Prediction of Common Weakness Probability in C/C++ Source Code Using Recurrent Neural Networks

Petr Vytovtov[1,2], Kirill Chuvilin[1,3]
[1]Moscow Institute of Physics and Technology (State University), Moscow, Russia
[2]Kalashnikov Izhevsk State Technical University, Izhevsk, Russia
[3]Institute of Computing for Physics and Technology, Protvino, Russia
osanwevpk@gmail.com, kirill@chuvilin.pro

Abstract—**The article considers source code written in C/C++ programming language. The problem is the automatic detection of potential vulnerabilities from the common weakness enumeration. The assumption is that the presence of a vulnerability is determined by the local context. Machine learning approaches based on recurrent neural networks are investigated. The training sample is built from known common weakness fixes in public software code repositories. A new static analysis approach based on recurrent neural networks is proposed. It is tested on source code blocks with different sizes and demonstrates good quality in the terms of accuracy, F1 score, precision and recall. The proposed method can be used as a part of the source code quality analysis system and can be improved for more deeply source code analysis or for collaboration with source code autofixing tools.**

I. INTRODUCTION

Nowadays, software is involved in all application areas. And the tools for its creation are accordingly rapidly developed. Now they are not only compilers and editors, but also a large number of utilities designed to analyze and improve the quality of the source code.

Depending on the application area, different requirements are imposed on the source code and the resulting software. But the lack of potential vulnerabilities is almost always critical in commercial projects. Therefore, a significant part of the software development process is devoted to the detection of the corresponding errors. It is important to note that the errors of this kind are difficult to detect, since they are often connected with logic and poorly formalized.

Therefore, the automating of searching and fixing vulnerabilities process is an extremely urgent task. Errors that are not related to logic can be handled by compilers and linkers. External code analysis tools are usually used to detect the rest. In section II we consider some well-known approaches. Common Weakness Enumeration (CWE) was developed to unify the vulnerability search process [1]. It contains a catalog of known software weaknesses and vulnerabilities. In section III-A we describe the data obtained from it in more detail.

In this paper we consider applications written in C/C++ programming language and apply machine learning approach (see section III) to automate the search for vulnerabilities from CWE. Results presented in the article are a part of the system for analyzing software source code quality developed as a dynamic library for clang compiler [2]. The source code of the used models and preprocessing scrips is available on GitHub under the MIT license terms [3].

II. RELATED APPROACHES AND SOLUTIONS

As it was stated in the introduction, the problem is extremely urgent, therefore several approaches to its solution are known. Existing tools for vulnerability detection can be divided into dynamic and static analyzers.

Dynamic analyzers detect vulnerabilities during the software execution. This approach is implemented in such tools as Avalanche [4], Valgrind [5], !exploitable [6], etc. Most of them were developed for detecting memory management bugs and are poorly adapted for other problems in software. Dynamic analysis provides very accurate information about vulnerabilities but requires a lot of time because of running software with different starting conditions. Also, (and as a result of previous statement) it is difficult to check all possible program execution paths. Thus, static analysis is more preferable for software checking.

Static analyzers, such as Klockwork [7], PVS Studio [8], Clang Static Analyzer [9], MOPS [10], PC-Lint [11], Parasoft C++test [12], CodeSurfer [13], FXCop [14], etc., do analysis without executing the program. As a result this approach can provide inaccurate information about vulnerabilities in software. Especially it is noticeable in static analyzers which use manually created templates and rules for detecting bugs. This behavior is illustrated in Fig. 1 and Fig. 2 gotten from CWE [15] and Microsoft [16] official sites respectively. These images show that different software analyzers can provide information about different vulnerabilities.

Several tools, for example BoundsChecker, implement both approaches. They can provide more accurate information about vulnerabilities because of combining two approaches, but require more time for analysis because of the same reason. Thus we have decided to use machine learning approach which will accurately detect vulnerabilities in software source code and will requires less time for analysis.

One more related research is dedicated to the automatic synthesis of formal correcting rules for LaTeX documents. In the work [17] each document is represented as a syntax tree. Tree node mappings of initial documents to edited documents form the training set, which is used to generate the rules. Rules

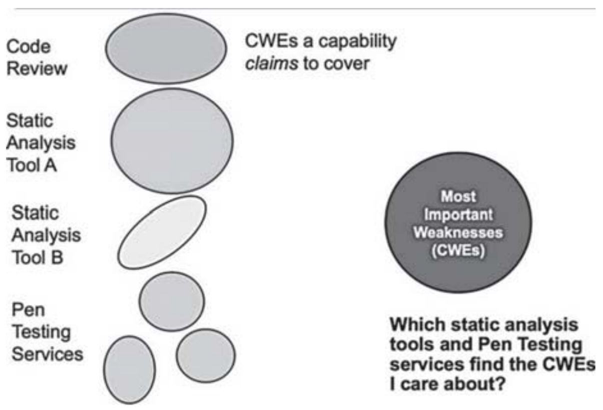
Code Review
CWEs a capability claims to cover
Static Analysis Tool A
Static Analysis Tool B
Pen Testing Services
Most Important Weaknesses (CWEs)
Which static analysis tools and Pen Testing services find the CWEs I care about?

