EAI Endorsed Transactions
on Security and Safety

Research Article **EAI.EU**

Identify Vulnerability Fix Commits Automatically Using Hierarchical Attention Network

Mingxin Sun[1], Wenjie Wang[1], Hantao Feng[2], Hongu Sun[2], Yuqing Zhang[1 2,*]

[1]National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, China
[2]School of Cyber Engineering, Xidian University, China

Abstract

The application of machine learning and deep learning in the field of vulnerability detection is a hot topic in security research, but currently it faces the problem of lack of dataset. Considering vulnerable code can be obtained from vulnerability fix commits, we propose an automatic vulnerability commit identification tool based on hierarchical attention network (HAN) to expand existing vulnerability dataset. HAN can model the input data at the word and sentence levels respectively and pay attention to the changes in the characteristics of different words in different categories, which improves the classification performance. Experimental results show that the accuracy and F1 of our model both achieve 92%. Through the vulnerability fix commit, researchers can quickly locate the vulnerable code. And extracting vulnerable code from open-source software can effectively expand the current dataset due to the enormous number of open-source software.

Received on 14 April 2020; accepted on 05 May 2020; published on 12 May 2020

Keywords: vulnerability detection, GitHub Commits, deep learning, vulnerability patch

doi:10.4108/eai.13-7-2018.164552

1. Introduction

To improve the efficiency of software development, most developers use open-source software in their system or some open source components in their software. However, as open-source code has becoming more and more common in commercial software products and some internal applications of institutions, attacks against vulnerabilities in the open-source software are increasing rapidly. Traditional vulnerability detection methods, such as static analysis and dynamic analysis, mostly require a large amount of professional security researches involvement. The detection efficiency mainly depends on the researcher's expertise.

As a result of that, lots of researchers are beginning to pay more attention to improve the efficiency of vulnerability detection. An important research topic is to apply machine learning to source code based vulnerability detection. Li et al.[1] use deep learning to train a classification predictive model for vulnerability detection. They use a bidirectional LSTM network on the SARD data set. But the code structure in this dataset is too simple and has a big gap with real vulnerable code, which makes the model not perform well on real programs. After that, Zhou et al.[2] proposed a vulnerability detection method using a graph neural network on real vulnerability dataset which costs them 600 person-hours, and the accuracy of their method achieves 72.26%.

The data used in the above studies either has a large gap with real vulnerabilities or takes a lot of time to collect. Moreover, there is no benchmark dataset in the area of vulnerability detection. When some researchers build their own dataset, they need to manually verify the data one by one to ensure the accuracy of vulnerable code extracted, which is challenging to automate. The enormous workload and low efficiency of manual collection caused by the diversity of vulnerabilities make it difficult for others to reproduce existing work or conduct further research.

It's not difficult to see that the biggest challenge in the application of machine learning on vulnerability detection is the lack of dataset. An efficient method for collecting vulnerability data that can reflect the real code structure is the key to this problem. Only when the amount of data that reflects the

*Corresponding author. Email: zhangyq@ucas.ac.cn actual vulnerabilities is large enough can machine learning show its superiority and then promote the realization of efficient and automatic detection of vulnerabilities. After the preliminary research, we found that the commits for vulnerability fixing can help us locate vulnerable code more efficiently. We can quickly locate the vulnerable code based on the modified code in commits. And the significant number of commits in open-source software makes it easier to collect. Therefore, we construct a system to identify vulnerability fix commit automatically. Our experiments on the collected commit data show that the vulnerable function can be positioned in the range of one or two functions when collecting vulnerability code via commit, thereby significantly improving the efficiency of extracting the vulnerability code.

In short, we chose the commit as the entry point based on the following facts: First, in most open source projects, each commit usually addresses only one issue, especially bug fix. Second, vulnerability fix commits are significantly different from other commits, such as new feature commits. The vulnerability fix commits usually modify fewer files and fewer lines than other commits. Third, analyzing the modified locations in commits can shorten the time cost in locating vulnerable code. Fourth, the information of source code before and after modifications are well included in commits, so we can easily access the fixed vulnerability-free code, furthermore, improve the quality of vulnerability dataset.

