UDC 004

DeeDP: vulnerability detection and patching based on deep learning

A. Savchenko[1,2], O. Fokin[1,2,a], A. Chernousov[1,2], O. Sinelnikova[2], S. Osadchyi[2]

[1] *National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Institute of Physics and Technology*

[2] *Samsung R&D Institute Ukraine (SRK)*

Abstract

We present the DeeDP system for automatic vulnerabilities detection and patch providing. DeeDP allows to detect vulnerabilities in C/C++ source code and generate patch for fixing detected issue. This system uses deep learning methods to organize rules for deciding whether a code fragment is vulnerable. Patch generation processes can be performed based on neural network and rule-based approaches. The system uses the abstract syntax tree (AST) representations of the source code fragments.

We have tested effectiveness of our approach on different open source projects. For example, Microsoft/Terminal (https://github.com/microsoft/Terminal) was analyzed with DeeDP: our system detected security issue and generated patch which was successfully approved and applied by Microsoft maintainers.

Keywords: vulnerability detection, patch generation, deep learning

Introduction

There are many cyber attacks which are rooted in software vulnerabilities. Prevention of software products compromising is related to application of different techniques e.g. Microsoft Security Development Lifecycle (SDL) and deep software analysis on early stages of development process. But involving huge amount of security experts for software analysis is too expensive, so most promising way is automation of each step: from code examination to errors correction.

This paper represents technology and system DeeDP for detection of vulnerabilities in source code and providing of a patch to fix detected errors. Our technology is based on deep learning approach [1] for extraction of vulnerable fragments of code represented as ASTs [2] and automatic patch generation.

This work is a continuation of research on automating the detecting and fixing vulnerabilities in software. The method of automating vulnerability detection is deeply described in the article [3]. Now we consider the procedure for fixing errors - generating patches.

In section Related works we review existing solutions in the field of patch generation and their disadvantages. Next, we consider overall design and approaches that were used for system creation, as well as a result of technology application.

1. Related works

Automatic patch generation allows you to fix vulnerabilities in software without spending time and money necessary for developers to understand process and correct detected defects[4]. There are two main methods of patch generation: 1) based on the study of a valid code (human patches) (Prophet, SPR, RSRepair, GenProg, AE), 2) based on the use of fixed patterns (Senx, PAR)[5].

The best-known "generate-and-validate" patch approach starts with collection of test input data, where at least one piece of it identifies vulnerability in the software. The patch generation system modifies the program and generates space of patches, then looking for plausible patches in this space (i.e. patches that give the correct output for all test input data).[4] Prophet, SPR, GenProg, RSRepair, AE work is based on this approach[6, 7, 8, 9, 10].

GenProg, AE and RSRepair use various search algorithms (genetic programming, stochastic search, random search) in combination with transformations that remove, insert or change existing program operators[11]. Prophet is focused on study of existing valid human patches [12]. It uses a parameterized logarithmic probabilistic model based on two features extracted from the abstract syntax trees (AST) of each patch: 1) the way the patch changes the source program, 2) the relationship of how the values associated with the patch are used in the source program and in the patched program. Prophet ranks the possible patches generated for the defect according to the probabilities of their correctness [5].

SPR [13] uses a set of conversion schemes to generate patch set. Then it uses a step-by-step program fix process to validate the generated patches by checking them on the original test suite, in which at least one detects vulnerability in the original program. Prophet works with the same search space as SPR, but differs in that it uses its own model for correctness studies and,

[a] onworkfokin@gmail.com

according to the study [4], shows better results than hand-coded SPR heuristics.

This approach has a significant drawback, which was confirmed by research [4], namely, the difficulty in correct evaluation of patches validity due to the small amount of input test data. In regards to this, these systems generate incorrect patches that pass the initial test cases, but remove the functionality of the program and create new vulnerabilities.

In the second method, patches are generated by applying fixed patterns that are written by a person based on the generalization of the rules to correct common vulnerabilities. PAR and Senx work in this way [5], [14]. The disadvantage of this method is the need to review and summarize a large number of written patches to fix vulnerabilities, as well as to write a large number of patterns and correctly process all the variables that are influenced by patch [15].