Fig. 1. CWE analyzers matching claims

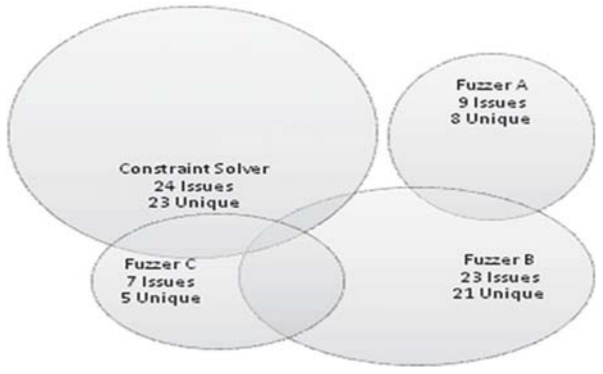

Fig. 2. Overlapping between different analyzers

with a simple structure, which implement removal, insertion or replacing operations of single node and use linear sequence of nodes to select a position are synthesized primarily. The constructed rules are grouped based on the positions of applicability and quality. The rules that use tree-like structure of nodes to select the position are studied. The changes in the quality of the rules during the sequential increase of the training document set are analyzed.

The common part with the task considered in this work is detection of hardly formalized errors in the source code. Moreover, in both cases the machine learning approaches described in detail in the section III are used. However, in the work [17] tree structures are investigated, while in this research we consider linear sequences that are easier to obtain and understand. In addition, the data sets suitable for training differs considerably.

III. Machine learning approach

In this work we consider an approach based on methods of machine learning. This means that the model parameters are adjusted (or *trained*) according to the *training set* of known results (in our case this set if formed by source code snippets that are known to contain weaknesses).

Here is the formal statement of the problem.
Given:

- S_0 is the source snippets with known vulnerabilities positions,
- V_0 is the CWE markup for S_0,
- S is the source code to detect vulnerabilities in.

Required: V (the CWE markup for S).

Visually the workflow of this approach is shown on Fig. 3. The training set is formed by V_0 corresponding to the precedents and S_0 forming the context. Mode detailed these sets are described in section III-A. This information is used to train the classification model in a such way that it will take a new portion of the source code S and predict vulnerability presence rate V. In more general case the model should specify the CWE identifier. The process of constructing the model is described in detail in section III-C.

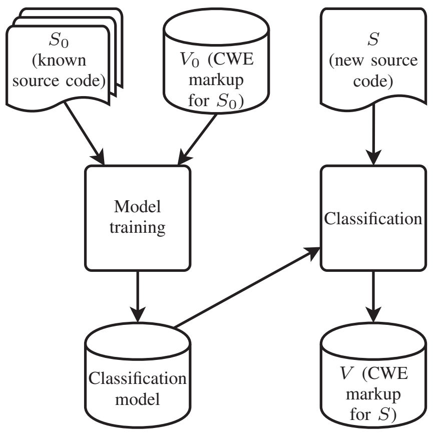

Fig. 3. Machine learning workflow

A. The dataset

As mentioned above, our task is to predict the probability of emerging weakness in functions written in C/C++ programming language. For this we decided to use CWE as a formal list of software weakness which was created for improving the process of analyzing the software source code, architecture, and design. This enumeration contains information about common weakness templates, what technical impact is possible because of the weakness, and which stages of software life cycle is important for this weakness.

National Institute of Standards and Technology has published the dataset based on this list [21]. It contains source code of open source software which has marked code lines with type of vulnerability. From this dataset, we chose the information about four open source software products: GNU Grep, Wireshark, FFmpeg, ang Gimp.

On the next step, we processed the data for getting and labeling blocks of source code, where the block is started with open brace, finished with close brace, and may contain other blocks. It is necessary because it is important to know the location of the weakness. And the less is source code block the better will be position prediction. Thus, we have got 136086 blocks without bugs and 2858 blocks with bugs.