Based on the above analysis and the composition of a commit, each has a brief description that contains at least one sentence in natural language summarizing the purpose of this commit. We choose the hierarchical attention mechanism that can characterize the data from word and sentence level to identify the vulnerability fix automatically. HAN has two layers, one for word-level, another for sentence-level. Each layer has an encoder with attention mechanism. With the attention mechanism, it can assign different weights to each word in the sentence, and then integrate each word and its weight to form a sentence vector, generate the document vector in the same way as sentence vector. Finally, it passes the document vector into a fully connected layer with *sigmoid* to get the class probability distribution of the document. To our knowledge, we were the first to use it for vulnerability fix commits classification and prediction. And the experiment achieved a good result with an accuracy of 92.81%, F1 of 92.71%. All the commits data are collected from open-source projects in C/C++ programming language on GitHub[1], the world's biggest hosting platform for open-source projects. Besides, we cross-compares the descriptions of vulnerabilities in the National Vulnerability Database(NVD)[2] and Common Vulnerabilities and Exposure(CVE)[3] with the commits log of open-source projects, to determine which commits are about vulnerability fix. We use the commit message as the feature of a commit, and utilize the hierarchical attention network as our vulnerability fix commits prediction model. We use the method in this paper to construct our own dataset of vulnerable function code about CWE-119 (Buffer Error) and CWE-399 (Resource Management Error), which cost our 100 person-hours.

[1]https://github.com/

In Summary, the major contributions of this paper are as follows:

- An approach of recognizing vulnerability fix commits based on a hierarchical attention network, which achieves the accuracy of 92.81%, F1 of 92.72%.
- A vulnerability fix commit dataset that reflects the real environment opens to other researchers, which contains commit hash, commit message, the number of changed files, the number of insertion code lines, and the number of deletion code lines. Researchers can build their own vulnerability dataset efficiently based on our commit dataset.

2. Approach

In this section, we describe the detail of our vulnerability fix commit recognition system.

First, we need to collect enough vulnerability data from open source projects. Considering the authority and accuracy of the vulnerability information, we choose NVD and CVE as our data source. The NVD and CVE's databases record a lot of information about vulnerabilities, such as description, vulnerability type, reference hyperlink, release time, etc. Each vulnerability has a unique identifier named CVE ID. In the beginning, we crawled all the information, indexed them by CVE ID, traversed to remove the duplicate records. Then we filter out the vulnerabilities unrelated to open-source software. Only the software with source code located on GitHub was retained because our research is mainly focused on GitHub commits.

In this way, we initially established a vulnerability database for open-source software. However, this is just a vulnerability descriptive database, which only records descriptive information about the vulnerabilities and contains no trigger code or fix code. Then, we extract the vulnerability fix information from all the

[2]https://nvd.nist.gov/
[3]https://cve.mitre.org/

descriptive information and access the commit content in the corresponding GitHub repository, which requires cloning the entire GitHub repository and extracting all commits in the project. Except for the vulnerability fix commits, other commits serve as negative samples of the commit data set. Next, we extract features to characterize each commit.

As for the classification predictive model, we use a hierarchical attention network to learn the characteristics of the vulnerability fix commit.

The detailed process steps are as follows.

2.1. Data Collection

After the vulnerability is disclosed, developers tend to fix the vulnerability as much as possible and commit the repair code to the project repository. After that, most bug reports will update the information about these fixes or patches. In these data, GitHub patches account for the largest proportion, and the patches on GitHub are all in diff format, which is convenient for direct text analysis. Therefore, we want to collect as many of GitHub patches as possible for experimental input data.