2. Task statement

Based on aforementioned discussion about fixing issues in software, let discuss our approaches. Before we want to determine some entities: P – product, some software project C/C++ language with available source code; X – fragment of vulnerable source code, which includes target function and context of execution (target function – it is function which can be used in wrong way and will lead to vulnerabilities in software); $\widetilde{X}$ – patch, fragment of code is related to X, but without vulnerability (target function is used in right way). General feature set and functionalities of product P have been the same as before applying changes $\widetilde{X}$.

We want to build function: $\widetilde{X} = F(X, A)$, where $F(\bullet)$ – is transformation of source code X (C/C++) represented as AST to respective source code $\widetilde{X}$ without weaknesses, A – additional parameters.

We are considering two approaches:

- creation patch $\widetilde{X}$, based on deterministic approach (rule-based), when $F(\bullet)$ is represented as rule (pattern) how to transform X, according to type of CWE [16]. So, in this case $A = CWE_{type}$, $F(X, CWE_{type})$
- Generation $\widetilde{X}$ from X, with neural network - $F(\bullet)$. Training neural network will be supervised, based on collected samples from existed open source repositories with detected weaknesses and after patch-fix from contributor. In this case transformation function doesn't need any additional parameters $\widetilde{X} = F(X)$, so neural network should remember dependency how was fixed such issue in represented dataset.

It means that the first approach more reliable, because it assumes that developer which create respective rule is high experience, but this approach cannot be scaled. The second approach can be fully automated and can be expanded on different cases: fixing known weaknesses; automatic converting code to approved project style (handling exception, using specific functions and etc.); applying specific code style. The trained neural network $F(\bullet)$ will apply extracted from dataset dependencies. The second approach is much more promising, but it needs enough examples for training procedure. In Result section we provided examples how was generated patch with rule and based on neural network.

3. General architecture of proposed method

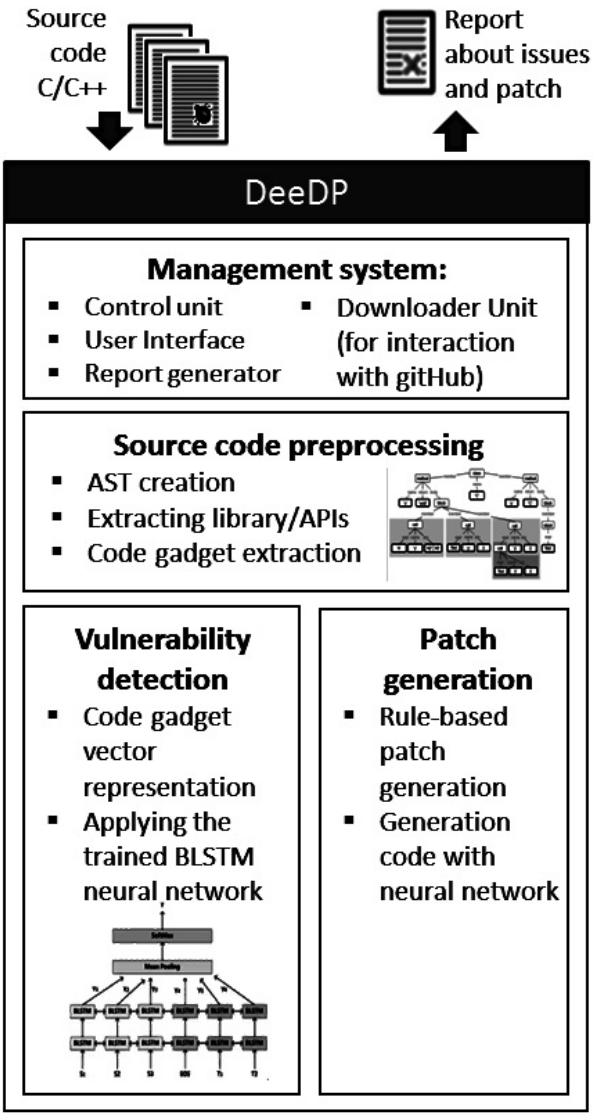

Fig. 1. General architecture of DeeDP

This paper discusses the automation technology of source code analysis for finding vulnerabilities and fixing detected errors. There are two main steps: vulnerability detection and patch generation figure 1.