On the last step of preparing dataset, we selected separately five groups of blocks, where the size of each block is less or equal to 1KiB, 1.5KiB, 2KiB, 3KiB, and 4KiB respectively. We think the greater size of source code blocks will not provide accurate information about weaknesses position. After this, the numbers of blocks with bugs and blocks without bugs were aligned with undersampling (so that their number will be equal), shuffled, and split to training and test sets. The sizes of these sets are shown in Table I.

TABLE I. THE SIZES OF THE TRAINING AND TEST SETS FOR CODE CLOCK SIZES

Source code block size (less or equal to)	**Training set size**	**Test set size**
1 KiB	554	278
1.5 KiB	1194	598
2 KiB	1688	846
3 KiB	2550	1276
4 KiB	3016	1510

When the dataset is prepared the data should be preprocessed. In our case blocks of source code should be converted to the sequences of lexemes. It is necessary for unification the source code representation in a sequence. For this we used Clang compiler [22] with `-dump-tokens` flag and got its result with a few modifications.

B. The baseline

As a baseline, we chose FastText tool developed by Facebook [18], [19]. This tool uses the hashing for building the embeddings for words and linear methods for text classifying or language modeling. Embeddings for lexemes are calculated as an average value of char n-gram hashes sum. In the following steps they can be used as is or as a base for embeddings for new lexemes calculated in runtime.

We consider FastText as baseline because it implements linear methods for sequences. Therefore, recurrent neural networks usage, such as long-short term memory neural network (LSTM) [20], should improve the results gotten with FastText.

We trained five models: one for each block size. The results are showed in Table II. FastText provides better results for longer sequences, and the sequences size which is less or equal to 3 KiB is enough for training the model.

Also we trained the character-level model with LSTM, but it did not provide useful results. More information about it will be shown in section IV.

C. The model

The architecture of our model is showed on Fig. 4. It consists of an embedding layer, one LSTM layer, and one linear layer with sigmoid function applied to the output.

The embedding layer uses the dictionary which contains all unique tokens which are appeared in chose datasets. Thus, vocabulary sizes are 1132, 2473, 3685, 6512, 8708 lexemes for datasets with blocks size less or equal to 1 KiB, 1.5 KiB, 2 KiB, 3 KiB, and 4 KiB respectively. The identifiers of variables and constants are used “as is” without any modifications.

TABLE II. THE BASELINE QUALITY ESTIMATIONS FOR SOURCE CODE BLOCKS

Source code block size (less or equal to)	**Training set**	**Test set**
Accuracy		
1 KiB	0.7	0.69
1.5 KiB	0.83	0.85
2 KiB	0.87	0.87
3 KiB	0.91	0.9
4 KiB	0.92	0.91
F1 score		
1 KiB	0.58	0.57
1.5 KiB	0.82	0.83
2 KiB	0.87	0.86
3 KiB	0.9	0.89
4 KiB	0.9	0.91
Precision		
1 KiB	0.95	0.97
1.5 KiB	0.88	0.93
2 KiB	0.93	0.93
3 KiB	0.96	0.94
4 KiB	0.96	0.96
Recall		
1 KiB	0.41	0.4
1.5 KiB	0.76	0.75
2 KiB	0.81	0.8
3 KiB	0.85	0.84
4 KiB	0.87	0.86

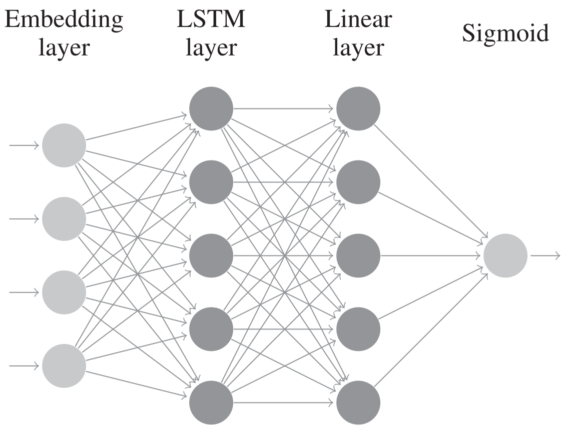

Fig. 4. The architecture of the model

The LSTM layer is defined [20] as

$$\begin{aligned}
f_t &= \sigma_g(W_f x_t + Y_f h_{t-1} + b_f),\\
i_t &= \sigma_g(W_i x_t + Y_i h_{t-1} + b_i),\\
o_t &= \sigma_g(W_o x_t + Y_o h_{t-1} + b_o),\\
c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + Y_c h_{t-1} + b_c),\\
h_t &= o_t \circ \sigma_h(c_t),
\end{aligned}$$