To gather the patch, we need to identify the commit where the patch is located. To collect the vulnerability fix commits, first, we crawled all the descriptive vulnerability data in NVD and CVE's database. Second, we deduplicated the vulnerability information according to the CVE id and dropped the withdrawn vulnerabilities. Third, we classified the vulnerability data based on whether the reference hyperlink contains the keywords of "github.com". It is not difficult to see that the vulnerability with GitHub hyperlink is a vulnerability of the open-source software whose source code is hosted on GitHub. In this way, we got 3,423 open-source software vulnerabilities from 126,185 vulnerabilities.

Among the vulnerability descriptive information, we put our mind to the external reference hyperlinks marked as "PATCH" by NVD. And keep only links whose domain is hosted on GitHub and contains the keyword "commit". Then we normalize the URL string by removing unnecessary information and only retain the information required in the form of *"https://www.github.com/vendor_name/product_name/commit/SHA-1"*. After doing this, we can easily get the software name, the vendor name, and the hash code of this commit by splitting the normalized URL with "/". The SHA-1 hash code is a cryptographic string that is generated based on the information contained in the commit. It is a commit's id and can uniquely represent a commit object. So, we can request the content of a commit using the command *"git show SHA-1"* in a repository. Here the content of a commit is a *patch* output, the difference introduced by the commit.

After the above processing, we finally collected 3,704 GitHub commits from 933 open-source projects. Among them, Linux contributed the most significant number of commit data, accounting for 15% of the total amount. 2% of these 933 open source projects contributed 41.7% of commits.

We randomly selected 200 commits from all the 3,704 commits for manual review to verify whether these commits are real vulnerability fixes. The review shows that they are all vulnerability fixes. Therefore, we believe that the commits data obtained by the above method is reliable enough to be directly used as positive samples. We identify the programming language based on the suffix of the file modified in the patch. According to our statistics, the number of patches that modify code files programming in C/C++ is 2,014, accounting for 48% of total patches, which is the largest. The second most is PHP (18%), and then is JavaScript (5%). In this paper, we focus on the open-source software programming in C/C++ because we want to extract vulnerable code based on these patches in the next work and C/C++ has the largest amount of data, and there are several open-source parsing tools for C/C++, such as Clang[4].

Negative samples are collected in the following way. First, run "git log –stat" command in the GitHub repository directory to get all the commit records and abbreviated stats for each commit, including commit hash, commit message, the number of files modified, the number of deleted lines, and the number of inserted lines and so on.

Except for the positive samples, the rest are candidates for negative samples. Then we apply regular expression shown in the *table 1* to the commit message of the candidate negative sample to confirm that there is no commit obviously related to a vulnerability fix. After that, a total of 4,774,882 commits were collected, of which 2,925 were positive samples, that is, vulnerability fix commits.

Table 1. The regular expression used to check the candidate negative commits

Regular Expression Statement
denial of service, dos, ReDos, DDos, remote code execution, open.*?redirect, XSS, XXE, XSRF, CSRF, SQL injection, CVE, NVD, malicious, attack, exploit, directory.*traversal, clickjack, hijack, advisory, insecure, security, unauthorize, attacker

[4]https://clang.llvm.org/

2.2. Feature selection

The log of commits contains information including commit id, date, author, commit message, the number of modified files, the number of deleted lines, the number of inserted lines. We finally choose commit message as the feature for a commit. A commit message consisted of two parts, title and thorough description. The title is the text up to the first blank line in the message. It is a single line less than 50 characters, which summarizes the changes. The thorough description is optional. It details the content and purpose of this commit. Here we use the whole message as the feature, title, and thorough description (if any).