A subsystem of DeeDP for vulnerability detection is based on deep learning approach and performs the following steps: source code preprocessing, AST creation, code gadget extraction, code gadget vectorization (word2vec), application of trained BLSTM neural network, preparation of human-readable report. We have pushed off the concepts for source code analysis which

are described in [14] and improved the steps with code representations.

DeeDP subsystem for patch generation performs transformation of code fragment (code gadget) which was identified as vulnerable with specific function to obtain a patch with improved code fragment. Transformation can be done with rule-based approach (specific AST transformation according to detected issue) and based on code generation with neural network (LSTM).

4. Patch generation based on rules

The procedure for generating patches based on rules consists of the four steps: collecting data from static analyzer; source code preprocessing, patch generation, final verification. Current investigation was targeted on the next list of weaknesses: division by zero pattern, closed resource patterns, double free pattern, out of bounds pattern, string buffer overflow pattern. Rule-based approach allows to apply specific transformation to source code represented as AST that relates to detected issue.

1) Collecting data from static analyzer: verification the source code with a static analyzer and detecting vulnerabilities/weaknesses. We used on this step VulDetect (our own solution for detection vulnerabilities based on deep learning approach [3] and SVACE [17]. Based on result analysis we are creating specific file with meta-information about issue location.
2) Source code preprocessing: building an AST representation of a code (with clang), extracting fragment of AST (code gadget) according to issue location. Code gadget – it is fragment of general AST representation of all source code, which includes detected issue and context of this function, based on data flow analysis. Extract useful information for each token in AST related to detected issue. After this we are transforming AST representation for the next step (replacing user names, functions etc.).
3) Patch generation: according to type of detected weaknesses we have specific rules (patterns) which determines how we have to transform AST for removing issue. After we are transforming the AST according to the pattern. The next step is converting the received AST into a patch and applying the patch to product.
4) Final verification: checking again the received code with static analyzer to determine if the weaknesses has been fixed and if new ones have been created.

```
Patterns format
• File path: src/patterns/[VUL_NAME]/[
    PATTERN_FILE]
• Patch pattern number (PPN) name format:
    1. Name of pattern: "ppn_" + #
    2. File format: .cpp
    3. In case several patterns for one
     vulnerability: add
    "a"/"b"/"c"/...
    (example: "ppn_1a.cpp", "ppn_1b.cpp")
• Requirements inside pattern:
    1. Entity name with vulnerability: ARG_1 or
      function name from CWE
    2. Entity type with vulnerability: TYPE_1
    3. Entity name for check: ARG_2
    4. Entity type for check: TYPE_2
    5. Variable name after vulnerability
     operation: ARG_3
    6. Variable type after vulnerability
     operation: TYPE_3
    7. Name for new variable: [NAME] + "
     _target_"
    (example: idx_target_, len_target_)
    8. Type for new variable: TYPE_4 or more
• Special characters:
    1. R - At start of line. Mean that need
     replace current line on line that contains
      vulnerability
    2. _ - Prefix for entity that marks it for
     deletion
    3. X - Some binary operator
```

5. Patch generation based on neural network

After building patches using rule-based approach we have understood that creating and applying patterns are monotonous and requires a lot of intensive manual labor, so we decided to automatize collecting and applying rule-based patterns. According to Microsoft SDL procedure there are may exists examples of good and bad usage of some target calls, so according to this examples we can generate some data set and fit Neural Network. Other data set can be generated from project source code, for example, most common usages of target calls, style patterns, etc.

First of all, idea was in replacing bad code to good code or adding some code for bad code become good code, so we need to replace some text on other. In one hand we can create bijection function (one-to-one correspondence) for replacing bad code to good code.