where x_t is an *input vector* (one lexeme in our case) and h_{t-1} is an *output vector* from the previous step. f_t is a *forget gate* which is calculated with sigmoid function σ_g weights matrices W_f and Y_f and bias vector b_f. The same way is used for calculating the *input gate* i_t and *output gate* o_t. The LSTM *cell state* on the current step c_t is calculated as element-wise multiplication between *forget gate* and *cell state* from previous state c_{t-1} (here some information can be saved or forgotten) with following addition the *input state* calculated as element-wise multiplication between *input gate* and $tanh$

function of *input vector* and *output vector* from the previous step. The output of the LSTM unit is calculated as a element-wise multiplication between *output gate* and $tanh$ function of current *cell state*. On the first step h and c are zero vectors.

The size of the vector h (hidden size; hidden units count) is a *hyperparameter*, which value should be set manually. We tested our model with different hidden sizes: 50, 100, 150, 200, 250, 300, 350, 400, 450, and 500 units, using *Grid Search* method. The results of the experiments are shown in section IV.

A linear layer with sigmoid function σ returns a single number showing the probability of weakness of the analyzed source code block.

For training the model we are minimizing the cross-entropy function:

$$L = \min(-y \log(\sigma(\hat{y})) - (1 - y) \log(1 - \sigma(\hat{y}))),$$

where y is a real target value, $\hat{y}$ is a predicted label for the same issue, and σ is a sigmoid function.

We used Adam [23] as an optimization algorithm with learning rate equals to 0.001, and batch size equal to 1 [24]. The training process was during 10 epochs for each pair of (hidden size; block size) values.

IV. Experiments

We searched optimal values for two hyperparameters: the LSTM hidden units count and the maximum size of analyzed source code blocks. As mentioned in the previous section, we tried five sizes of blocks. Also, we tried five sizes of LSTM hidden layer. The results are presented in Table III.

The table shows that the best value of the loss function appears after eight epochs with 250 LSTM hidden units and source code blocks size less or equal to 3 KiB. The graph of the loss function for this configuration is shown on Fig. 9. The Table III also shows that one layer LSTM network with 250 hidden units provides a local minimum for all test source code blocks from our dataset for the CWE prediction task.

It is clear from Fig. 9 that the model is overfitted after eight epochs. For configuration with 250 LSTM hidden units and source code blocks size less or equal to 3 KiB the overfitting is not giant. However Figs. 5–10 show that the overfitting is much significant for the other configurations. The general conclusion is that there is an epoch after which overfitting begins. We choose such epoch as the optimal for the model adjustment.

Lets consider the best and the worst variants of out model in detail and compare them with the best FastText baseline (section III-B). The summary results are presented in Table IV.

It could be mentioned that the worst variant of our model is better than the FastText baseline for the same source code blocks size (less or equal to 1 KiB), and it is similar to the FastText baseline for blocks size less or equal to 2 KiB. The best variant of our model is better than all FastText baselines. The ROC curves for both models are shown in Fig. 11 and Fig. 12.

Thus, we claim that our model has good quality in the problem of software common weakness prediction. It provides

TABLE III. Loss function values for the best epoch depends on the LSTM hidden units count

Source code block size (less or equal to)	**The best epoch number**	**Training set loss**	**Test set loss**
50 **hidden units**			
1 KiB	3	0.37	0.38
1.5 KiB	2	0.32	0.37
2 KiB	3	0.27	0.28
3 KiB	3	0.22	0.21
4 KiB	3	0.23	0.24
100 **hidden units**			
1 KiB	3	0.38	0.36
1.5 KiB	3	0.32	0.33
2 KiB	7	0.26	0.29
3 KiB	2	0.24	0.24
4 KiB	2	0.26	0.27
150 **hidden units**			
1 KiB	6	0.39	0.35
1.5 KiB	3	0.31	0.33
2 KiB	2	0.27	0.31
3 KiB	4	0.24	0.22
4 KiB	2	0.24	0.24
200 **hidden units**			
1 KiB	4	0.36	0.38
1.5 KiB	2	0.32	0.35
2 KiB	3	0.25	0.25
3 KiB	5	0.18	0.19
4 KiB	2	0.24	0.24
250 **hidden units**			
1 KiB	9	0.29	0.32
1.5 KiB	4	0.27	0.29
2 KiB	4	0.21	0.24
3 KiB	8	0.17	0.17
4 KiB	5	0.19	0.2
300 **hidden units**			
1 KiB	6	0.36	0.37
1.5 KiB	3	0.26	0.3
2 KiB	5	0.25	0.28
3 KiB	3	0.2	0.19
4 KiB	3	0.2	0.22
350 **hidden units**			
1 KiB	5	0.33	0.41
1.5 KiB	5	0.25	0.27
2 KiB	2	0.28	0.3
3 KiB	6	0.26	0.26
4 KiB	2	0.2	0.21
400 **hidden units**			
1 KiB	8	0.31	0.39
1.5 KiB	3	0.26	0.29
2 KiB	10	0.22	0.22
3 KiB	8	0.16	0.18
4 KiB	3	0.18	0.22
450 **hidden units**			
1 KiB	7	0.29	0.32
1.5 KiB	4	0.29	0.36
2 KiB	5	0.22	0.26
3 KiB	4	0.18	0.20
4 KiB	2	0.22	0.22
500 **hidden units**			
1 KiB	6	0.36	0.39
1.5 KiB	3	0.3	0.36
2 KiB	4	0.22	0.24
3 KiB	3	0.18	0.21
4 KiB	2	0.21	0.21

