. The difference is that they use a token itself to represent it and we also use its type. This is because the text of tokens such as identifiers is not relevant to the semantic of the function, so that it would not be relevant to a vulnerability. The types of these tokens can indicate the functions of the tokens, so that they provide more information to the vulnerability detection model. However, the text of the identifier still needs to be kept, because we need it to differentiate tokens of the same type.

4.2. Neural Network Architecture

The proposed neural network is aimed at predicting the label of a query line of code given its context of the whole function. By applying the network on every line of a function, we can get the entire set of labels for the function. The network is organized in a hierarchical manner. The lower part encodes each line of code to a vector. The higher part uses the vector representations of the query line and other lines of the function to compute the label of the query line.

4.3. Line Encoding

After preprocessing, each line is represented as $line = \langle (tok_1, typ_1), (tok_2, typ_2), \dots \rangle$. We first encode a token and a type to a vector. Let n_1 and n_2 be the number of different tokens and types of tokens respectively. We create a token embedding matrix and a token type embedding matrix of the shape $E_1 \in \mathbb{R}^{n_1 \times d_{emb1}}$ and $E_2 \in \mathbb{R}^{n_2 \times d_{emb2}}$. Each row of the matrices is the vector representation of a token and a token type respectively. Thus, for each token and token type, we have an embedding.

We also use the positional encoding (PE) proposed by Sukhbaatar et al. [26] to introduce position information for each token:

$$PE[i]_j = (1 - \frac{i}{I} - (\frac{j}{d_{emb}})(1 - \frac{2i}{I})) \tag{4}$$

where $PE[i]_j$ is the j-th element of the positional encoding of the i-th token, and d_{emb} can be d_{emb1} or d_{emb2}. The positional encoding is applied to both the token embeddings and the type embeddings by element-wise product to form the vector representations of them. We concatenate the vector of the i-th token and the vector of its type to be its final vector representation:

$$v_i = Concat(E_1[tok_i] \cdot PE[i], E_2[typ_i] \cdot PE[i]) \tag{5}$$

Thus, each line is represented as a sequence of vectors: $\langle v_1, v_2, \dots \rangle$. Then, we use the summation of them to be the vector representation of the line: $v_{line} = \sum_i v_i$.

4.4. Hybrid Memory And Multi-head Attention Network

Whether a line of code is vulnerable is related to other lines of the function. For example, whether the statement of line 12: $entity\_5[entity\_7] =' 0';$ in Figure 1 causes buffer overflow depends on the line that define the length of the array $entity\_5$, the line that defines $entity\_7$, and whether $entity\_7$ has been ensured to be smaller than the length of $entity\_5$. This means that the lines of code are related with each other in different ways. The vanilla attention mechanism [6, 2, 26] was proposed to allow interactions between different items. The disadvantage of vanilla attention mechanism is that it only compares the vectors of the items as themselves, so that it is limited in considering the different kinds of relations between them. Multi-head attention [27] is proposed to allow different items to interact with each other from multiple aspects by mapping the vectors of them to different spaces and interact in these spaces. Memory network is a kind of network architecture that allows a query item to interact with contextual items in multiple layers [26]. The attention mechanism used in it is the vanilla attention mechanism. We replaced it with multi-head attention, so that we get a hybrid memory and multi-head attention network.

The input to a multi-head attention network includes three matrices: $Q \in \mathbb{R}^{m \times d_Q}$, $K \in \mathbb{R}^{n \times d_K}$, and $V \in \mathbb{R}^{n \times d_V}$. Q represents m query vectors; K represents n key vectors; V represents n value vectors. The columns of K and V correspond to the same n objects. The computation of the multi-head attention network is as follows:

$$MultiHead(Q, K, V) = W^O Concat(head_1, \dots, head_h) \tag{6}$$

$$where\ head_i = softmax(\frac{(W_i^Q Q)(W_i^K K)}{\sqrt{d_k}})(W_i^V V) \tag{7}$$

where $W_i^Q \in \mathbb{R}^{d_k \times d_Q}$, $W_i^K \in \mathbb{R}^{d_k \times d_K}$, $W_i^V \in \mathbb{R}^{d_k \times d_V}$, and $W_i^O \in \mathbb{R}^{d_Q \times hd_V}$.