Eventually, we need to create application that can understand context of code and transform it to good code, so we tried to use method that translating sentences from one language to other with some modifications. One of standard methods of translating from one language to other is seq2seq model (encoder-decoder model). This model can be splitted up on two parts: -encoder and -decoder part. On figure 2 illustrated standard encoder-decoder architecture. Encoder takes a raw input text data just like any other RNN architectures do. At the end, Encoder outputs a neural representation ('thought' vector). The output of Encoder is going to be the input data for Decoder. Then Decoder transform neural representation to words.

All models vary in terms of their architecture. A natural choice for sequential data is the recurrent neural network (RNN). Usually an RNN is used for both the encoder and decoder parts [18]. The RNN models differ in some aspects:

- directionality: unidirectional or bidirectional
- depth: single or multi-layer
- type: Long Short-term Memory (LSTM), or a gated recurrent unit (GRU)

In this article, we used a single RNN which is unidirectional and uses GRU as a recurrent unit. figure

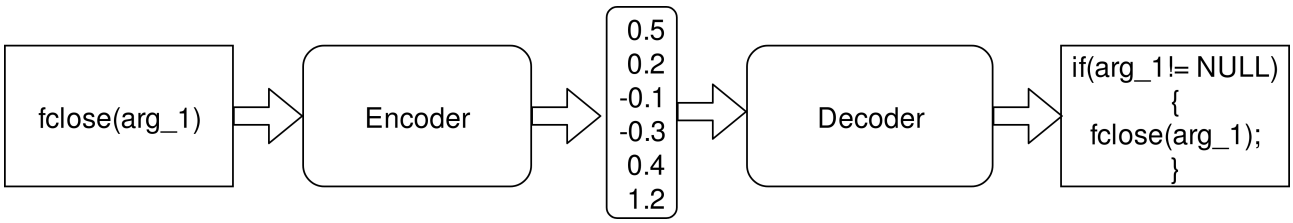

Fig. 2. Standard encoder-decoder architecture

3. Is an example of model that translate a source code «fclose(arg_1);» to «if (arg_1) fclose(arg_1);». Here «<s>» marks the start of the decoding process while «</s>» tells the decoder stop.

For creating more quality model we need to add attention mechanism [19]. The main idea of the attention mechanism is to establish direct short-cut connections between the target and the source by paying 'attention' to relevant source content as we translate. A nice byproduct of the attention mechanism is an easy-to-visualize alignment matrix between the source and target sentences.

In the simple seq2seq model we pass the last source state from the encoder to the decoder when starting the decoding process. This works well for short and medium-length sentences; but for long sentences, the single fixed-size hidden state becomes an information bottleneck. Instead of discarding all of the hidden states computed in the source RNN, the attention mechanism provides an approach that allows the decoder to peek at them (treating them as a dynamic memory of the source information). By doing so, the attention mechanism improves the translation of longer sentences. Nowadays, attention mechanisms are the defacto standard and have been successfully applied to many other tasks (including image caption generation, speech recognition, and text summarization).

The attention computation happens at every decoder time step. It consists of the following stages:

1) The current target hidden state is compared with all source states to derive attention weights.
2) Based on the attention weights we compute a context vector as the weighted average of the source states.
3) Combine the context vector with the current target hidden state to yield the final attention vector.
4) The attention vector is fed as an input to the next time step (input feeding).

6. Results

Training BLSTM neural network for vulnerabilities detection module was performed on dataset with more than 15000 code gadgets (code samples with presence of buffer overflow vulnerability, as well as samples with vulnerabilities associated with incorrect resource management). Training samples were created based on source codes taken from the National Vulnerability Database (NVD), and from the NIST Software Assurance Reference Dataset (SARD) [20].

Table 1. Confusion matrix of weaknesses detection

Name of metric	Counts	%
Total test code gadgets	6164	100%
True positive	4091	66%
True negative	1257	22%
Total true	5348	87%
False positive	524	8.5%
False negative	292	5%
Total false	816	13%

6.1. Weaknesses detection based on deep learning

Results of detection procedure were described deeply in previous paper [3], but we have updated our detector with retraining on expanded data set.