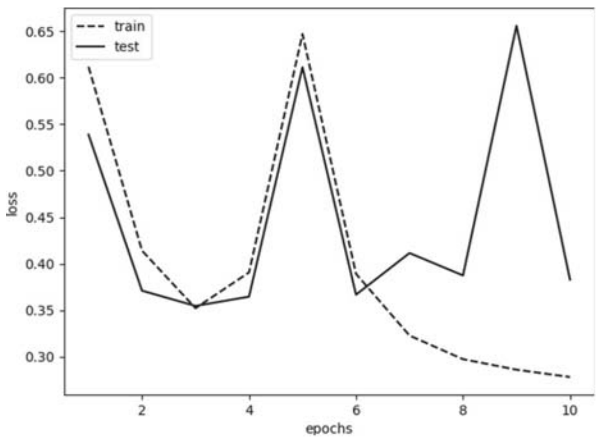

Fig. 5. The loss function of training and test sets for model with 150 LSTM hidden units trained with source code blocks with size less or equal to 1 KiB

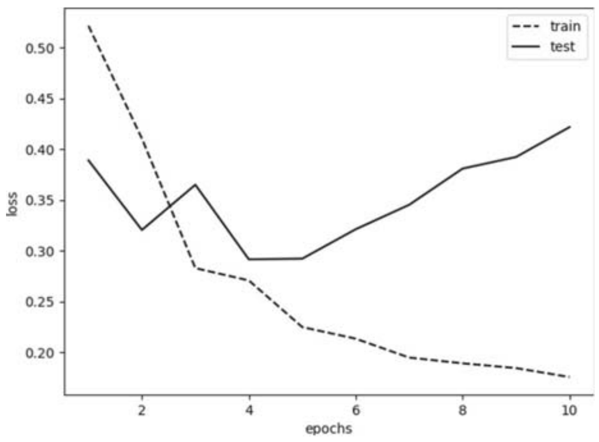

Fig. 7. The loss function of training and test sets for model with 250 LSTM hidden units trained with source code blocks with size less or equal to 1.5 KiB

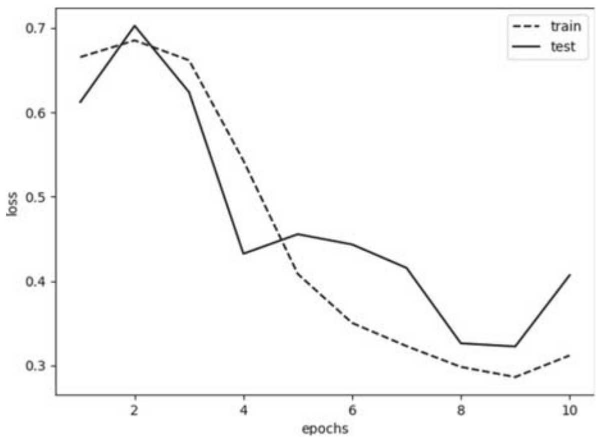

Fig. 6. The loss function of training and test sets for model with 250 LSTM hidden units trained with source code blocks with size less or equal to 1 KiB