Following the memory network framework, we apply multiple layers of multi-head attention networks. In the bottom layer multi-head attention, the vector representation of the query line is used in the query matrix $Q_0 = [v_q] \in$

$\mathbb{R}^{1\times(d_{emb1}+d_{emb2})}$. The vectors representing all the lines of the function $V_{lines} = [v_{line_1}, ..., v_{line_n}] \in \mathbb{R}^{n\times(d_{emb1}+d_{emb2})}$ make up both K and V. Thus, we have $d_Q = d_K = d_V = d_{emb1} + d_{emb2}$. For the $i+1$ layer of multi-head attention, the K and V stay the same, and Q_i is the output of the previous layer. The complete computation is as follows:

$$Q'_{i+1} = MultiHead(Q_i, V_{lines}, V_{lines}),\ 0 \le i < l_1 \tag{8}$$

$$Q_{i+1} = Q_i + LayerNorm(W_i Q'_{i+1} + b_i) \tag{9}$$

where $W_i \in \mathbb{R}^{(d_{emb1}+d_{emb2})\times(d_{emb1}+d_{emb2})}$,$b_i \in \mathbb{R}^{d_{emb1}+d_{emb2}}$.It should be noted that this is different from a multi-layer self-attention architecture where $Q = K = V$, and all three matrices are updated at every layer.

The output of the top layer Q_{l_1} contains the representation of the queried line in the context of the function. It is fed to a fully connected network to compute the label of the line as follows:

$$\boldsymbol{v}_{l_2} = FC^{l_2}(...FC^1(\boldsymbol{Q}_{l_1})...) \tag{10}$$

$$\boldsymbol{y} = softmax(W\boldsymbol{v}_{l_2} + \boldsymbol{b}) \tag{11}$$

where $softmax(z) = \frac{1}{\sum_{j=1}^{c} e^{z_j}}(e^{z_1}, ..., e^{z_c}), \boldsymbol{b} \in \mathbb{R}^c$. c is the number of different labels.

The key differences from [25] are as follows:

1. We include the types of tokens in encoding the tokens to capture the similarities of different tokens in the same type.
2. We use the same token embedding matrix and type embedding matrix to encode tokens and types to form Q, K, and V while [25] use one embedding matrix to encode tokens for Q and K and another embedding matrix to encode tokens for V.
3. We use multi-head attention instead of the vanilla attention mechanism to allow lines of code to interact with each other from different aspects.
4. We replaced BatchNorm [10] with LayerNorm [1] in the hybrid memory and multi-head attention network since the latter works better with attention-based models [29].

5. Experiments

In this section, we describe the evaluation of the proposed vulnerability detection method in terms of its performance.

5.1. Dataset

The dataset we use to evaluate the proposed method is generated with s-bABI [25]. It generates unlimited number of buffer overflow vulnerable C/C++ programs. There are five different labels for the lines:

1. BUFWRITE_COND_SAFE: the buffer write is safe since the index used in the buffer write is checked in a control flow
2. BUFWRITE_COND_UNSAFE: the buffer write is unsafe because there is no appropriate index safety check using a control flow
3. BUFWRITE_TAUT_SAFE: the buffer write is safe without consulting a control flow
4. BUFWRITE_TAUT_UNSAFE: the buffer write is unsafe without consulting a control flow
5. Body/Other: the line does not have a buffer write

Table 1: Hyper-parameters of our vulnearbility detection model.

Hyper-parameter	Description	Candidate Values
d_{emb1}	Dimension of token embeddings	128,64,32
d_{emb2}	Dimension of type embeddings	128,64,32
d_k	Dimension of mapped space in multi-head attention	Same as $d_{emb1} + d_{emb2}$
h	Number of heads in multi-head attention	2,4,8
l_1	Number of multi-head attention layers	2,3,4
l_2	Number of fully connected layers	2,3,4
d_f	Dimension of fully connected layers	Same as $d_{emb1} + d_{emb2}$

For the evaluation, only buffer write lines are used as query lines in the datasets because it is trivial to recognize lines without buffer writes and label them as Body/Other. We generate 1 million C/C++ files and split them for the training and validation sets, and another 100 thousand C/C++ files for the test set. They are generated with different random seeds (i.e., 0 and 1 respectively). In each file, there could be multiple lines of buffer writes. This results in 1,600,326 samples in the training set, 400,073 samples in the validation set, and 199,659 samples in the test set. Each sample is a query line of buffer write paired with the lines of the function as the context.