Training BLSTM neural network for vulnerability detection module was performed on dataset with more than 15000 code gadgets (code samples with presence of buffer overflow vulnerability, as well as samples with vulnerabilities associated with incorrect resource management). Training samples were created based on source codes taken from the National Vulnerability Database (NVD), and from the NIST Software Assurance Reference Dataset (SARD) [21]. Moreover, we improved step with converting code gadget representation to vector based on word2vec method. Result of estimation accuracy of weaknesses detection performed in the table 1.

6.2. Results of generation rule-base patches

Proposed technology was verified on different open source projects, for example, jsoncpp, Microsoft/Terminal, and others.

DeeDP detected a security issue with resource management in Microsoft/Terminal and created a patch which was successfully approved and applied by Microsoft maintainers: (https://github.com/microsoft/Terminal/commit/99555ef9e9ba89b03bbeedf238b7e65375775b56).

6.3. Results of generation patches base on neural network

We have performed test of approach for creation patch with seq2seq concept, when neural network itself extracts statistical dependency between code with weaknesses and improved code.

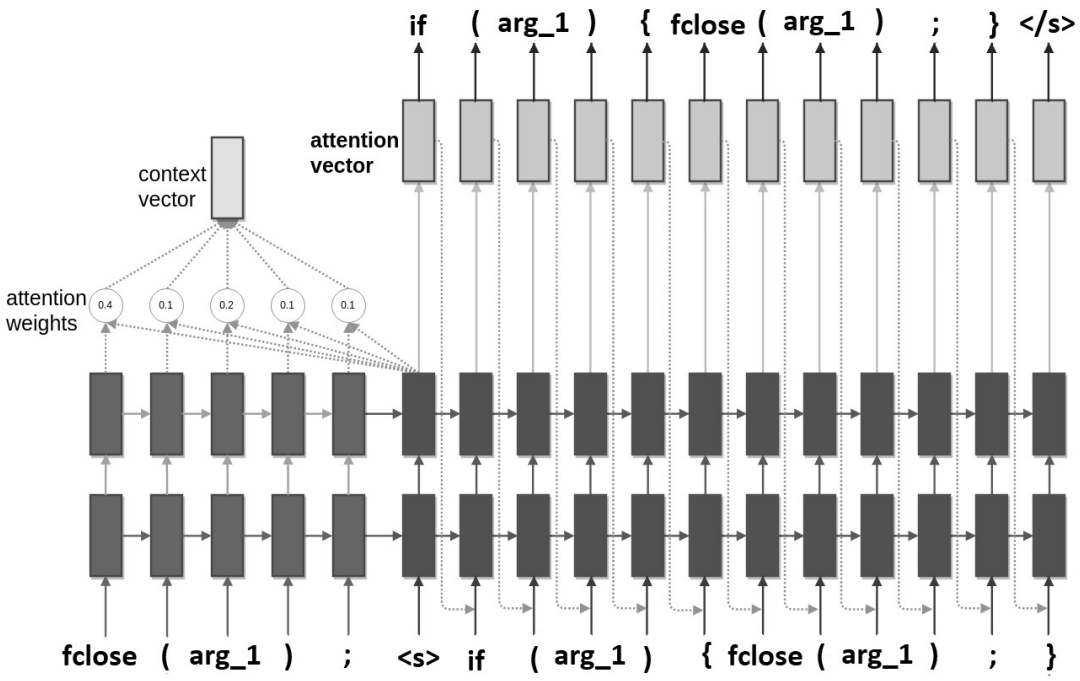

Fig. 3. Example of a deep recurrent architecture for transforming

5 ■■■■■ src/terminal/parser/ft_fuzzwrapper/main.cpp

```
@@ -70,7 +70,10 @@ int __cdecl wmain(int argc, wchar_t* argv[])
70  70              fGotChar = GetChar(&wch);
71  71          }
72  72
73      -       fclose(hFile);
    73  +       if (hFile)
    74  +       {
    75  +           fclose(hFile);
    76  +       }
74  77          wprintf(L"Done.\r\n");
75  78      }
76  79
```