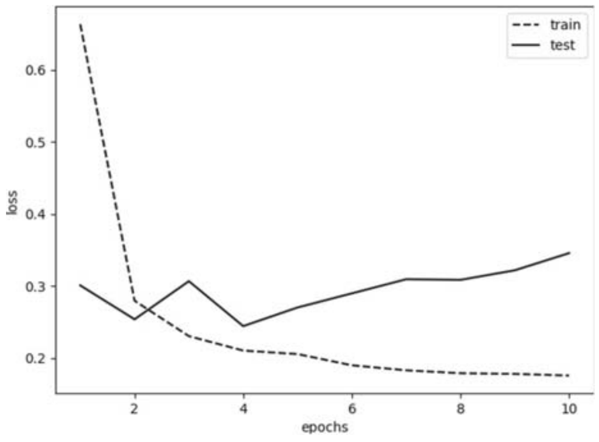

Fig. 8. The loss function of training and test sets for model with 250 LSTM hidden units trained with source code blocks with size less or equal to 2 KiB

TABLE IV. THE COMPARISON OF THE QUALITY ESTIMATIONS

Model	Training set	Test set
Accuracy		
Our best	0.94	0.94
Our worst	0.87	0.86
FastText best	0.92	0.91
F1 score		
Our best	0.94	0.93
Our worst	0.85	0.85
FastText best	0.9	0.91
Precision		
Our best	0.98	0.96
Our worst	0.97	0.95
FastText best	0.96	0.96
Recall		
Our best	0.91	0.91
Our worst	0.76	0.76
FastText best	0.87	0.86
Recall		
Our best	0.91	0.91
Our worst	0.76	0.76
FastText best	0.87	0.86

94% of accuracy on the training and test sets which is better than our baseline based on FastText. The precision and recall values also are better.

As mentioned in section III-B, we also tried the character-level recurrent model with the same dataset. Unfortunately, it did not give us the satisfactory level of prediction. The graphs of loss function for this model are shown in Fig. 13 and Fig. 14.

V. DISCUSSION

Three approaches to software common weakness prediction are considered: FastText and two types of recurrent neural networks. The results of our work and experiments show that the recurrent neural networks provide better quality in prediction than linear method FastText. It is because linear models do not save dependencies throw time, and weakness prediction in software is possible only if we have the information about the history of program execution. Furthermore the usage of lexemes in model got better quality than single characters usage because of because of the need to take the context into account during weakness prediction.

The observed features form an understanding of what is needed to improve the results. We propose to build further research from the following sequential steps.

1) Improving the preprocessing stage using embeddings. It is necessary because it is expensive to store all

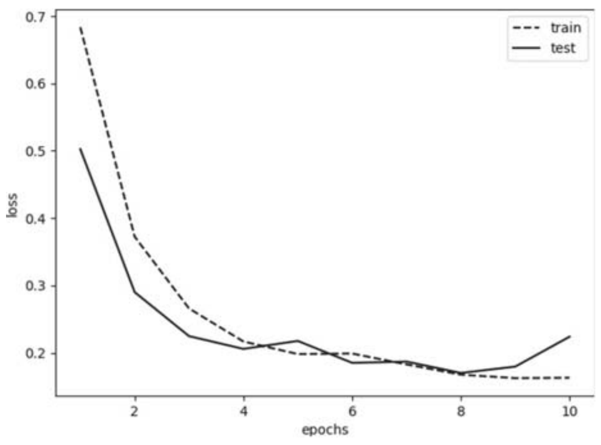

Fig. 9. The loss function of training and test sets for model with 250 LSTM hidden units trained with source code blocks with size less or equal to 3 KiB

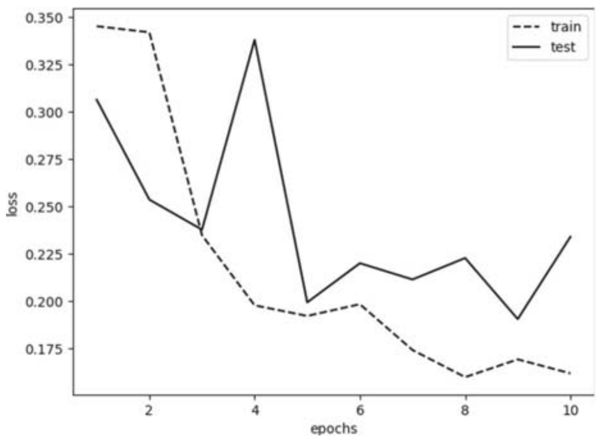

Fig. 10. The loss function of training and test sets for model with 250 LSTM hidden units trained with source code blocks with size less or equal to 4 KiB