5.2. Evaluation Metric

We evaluate our vulnerability detection model in both fine grained and coarse grained settings. In the fine grained setting, the result for a sample is correct only if the predicted label is the exact gold standard label. We use accuracy as the metric for this setting. In the coarse grained setting, the four labels collapse to two labels: SAFE and UNSAFE. As long as the predicted label and the gold standard label are both SAFE(negative) or UNSAFE (positive), the result is considered to be correct. So, the metrics for binary classification problems, i.e., precision, recall, F1, and accuracy are all used in this setting.

5.3. Models for Comparison

We compare our vulnerability detection method with some static analysis tools and other deep learning-based vulnerability detection models. The static analysis tools include Clang, Cppcheck, and Frama-c. They are only evaluated at the coarse grained level since they cannot distinguish between different situations of safe or unsafe.

The other deep learning-based vulnerability detection methods:

1. **Memory Network**: The model proposed by Sestili et al. [25]. We summarized the differences between our model and their model in the previous section.
2. **Long Short-Term Memory (LSTM) Network**: LSTM [9] is a kind of RNN model. The lines are encoded in the same way as our models. We then form a sequence with the vector representations of all the lines and the query line and apply multiple layers of LSTM on the sequence. Then, we feed the output of the top layer LSTM at the query line to an FCN to determine the label of the query line.
3. **Gated recurrent units (GRUs) Network**: GRU [4] is a another kind of RNN models, which has fewer parameters than LSTM. It is used the same way as LSTM.
4. **Transformer Network**: Transformer [27] was proposed to improve the performance of RNN models. The lines are encoded in the same way as our model. We then form a sequence with the vector representations of all the lines and the query line and apply the encoder of Transformer on the sequence. Then, we feed the output of the layer at the query line to an FCN to determine the label of the query line.
5. **Fully Connected Network (FCN)**: We concatenate the vector representations of all the lines and the query line to form a vector. All samples are padded to the length of the longest function, so that the vectors of all functions have fixed dimension. Then, we apply a multi-layer FCN on the vector to compute the label of the query line.
6. **Hierarchical LSTM Network**: We use LSTM networks in a hierarchical manner. We apply multiple layers of LSTM on the vector sequence of tokens of each line, and use the output at the last token as the vector representa-

Table 2: Fine grained experiment results of all models.

Model	Accuracy	p-value
Memory Network	93.3%	3.4e-10
FCN	77.9%	3.2e-15
LSTM	79.6%	2.4e-15
GRU	79.6%	2.3e-15
Transformer	96.8%	3.1e-7
Hierarchical LSTM	79.5%	1.2e-15
Hierarchical GRU	79.5%	3.2e-15
Hierarchical Transformer	53.6%	2.1e-18
Our method	**98.8**%	N/A

Table 3: Coarse grained experiment results of all models.

Model	Precision	Recall	F1	p-value	Accuracy	p-value
Clang	99.9%	57.6%	73.1%	1.1e-17	75.0%	2.0e-17
Cppcheck	96.7%	69.2%	80.6%	1.7e-16	80.4%	1.5e-16
Frama-c	100%	74.3%	85.3%	1.8e-15	84.9%	1.4e-15
Memory Network	95.0%	93.5%	94.2%	2.2e-10	93.4%	3.9e-11
FCN	85.6%	86.9%	86.2%	2.4e-14	83.5%	7.3e-15
LSTM	88.6%	85.9%	87.2%	2.3e-13	85.2%	3.2e-15
GRU	87.3%	87.5%	87.4%	6.2e-14	85.1%	3.7e-14
Transformer	97.5%	97.0%	97.3%	6.5e-7	96.8%	1.0e-7
Hierarchical LSTM	85.6%	89.8%	87.7%	7.0e-14	85.1%	3.2e-14
Hierarchical GRU	87.0%	87.9%	87.5%	1.9e-14	85.1%	7.2e-14
Hierarchical Transformer	60.3%	84.4%	71.1%	1.3e-17	58.5%	1.8e-18
Our method	98.8%	99.1%	**99.0**%	N/A	**98.8**%	N/A

tion of the line. Then, we apply multiple layers of LSTM the same way on the sequence of vectors representing the lines to compute the label of the query line.