Lines 5-17 of Figure 1 is an example of commit message. The message is written in natural language, and it briefly describes the purpose of this commit. The result would not be affected by the type of vulnerability if using the commit message as the feature.

```
1   From a206b0ea12eb4606b93323268fc81a4f1f952531 Mon Sep 17 00:00:00 2001
2   From: Pieter Wuille <pieter.wuille@gmail.com>
3   Date: Fri, 17 Feb 2012 17:58:02 +0100
4
5   Do not allow overwriting unspent transactions (BIP 30)
6
7   Introduce the following network rule:
8    * a block is not valid if it contains a transaction whose hash
9      already exists in the block chain, unless all that transaction's
10     outputs were already spent before said block.
11
12  Warning: this is effectively a network rule change, with potential
13  risk for forking the block chain. Leaving this unfixed carries the
14  same risk however, for attackers that can cause a reorganisation
15  in part of the network.
16
17  Thanks to Russell O'Connor and Ben Reeves.
18  ---
19   src/main.cpp | 26 ++++++++++++++++++++++++--
20   1 file changed, 24 insertions(+), 2 deletions(-)
21
22  diff --git a/src/main.cpp b/src/main.cpp
23  index 995195289f8c..20aa069a7931 100644
24  --- a/src/main.cpp
25  +++ b/src/main.cpp
26  ...
```

Figure 1. An example of commit message

2.3. Data Processing

Each commit message has a corresponding label of "0" or "1", where "1" means it is a description for vulnerability fix commit, 0 means not. We randomly select a certain number of negative samples to address the problem of a serious imbalance between positive and negative samples. After a few comparative tests, we decide to use the dataset with a positive and negative ratio of 1:7 and use random oversampling for data enhancement. Finally, we have 20,584 samples used in the model learning.

Considering the structure of HAN, we have performed tokenization on sentence-level and word-level. There's no way to split the message text directly because of the title line without end punctuation. So, we first split the title and the detailed description, then utilize NLTK's[3] sentence tokenize API on each section and merge them.

Next, we use the word tokenizer to tokenize the sentences. There we get a dataset in the form of document-sentence-word. Finally, we convert all words to lower case, remove stop words and words whose length is less than three. To prevent the influence of affixes required, e.g., Grammatical role, tense, we remove the morphological affixes from words, leaving only the stem of the word.

2.4. Classification Model

We use the hierarchical attention network (HAN)[4] as our classification model to identify vulnerability fix commits. The commit message is like a document. Each commit message contains at least one sentence; each sentence contains more than one word. Words and sentences that consist commit message describes the purpose of this commit together. Anyone of these words or sentences may mean different things in different contexts. HAN has a comprehensive consideration of word-level and sentence-level, which would result in better performance. The structure of HAN is shown in Figure 2.

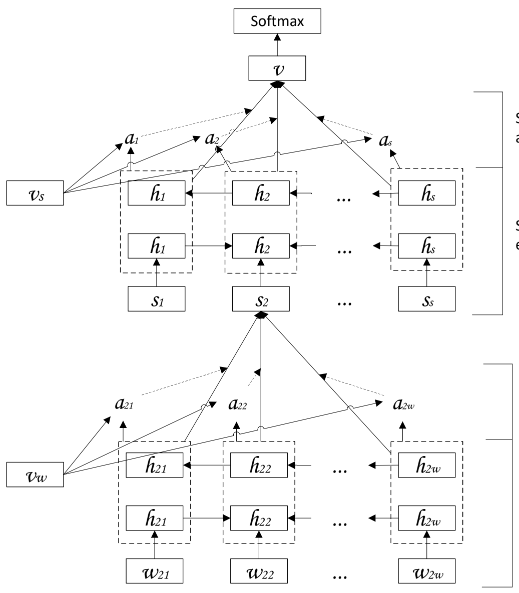

Figure 2. The structure of HAN

The attention mechanism lets the model pay more or less attention to individual words and sentences when constructing the representation of the document. Words or sentences that are more important to the document will get more attention and higher weight values, which enables HAN to generate a vector representation for the entire document based on the different importance of each word and sentence. HAN's two layers both have a bidirectional GRU encoder to represent words or sentences and a corresponding attention mechanism.

GRU (gated recurrent units)[5] is a gating mechanism in recurrent neural networks. Similar to LSTM[6], it is proposed to solve problems about long-term memory and vanishing gradient.