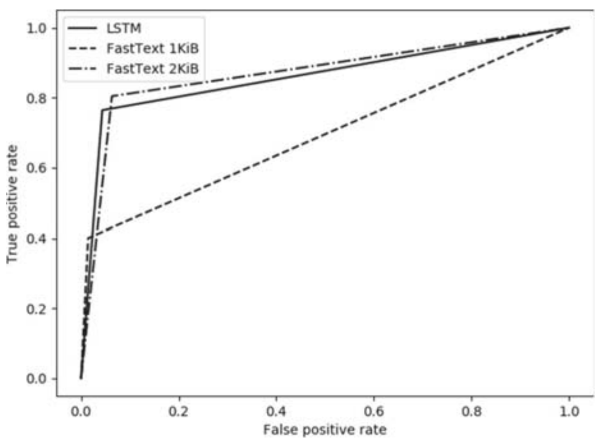

Fig. 11. The ROC curves for the worst variant of our model and for FastText models with source code blocks sizes less or equal to 1 KiB and to 2 KiB

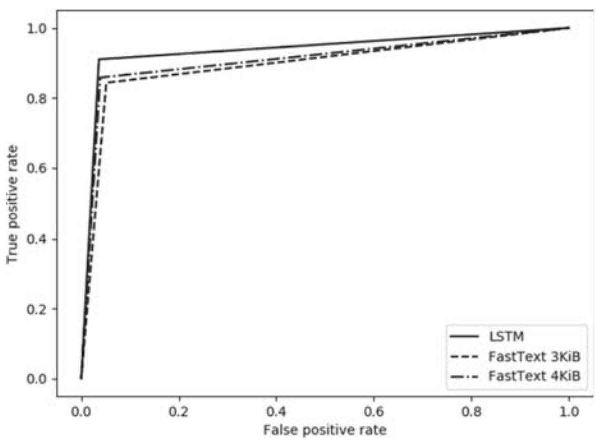

Fig. 12. The ROC curves for the best variant of our model and for FastText models with source code blocks sizes less or equal to 3 KiB and to 4 KiB

possible variables names in the vocabulary. Therefore, the approach to calculate embeddings for variables names in the inference time is needed. This approach is provided, for example, by FastText.

2) Using an abstract syntax tree (AST) as input to the model instead of the sequence of lexemes. In out mind it should improve the model quality because the information about the source code structure can be useful in the task of common weakness prediction forming more complex context. Moreover, it is possible that variables names will not be important if the model know the analyzed source code structure.
3) Testing the model scaling to lager data and longer sequences. We have an assumption that the model can be approximated to long sequences form short. If it is true than the model can be trained much faster and with the less data.
4) Moving the model from C/C++ programming language to LLVM IR language. It is quite useful because we plan to analyze source code written in different programming languages and LLVM IR provides the universal approach for representation high-level programming languages. Also, it will be interesting to check the correlation between predictions based on C/C++ programming language and based on LLVM IR.
5) Changing the binary classification model to multiclass classification and predicting the probability of common weakness types. It can be useful for application fields to know what type of weakness the program has.
6) Integrating the model into clang dynamic library for analyzing software source code during the compilation (or precompilation) time.

VI. Conclusion

In work described in this paper, we considered machine learning approach for the common weaknesses prediction in the source code written in C/C++ programming language. The dataset is formed from the source code and common weakness positions of GNU Grep, Wireshark, FFmpeg, and Gimp.

We managed to achieve the desirable quality of the weakness detection (with 94% if accuracy on test set), thanks to the method based on Recurrent Neural Networks. The LSTM model with 250 hidden units which is trained with source

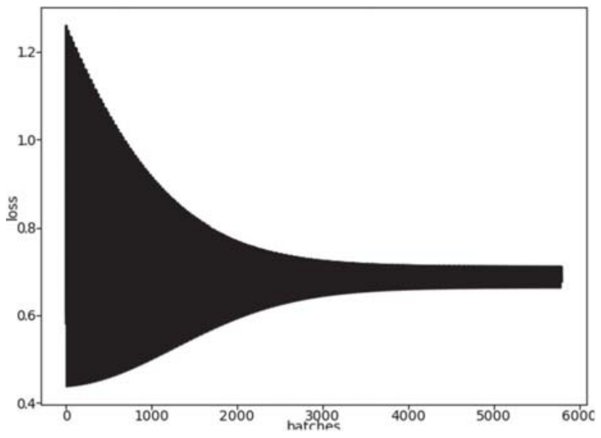

Fig. 13. The graph of character recurrent model loss function throw batches

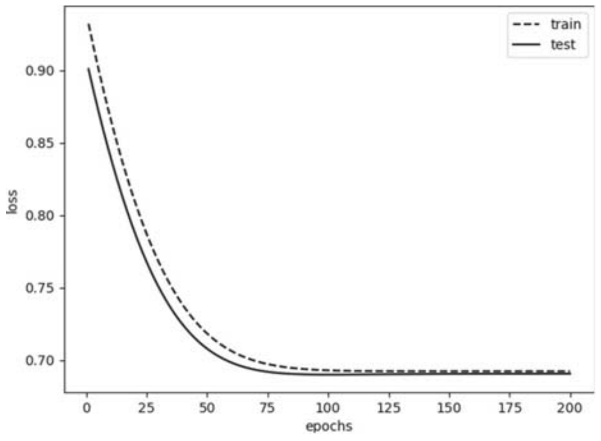

Fig. 14. The graph of character recurrent model loss function throw epochs

code blocks with size less or equal to 3 KiB provides the best prediction quality.