7. **Hierarchical GRU Network**: We use GRU in the same way as we use LSTM in the hierarchical LSTM network.
8. **Hierarchical Transformer Network**: One Transformer is applied on the vectors of the tokens of each line to compute a vector representation of the line as the output of the first token. We then form a sequence with the vector representations of all the lines and the query line and apply the encoder of Transformer on this sequence. Then, we feed the output of the top layer of the Transformer at the query line to an FCN to determine the label of the query line.

5.4. Model Training

The objective function to train the models is the cross entropy loss of the predicted labels and the gold standard labels. We use Adam [13] with the initial learning rate $1e-4$. Early stopping on the validation set is applied to avoid overfitting.

The hyper-parameters of our complete model and the candidate values are summarized in Table 1. We use grid search to tune hyper-parameters of all models with the validation set.

5.5. Results

We repeat each groups of experiments 5 times with random seeds from 0 to 4 and report the average accuracy in Table 2. We use student t-test to indicate statistically significant difference in accuracy between other models and our model. As is shown, memory network, Transformer network, and our model achieve decent results ($>$ 90% accuracy). Our model outperforms them as well with statistically significant difference in both fine grained and coarse grained

Table 4: Fine grained experiment results for ablation study.

Model	Accuracy	p-value
Use Batch Norm	98.1%	5.3e-7
Use No Norm	98.7%	0.35
Without Type Embeddings	**98.8%**	0.76
Without Multi-head Attention	96.5%	3.8e-11
Different Embeddings for Q/K/V	98.2%	9.0e-7
Complete model	**98.8%**	N/A

evaluation settings. Transformer network and our model outperform memory network by a large margin, and this shows the effectiveness of incorporating multi-head attention to capture the relations between different lines of code. The fact that our model significantly outperforms the Transformer network also shows the superiority of applying the multi-head attention in the memory network framework than applying it from the Transformer network.

As can also be seen that FCN, LSTM and GRU achieve similarly worse results. This is expected since they have inferior modelling ability than the more advanced network architectures. We also observe that hierarchical networks does not improve the performance of their non-hierarchical counterparts. For the Transformer, the hierarchical version achieves even worse performance. This may because that the hierarchical Transformer architecture is too complex so that the training signal only from the top layer is not enough. Previous studies have shown that complex hierarchical Transformer networks may need to be trained separately at different levels [15].

The static analysis tools achieve overall worse performance than deep learning-based vulnerability detection methods. They achieve high precision but low recall. This means the rules they use are accurate, but are not exhaustive so that they cannot cover the variety of buffer overflow cases.

5.6. Ablation Study

To separate the effects of different improvements of our proposed model on the final result, we also show the classification performance when we modify each of them as the ablation study. Table 4 shows the results.

We can see that whether multi-head attention is applied causes the largest difference in the classification performance. The multi-head attention module is also a core component of our complete model. It can also be seen that without type embeddings, the performance is not significantly changed. And we find that the best result without type embeddings is achieved when the dimension of token embeddings is 64 and with type embeddings, it is achieved when the dimension of both token and type embeddings is 32. That means both settings achieve the best performance when the sizes of the two neural networks are the same. We think the reason that the type embeddings do not improve the performance of the network is that the type information could already be learned in token embeddings as the dataset is large enough. This is also the reason why POS tag embeddings are not used in state-of-the-art neural networks for NLP [27, 3]. Additionally, we are surprised to find that with batch norm, the performance drops significantly. Even though without any norm, the performance is not significantly affected, the layer norm [1] makes the training converge faster than without any norm by more than 20 epochs.

6. Conclusion

In this paper, we propose a hybrid memory and multi-head attention network for vulnerability detection at the line-of-code level. It uses the types of tokens to capture the similarities of different tokens of the same type. The multi-head attention mechanism computes the relations between the query line and other lines of the function from multiple aspects. Experiment results show that the proposed method outperforms all models for comparison.

Acknowledgements

This research was funded by NSERC Discovery Grants (RGPIN-2018-03872) and Canada Research Chairs Program (950-230623).

References

[1] Ba, J.L., Kiros, J.R., Hinton, G.E., 2016. Layer normalization. arXiv preprint arXiv:1607.06450 .

[2] Bahdanau, D., Cho, K., Bengio, Y., 2014. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 .