GRU has two gates, update gate and reset gate. The update gate determines how much information should pass in the past, and the reset gate decides how much information should be discarded in the past.

The update gate used the sigmoid function to decide which values to pass. At time step t, the update gate z_t:

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

The reset gate generates the value it wants to discard from the previous time step by multiplying the concatenated value of the previous time step and the current time step by r_t.

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

The current memory is:

$$h'_t = \tanh(W_{x_t} + r_t \odot Uh_{t-1})$$

The final memory of the present time step is:

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h'_t$$

Bidirectional GRU can be seen as two unidirectional GRUs.The current hidden layer state is jointly determined by the current input x_t, the input of the forward hidden state, the output of the inverted hidden state at time $t - 1$.

$$\overrightarrow{h_t} = GRU(x_t, \overrightarrow{h_{t-1}}) \quad (1)$$

$$\overleftarrow{h_t} = GRU(x_t, \overleftarrow{h_{t-1}}) \quad (2)$$

The hidden layer state at time t is obtained by the weighted sum of the forward hidden layer state and the reverse hidden layer state:

$$h_t = w_t \overrightarrow{h_t} + v_t \overleftarrow{h_t} + b_t$$

W_t and v_t respectively represent the weights of the forward hidden state h_t and the reverse hidden state h_t corresponding to the bidirectional GRU at time t, and b_t represents the bias corresponding to the hidden state at time t.

Attention mechanism

In the word-level layer, the Attention mechanism can give different weights to each word in a sentence and pick the words that are important to the sentence and integrate these word representations into a sentence vector s.

$$u_{it} = \tanh(W_w h_{it} + b_w) \quad (3)$$

$$a_{it} = \frac{exp(u_{it} \top u_w)}{\sum_t exp(u_{it} \top u_w)} \quad (4)$$

$$s_i = \sum_t a_{it} h_{it} \quad (5)$$

Here we use word2vec to train the word vector model and directly import the word embedding model in the experiment.

Document vectors are obtained in the same way as sentence vectors. The sentence-level vector u is introduced to measure the importance of sentences. The attention mechanism can reward sentences that have contributed to document classification.

$$u_i = \tanh(W_s h_i + b_s) \quad (6)$$

$$a_i = \frac{exp(u_i \top u_s)}{\sum_i exp(u_i \top u_s} \quad (7)$$

The document vector v synthesizes all the information of sentences.

$$v = \sum_i a_i h_i$$

3. Experiment

Environment We implement the hierarchical attention network in Python by using Pytorch and using NLTK[3] to preprocess text data, Gensim to training pre-trained word embedding models. We run our experiments on a server with NVIDIA GeForce RTX 2070 GRU and Intel Xeon E5-2650 v4 CPU.

Training We use the data collected in section 2 to train the hierarchical attention network. After tuning the parameters for our models, we set the hidden size of word level and sentence level attention to 200, the number of layers in GRU to 2, the dropout to 0.5.

To reduce the computational overhead and ensure convergence, we use the minibatch stochastic gradient descent with the optimization proposed by Kingma et al.[6]. The batch size is set to 32, and the learning rate is set to 0.002.

We use the word2vec embedding method[7] to embed the words to vectors. To get better representation performance, we decided to use a 200-dimensional vector to represent the message text and build the word2vec model on over one million commit messages that is.

To evaluate the effectiveness of our automatic vulnerability fix commits recognition system. We randomly split the dataset into three parts at the ratio of 8:1:1, which are training set, validation set and test set.

Evaluation We aimed to construct a classification model to predict whether a given commit is a vulnerability fix commit. To measure vulnerability prediction results, we use the following metrics: Precision (Pre), Recall (Rec), Accuracy (Acc), F1 score (F1) and False Positive Rate (FPR). Here is a brief definition:

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{8}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{9}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{10}$$

$$FPR = \frac{FalsePositive}{TrueNegative + FalsePositive} \tag{11}$$

$$\tag{12}$$

Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. *Recall* is the ratio of correctly predicted positive observations to the all observations in actual class positive. A higher *Precision* ensures that the classifier can correctly predict as many vulnerability fix commits as possible. And higher *Recall* ensures that there are as few false positives as possible in the observations predicted to be positive. *F*1 Score is the weighted average of *Precision* and *Recall*. It takes both false positives and false negatives into account.