Our research shows that the model has a high predictive ability. We see the potential to extend this approach further in order to implement dynamic library for analyzing software source code. The further work will be aimed to improving and scaling our model. The important part of it is the migrating of the model from C/C++ programming language to LLVM IR, and integrating it into the clang dynamic library.

Also, we can say that FastText provides slightly worse results but it works faster than recurrent neural network. Thus, if the time of analysis is preferable and accuracy is not critical linear methods like FastText could be more suitable.

The source code of our model and preprocessing scripts are available on GitHub under the MIT license terms [3].

ACKNOWLEDGMENT

This work was supported by the RFBR grant 16-07-01267.

REFERENCES

[1] CWE - About DWE. Web: http://cwe.mitre.org/about/index.html

[2] P. Vytovtov, E. Markov, "Source Code Quality Classification Based On Software Metrics", *Proceedings of the 20th Conference of Open Innovations Association FRUCT*, Apr. 2017, pp.505–511.

[3] GitHub - osanwe/source-code-cwe-analyzer Web: https://github.com/osanwe/source-code-cwe-analyzer

[4] Avalanche - Testing the security of app aware devices and networks - Spirent. Web: https://www.spirent.com/Products/Avalanche

[5] Valgrind Home. Web: http://valgrind.org

[6] !exploitable Crash Analyzer - MSEC Debugger Extensions - Home. Web: https://msecdbg.codeplex.com

[7] Source Code Analysis Tools for Security & Reliability — Klocwork. Web: https://www.klocwork.com

[8] PVS-Studio: Static Code Analyzer for C, C++ and C#. Web: https://www.viva64.com/en/pvs-studio

[9] Clang Static Analyzer. Web: https://clang-analyzer.llvm.org

[10] MOPS. Web: http://web.cs.ucdavis.edu/~hchen/mops

[11] Gimpel Software PC-lint Overview. Web: http://www.gimpel.com/html/pcl.htm

[12] Parasoft C/C++test: Comprehensive dev testing tool for C/C++. Web: https://www.parasoft.com/product/cpptest

[13] CodeSurfer — GrammaTech. Web: https://www.grammatech.com/products/codesurfer

[14] FxCop. Web: https://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx

[15] CWE - Detection Methods. Web: https://cwe.mitre.org/community/swa/detection_methods.html

[16] Microsoft TechNet - The History of the !exploitable Crash Analyzer. Web: https://blogs.technet.microsoft.com/srd/2009/04/08/the-history-of-the-exploitable-crash-analyzer/

[17] K. V. Chuvilin, "Machine Learning Approach to Automated Correction of LaTeX Documents", *in Proceedings of the 18th FRUCT & ISPIT Conference, 18–22 April 2016*, Technopark of ITMO University, Saint-Petersburg, Russia. FRUCT Oy, Finland. ISSN 2305-7254, ISBN 978-952-68397-3-8, pp. 33–40. Web: http://fruct.org/publications/fruct18/files/Chu.eps.

[18] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov, "Enriching Word Vectors with Subword Information", arXiv:1607.04606 [cs.CL]

[19] A. Joulin, E. Grave, P. Bojanowski, T. Mikolov, "Bag of Tricks for Efficient Text Classification", arXiv:1607.01759 [cs.CL]

[20] F. A. Gers, J. Schmidhuber, F. Cummins, "Learning to Forget: Continual Prediction with LSTM", *Neural Computation*, 12 (10), 2000, pp. 2451–2471.

[21] Software Assurance Reference Dataset. Web: https://samate.nist.gov/SRD/testsuite.php

[22] "clang": a C language family frontend for LLVM. Web: https://clang.llvm.org

[23] D. P. Kingma, J. Ba, "Adam: A Method for Stochastic Optimization", arXiv:1412.6980 [cs.LG]

[24] T. M. Breuel, "The Effects of Hyperparameters on SGD Training of Neural Networks", arXiv:1508.0278