[3] Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al., 2020. Language models are few-shot learners. arXiv preprint arXiv:2005.14165 .

[4] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078 .

[5] Choi, M.j., Jeong, S., Oh, H., Choo, J., 2017. End-to-end prediction of buffer overruns from raw source code via neural memory networks. arXiv preprint arXiv:1703.02458 .

[6] Graves, A., 2013. Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850 .

[7] Grieco, G., Grinblat, G.L., Uzal, L., Rawat, S., Feist, J., Mounier, L., 2016. Toward large-scale vulnerability discovery using machine learning, in: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, pp. 85–96.

[8] Harer, J.A., Kim, L.Y., Russell, R.L., Ozdemir, O., Kosta, L.R., Rangamani, A., Hamilton, L.H., Centeno, G.I., Key, J.R., Ellingwood, P.M., et al., 2018. Automated software vulnerability detection with machine learning. arXiv preprint arXiv:1803.04497 .

[9] Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural computation 9, 1735–1780.

[10] Ioffe, S., Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift, in: International conference on machine learning, PMLR. pp. 448–456.

[11] Jang, J., Agrawal, A., Brumley, D., 2012. Redebug: finding unpatched code clones in entire os distributions, in: 2012 IEEE Symposium on Security and Privacy, IEEE. pp. 48–62.

[12] Kim, S., Woo, S., Lee, H., Oh, H., 2017. Vuddy: A scalable approach for vulnerable code clone discovery, in: 2017 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 595–614.

[13] Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 .

[14] Lee, Y.J., Choi, S.H., Kim, C., Lim, S.H., Park, K.W., 2017. Learning binary code with deep learning to detect software weakness, in: KSII the 9th international conference on internet (ICONI) 2017 symposium.

[15] Li, M.Q., Fung, B.C.M., Charland, P., Ding, S.H.H., 2021a. I-MAD: Interpretable malware detector using Galaxy Transformers. Computers & Security (COSE) 108, 1–15.

[16] Li, Z., Zou, D., Tang, J., Zhang, Z., Sun, M., Jin, H., 2019. A comparative study of deep learning-based vulnerability detection system. IEEE Access 7, 103184–103197.

[17] Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., Hu, J., 2016. Vulpecker: an automated vulnerability detection system based on code similarity analysis, in: Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 201–213.

[18] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021b. Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing .

[19] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681 .

[20] Lin, G., Wen, S., Han, Q.L., Zhang, J., Xiang, Y., 2020. Software vulnerability detection using deep neural networks: A survey. Proceedings of the IEEE 108, 1825–1848.

[21] Lin, G., Xiao, W., Zhang, L.Y., Gao, S., Tai, Y., Zhang, J., 2021. Deep neural-based vulnerability discovery demystified: data, model and performance. Neural Computing and Applications , 1–14.

[22] Lin, G., Zhang, J., Luo, W., Pan, L., De Vel, O., Montague, P., Xiang, Y., 2019. Software vulnerability discovery via learning multi-domain knowledge bases. IEEE Transactions on Dependable and Secure Computing .

[23] Peng, H., Mou, L., Li, G., Liu, Y., Zhang, L., Jin, Z., 2015. Building program vector representations for deep learning, in: International conference on knowledge science, engineering and management, Springer. pp. 547–553.

[24] Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M., 2018. Automated vulnerability detection in source code using deep representation learning, in: 2018 17th IEEE international conference on machine learning and applications (ICMLA), IEEE. pp. 757–762.

[25] Sestili, C.D., Snavely, W.S., VanHoudnos, N.M., 2018. Towards security defect prediction with ai. arXiv preprint arXiv:1808.09897 .

[26] Sukhbaatar, S., Szlam, A., Weston, J., Fergus, R., 2015. End-to-end memory networks. arXiv preprint arXiv:1503.08895 .

[27] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need, in: Advances in neural information processing systems, pp. 5998–6008.

[28] Wu, F., Wang, J., Liu, J., Wang, W., 2017. Vulnerability detection with deep learning, in: 2017 3rd IEEE international conference on computer and communications (ICCC), IEEE. pp. 1298–1302.

[29] Xu, J., Sun, X., Zhang, Z., Zhao, G., Lin, J., 2019. Understanding and improving layer normalization. arXiv preprint arXiv:1911.07013 .

[30] Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. arXiv preprint arXiv:1909.03496 .