In this paper, we try to keep the *FalsePositiveRate* as low as possible while keeping the high *Recall* and high *Precision*. The high *Precision* and low *FalsePositiveRate* ensure that all the positive result is credible. After a set of experiments, our model achieves a new state-of-art result. Our F1 score is 92.72%, precision score is 93.27%, recall score is 92.17%, accuracy score is 92.81%, and false positive rate is 6.56%.

We also compared our method with three existing method for vulnerability commit identification, Stacking algorithm[8], Voting algorithm[9] and LSTM on the same dataset. Comparative experimental results are shown in the Table 2. It can be seen that HAN has an obvious advantage over other methods.

Table 2. Comparative experimental results with other methods

	F1(%)	Pre(%)	Rec(%)	Acc(%)
Stacking	83.46	81.49	85.52	83.06
Voting	65.49	87.89	52.29	72.50
LSTM	57.99	52.82	64.29	51.87
HAN	**92.72**	**93.27**	**92.17**	**92.81**

The *Precision* score of our method is significantly higher than other methods, which means HAN can correctly predict more vulnerability fix commits. We also used the collected commit data to build our own vulnerability database. Practice proved that this method could greatly improve the efficiency of collecting and verifying vulnerable code. We costed 100 person-hours to build a dataset of vulnerable code for vulnerability research in Chinese National Computer Network Intrusion Protection Center[5].

Discussion Compared to other methods, HAN can have a better understanding of the sample by applying the attention mechanism at both word and sentence level to distinguish which word or which sentence is more important for vulnerability fix commit. We find that the stacking algorithm and the voting methods improve the classification effect by increasing the complexity of the model, but they cannot understand the natural language text well, resulting in the inability to achieve the best results. The LSTM model widely used in the NLP field can only learn the sequence structure and cannot deeply understand the weight of different sentences and different words, which leads to a low precision rate. The false positive rates of Stacking, Voting, LSTM are 19.39%, 7.19%, 61.41%, while the false positive rate of our method is 6.56%. We think that the higher false positive rate of other tools is because they only consider the words relationships in the samples, and not segment the sample at sentence level. The same group of words could express different meanings in one sentences and in two different sentences. By segmenting samples at both sentence level and word level, HAN can identify vulnerability fix commits in better performance. On the other hand, some commit messages are not standardized enough, the comments are not submitted in strict accordance with the git instructions. The expressions is too ambiguous that our model can not learn them well which causes some false positive and false negative.

4. Related Work

In recent years, more and more works realize the significance of patches for vulnerability discovery and exploitation. In the following, we discuss related work from the aspects of data collection methods and classification algorithms.

About data collection, Kim et al.[10],Weiss et al.[11] and other papers[12][13][14] collected vulnerabilities related commits by retrieving keywords such as "CVE", "bug", "buffer overflow", etc. in the git log. The searching process aims at the commit message. The commit would match the searching conditions when the commit message contains the keywords, which shows that our training for the commit message is correct from the side. However, the set of searching keywords is highly dependent on the researchers' professionalism and it is easy to miss some atypical samples. Locating vulnerable code areas by simply searching commit and comments with key words would lead to a relative high false negative rate. Table 3 shows that the frequency

[5]http://nipc.org.cn/

Table 3. The frequency of specific keywords in the vulnerability fix commit

fix	cve	bug	security	overflow	patch	vulnerability
83.90%	23.02%	20.22%	14.56%	10.18%	7.88%	4.76%

of different keywords in the vulnerability fix commits' comments. We can see that not all fixes contain obvious vocabulary related to the vulnerability. It is fast to search for keywords directly, but the accuracy is not high enough.