Fig. 4. Example of generated automatic patch for Microsoft Terminal

Collect dataset. Our data set was collected from common usages of target call of Microsoft popular projects. Then, for removing user variables we replace by ARG_1, ARG_2, etc. Prepare dataset:

1) Add a «<start>» and «<end>» token to each sentence.
2) Clean the sentences by removing special characters.
3) Create a word index and reverse word index (dictionaries mapping from word → id and id → word).
4) Pad each sentence to a maximum length.

Build and train model:

1) Pass the «input» through the Encoder which return «encoder output» and the «encoder hidden state».
2) The encoder output, encoder hidden state and the decoder input (which is the «start token») is passed to the decoder.
3) The decoder returns the «predictions» and the «decoder hidden state».
4) The decoder hidden state is then passed back into the model and the predictions are used to calculate the loss.
5) Use «teacher forcing» to decide the next input to the decoder.
6) «Teacher forcing» is the technique where the «target word» is passed as the «next input» to the decoder.
7) The final step is to calculate the gradients and apply it to the optimizer and backpropagate.

Create patch

- The evaluate function is similar to the training loop, except we don't use «teacher forcing» here. The input to the decoder at each time step is its

previous predictions along with the hidden state and the encoder output.
- Stop predicting when the model predicts the «end token».
- And store the «attention weights for every time step».

```
transform(u'fclose(ARG_1);')
Input: <start> fclose ( arg_1 ) ; <end>
Predicted transform: if ( arg_1 ) {
    fclose ( arg_1 ) ; } <end>
==========

transform(u'FILE *ARG_1 = fopen(ARG_2,
    ARG_3);')
Input: <start> file * arg_1 = fopen (
    arg_2 , arg_3 ) ; <end>
Predicted transform: file * arg_1 =
    fopen ( arg_2 , arg_3 ) ; if ( arg_1
    == null ) return false ; <end>
==========

transform(u'FILE *ARG_1; ARG_1 = fopen(
    ARG_2, ARG_3);')
Input: <start> file * arg_1 ; arg_1 =
    fopen ( arg_2 , arg_3 ) ; <end>
Predicted transform: file * arg_1 ;
    arg_1 = fopen ( arg_2 , arg_3 ) ;
    if ( arg_1 == null ) { printf ( \
    unable to open file \ \ r \ \ n ) ;
    arg_3 = 1 ; return arg_3 ; } <end>
```

Listing 1. Examples of generated patches based on neural network

We have developed DeeDP system with web UI where in which direct links to GitHub can be pasted for analysis. After analysis the system shows results of vulnerability verification and suggested patches. The next steps of our investigation are expansion list of detectable vulnerabilities and improvement of patch generation techniques.

Conclusion

This paper describes approaches for automation of detection and fixing vulnerabilities in C/C++ source code based on different approaches. Presented rule-based approach for fixing issue shows robustness, but cannot be easily scaled. We implemented this approach first of all for baseline and collecting dataset, which will be used in the second method.

We have verified ability to use neural network for generation patch and got very good result. In future we are planning to expend set of weaknesses which can be detected with our VulDetect module and can be automatically fixed by Patch generation module. Moreover we want to use Generative adversarial network (GAN) for generation code without issue from primary source code. In general we need to add ability to continuously train both neural networks (for detection and for generation processes) on new appeared samples.

References

[1] Bruckner Daniel. Improved Vulnerability Detection using Deep Representation Learning. — 2019. — Access mode: https://blog.mi.hdm-stuttgart.de/index.php/2019/03/04/improved-vulnerability-detection-using-\deep-representation-learning/.

[2] Nunes Sheldon. Exploring the Abstract Syntax Tree. — 2018. — Access mode: https://dev.to/sheldonnunes/exploring-the-abstract-syntax-tree-2ce8.