Many papers[8][15][16] collected commit data by mapping from NVD's description for vulnerabilities to GitHub. They consider all the reference URLs in the description which hosted on GitHub as the vulnerability related commit, whether or not there is a "PATCH" tag. In this paper, we also start with the reference link. In addition, we considered the tag of the reference link. We only used the URLs with a "PATCH" tag to ensure the accuracy.

About commit classification, Zaman et al.[17] made a case study on different types of bugs in Firefox. They analyze the difference between security and performance bug in Firefox from the following aspects: the time to fix, the number of authors, the number of changed lines and changed files. Tian et al.[12] presented an approach to identify bug fixing patches in Linux kernel based on both number information, including the number of files changed, the number of hunks, the number of loops added, etc. and commit message. Their classification algorithm integrates Learning from Positive and Unlabeled Examples (LPU)[18] and Support Vector Machine (SVM)[19]. The feature extraction of these works is somewhat complicated and it is difficult to realize automated extraction. Besides, Zhou et al.[9] identify the vulnerability commit based on commit message using an ensemble learning algorithm. They use logistic regression to choose the best one of the six basic classifiers through a set of K-fold stacking algorithms; the six basic classifiers are Random Forest, Gaussian Naive Bayes, K-nearest Neighbors, Linear SVM, Gradient Boosting and AdaBoost. Wang et al.[8] adopt a voting classifier that ensembles five classifiers, Random Forest, Bayes Net, Stochatic Gradient Descent, Sequential Minimal Optimization and Bagging. The features they used are all number information, such as the number of changed files, the number of total/net modified functions. In this paper, we use the commit message as the only feature and adopted the hierarchical attention network as the classification predictive model. HAN can have a better understanding of the sample by applying the attention mechanism at both word and sentence level and the experiment shows that it is better than the nonhierarchical methods.

5. Conclusion

To solve the problem of insufficient data for deep learning applied to vulnerability detection, we proposed an automatic approach for vulnerability fix commit detection.

Considering the commit message's organizational characteristics, paragraph consisting of one or more sentences, describing the purpose of a commit, we choose to apply bidirectional GRU with attention mechanism at the word-level and sentence-level separately. Our method can analyze the commit message in the entire GitHub repository and identify the vulnerability fix commit in them. If any developer uses an existing open-source software code from GitHub, this approach can help them to discover secret patches[8] and potential threats in the reference repositories. Our initial dataset used to train the hierarchical attention network model is 8,814 commit messages from 316 open-source software's source code repositories. The experiment shows that our model works well, with F1 of 92.71%, and can significantly improve the efficiency of vulnerable code collection.

The commit dataset we constructed in this paper can not only serve as security suggestions for other researchers but also provide data foundation for source-based vulnerability detection. In future work, we will test this model on more open-source software and extract more vulnerable code based on this dataset.

6. Acknowledgements

This research was supported by The National Natural Science Foundation of China (No.U1836210, No.61572460).

References

[1] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z. *et al.* (2018) Vuldeepecker: A deep learning-based system for vulnerability detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018* (The Internet Society).

[2] Zhou, Y., Liu, S., Siow, J.K., Du, X. and Liu, Y. (2019) Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B. and Garnett, R. [eds.] *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information*

Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada: 10197–10207.

[3] Xue, N. (2011) Steven bird, evan klein and edward loper. *Natural Language Processing with Python*. o'reilly media, inc 2009. ISBN: 978-0-596-51649-9. *Nat. Lang. Eng.* **17**(3): 419–424. doi:10.1017/S1351324910000306.

[4] Yang, Z., Yang, D., Dyer, C., He, X., Smola, A.J. and Hovy, E.H. (2016) Hierarchical attention networks for document classification. In Knight, K., Nenkova, A. and Rambow, O. [eds.] *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016* (The Association for Computational Linguistics): 1480–1489.

[5] Cho, K., van Merrienboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H. and Bengio, Y. (2014) Learning phrase representations using RNN encoder-decoder for statistical machine translation. In Moschitti, A., Pang, B. and Daelemans, W. [eds.] *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL* (ACL): 1724–1734.

[6] Kingma, D.P. and Ba, J. (2015) Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y. [eds.] *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[7] Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S. and Dean, J. (2013) Distributed representations of words and phrases and their compositionality. In Burges, C.J.C., Bottou, L., Ghahramani, Z. and Weinberger, K.Q. [eds.] *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*: 3111–3119.

[8] Wang, X., Sun, K., Batcheller, A.L. and Jajodia, S. (2019) Detecting "0-day" vulnerability: An empirical study of secret security patch in OSS. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019* (IEEE): 485–492. doi:10.1109/DSN.2019.00056.

[9] Zhou, Y. and Sharma, A. (2017) Automated identification of security issues from commit messages and bug reports. In Bodden, E., Schäfer, W., van Deursen, A. and Zisman, A. [eds.] *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017* (ACM): 914–919. doi:10.1145/3106237.3117771.

[10] Kim, S., Woo, S., Lee, H. and Oh, H. (2017) VUDDY: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017* (IEEE Computer Society): 595–614. doi:10.1109/SP.2017.62.

[11] Weiss, C., Premraj, R., Zimmermann, T. and Zeller, A. (2007) How long will it take to fix this bug? In *Fourth International Workshop on Mining Software Repositories, MSR 2007 (ICSE Workshop), Minneapolis, MN, USA, May 19-20, 2007, Proceedings* (IEEE Computer Society): 1. doi:10.1109/MSR.2007.13.

[12] Tian, Y., Lawall, J.L. and Lo, D. (2012) Identifying linux bug fixing patches. In Glinz, M., Murphy, G.C. and Pezzè, M. [eds.] *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland* (IEEE Computer Society): 386–396. doi:10.1109/ICSE.2012.6227176.

[13] Lukins, S.K., Kraft, N.A. and Etzkorn, L.H. (2008) Source code retrieval for bug localization using latent dirichlet allocation. In Hassan, A.E., Zaidman, A. and Penta, M.D. [eds.] *WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15-18, 2008* (IEEE Computer Society): 155–164. doi:10.1109/WCRE.2008.33.

[14] Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M. and Nguyen, T.N. (2010) Recurring bug fixes in object-oriented programs. In Kramer, J., Bishop, J., Devanbu, P.T. and Uchitel, S. [eds.] *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010* (ACM): 315–324. doi:10.1145/1806799.1806847.

[15] Li, F. and Paxson, V. (2018) A large-scale empirical study of security patches. *login Usenix Mag.* **43**(1). URL `https://www.usenix.org/publications/login/spring2018/li`.

[16] Gkortzis, A., Mitropoulos, D. and Spinellis, D. (2018) Vulinoss: a dataset of security vulnerabilities in open-source systems. In Zaidman, A., Kamei, Y. and Hill, E. [eds.] *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018* (ACM): 18–21. doi:10.1145/3196398.3196454, URL `https://doi.org/10.1145/3196398.3196454`.

[17] Zaman, S., Adams, B. and Hassan, A.E. (2011) Security versus performance bugs: a case study on firefox. In van Deursen, A., Xie, T. and Zimmermann, T. [eds.] *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings* (ACM): 93–102. doi:10.1145/1985441.1985457, URL `https://doi.org/10.1145/1985441.1985457`.

[18] Li, X. and Liu, B. (2003) Learning to classify texts using positive and unlabeled data. In Gottlob, G. and Walsh, T. [eds.] *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003* (Morgan Kaufmann): 587–594. URL `http://ijcai.org/Proceedings/03/Papers/087.pdf`.

[19] Cortes, C. and Vapnik, V. (1995) Support-vector networks. *Mach. Learn.* **20**(3): 273–297. doi:10.1007/BF00994018.