[3] Artem Chrenousov Artem Savchenko Serhii Osadchyi Yevhen Kubiuk Yevhen Kostenko Dmytro Likhomanov. Deep learning based automatic software defects detection framework. — Department of Computer Science and Engineering, 2017. — Access mode: http://tacs.ipt.kpi.ua/article/view/169086.

[4] Zichao Qi Fan Long Sara Achour, Rinard Martin. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. — MIT EECS & CSAIL. — P. 24–36.

[5] Zhen Huang David Lie. Sound Patch Generation for Vulnerabilities. — Access mode: https://arxiv.org/pdf/1711.11136.pdf.

[6] Long Fan, Rinard Martin. Prophet - Automatic Patch Generation by Learning Correct Code. — Georgia Institute of Technology, Peking University, 2016. — Access mode: https://people.csail.mit.edu/fanl/papers/prophet-popl16.pdf.

[7] Long F., Rinard. M. SRP-Staged program repair with condition synthesis. — 2016. — Access mode: https://people.csail.mit.edu/fanl/papers/spr-fse15.pdf.

[8] GenProg. Evolutionary Program Repair. — 2019. — Access mode: https://squareslab.github.io/genprog-code/.

[9] Qi Yuhua. RSRepair - The Strength of Random Search on Automated Program Repair. — 2013. — Access mode: https://personal.utdallas.edu/~lxz144130/cs6301-readings/repair-qi-icse14.pdf.

[10] W. Weimer Z. P. Fry, Forrest. S. AE-Leveraging program equivalencefor adaptive program repair: Models and first results. — 2013.

[11] Yuhua Qi Xiaoguang Mao Yan Lei Ziying Dai, Wang Chengsong. The Strength of Random Search on Automated Program Repair. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). — New York, NY, USA, 2014. — P. 254–265. — Access mode: http://dx.doi.org/10.1145/2568225.2568254.

[12] Long Fan, Rinard Martin. Automatic Patch Generation by Learning Correct Code. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16). — ACM, New York, NY, USA, 2016. —

P. 298–312. — Access mode: https://doi.org/10.1145/2837614.2837617.

[13] Long F., Rinard M. Staged program repair with condition synthesis. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE. — New York, NY, USA, 2015. — P. 166–178.

[14] Z. Li D. Zou S. Xu X. Ou H. Jin S. Wang Z. Deng, Zhong Y. "VulDeePecker: A deep learning-based system for vulnerability detection," in Proceedings 2018 Network and Distributed System Security Symposium, Internet Society. — 2018. — P. 802–811. — Access mode: http://dl.acm.org/citation.cfm?id=2486788.2486893.

[15] Dongsun Kim Jaechang Nam Jaewoo Song, Kim Sunghun. Automatic Patch Generation Learned from Human-written Patches. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). — IEEE Press, Piscataway, NJ, USA, 2013. — P. 802–811. — Access mode: http://people.csail.mit.edu/hunkim/papers/kim-icse2013.pdf.

[16] Enumeration Common Weakness. Overview - What Is CWE? — 2019. — Access mode: https://cwe.mitre.org/about/.

[17] Developers Tizen. Static Analysis Tool (SVACE) Overview. — 2019. — Access mode: https://developer.tizen.org/community/tip-tech/how-process-potential-defects-static-\analysis-tool-using-public-jira-system.

[18] Dzmitry Bahdanau Kyunghyun Cho Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. — 2016. — Access mode: https://arxiv.org/pdf/1409.0473.pdf.

[19] Bahdanau Dzmitry. Neural machine translation by jointly learning to align and translate. — 2016. — Access mode: https://arxiv.org/pdf/1409.0473.pdf.

[20] Louis Kim Rebecca Russell. Draper VDISC Dataset - Vulnerability Detection in Source Code. — 2018. — Access mode: https://osf.io/d45bw/.

[21] NIST. Software Assurance Reference Datase. — 2006. — Access mode: https://samate.nist.gov/SARD/testsuite.php.