Research Article

Study on Software Vulnerability Characteristics and Its Identification Method

Chenlan Luo , Wang Bo, Huang Kun, and Lou Yuesheng

Yiwu Industrial & Commercial College, Yiwu 322000, China

Correspondence should be addressed to Chenlan Luo; baldwin930@protonmail.com

Received 2 August 2019; Revised 2 February 2020; Accepted 5 March 2020; Published 31 March 2020

Academic Editor: Vassilios Constantoudis

Copyright © 2020 Chenlan Luo et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

A method for identifying software data flow vulnerabilities is proposed based on the dendritic cell algorithm and the improved convolutional neural network to effectively solve the transmission errors in software data flow. In this method, we first gave the software data flow propagation model and constructed the data propagation tree structure. Secondly, we analyzed the running characteristics of the software, took the interaction among indexes into account, and identified data flow vulnerabilities using the dendritic cell algorithm and the improved convolutional neural network. Finally, we conducted an in-depth study on the performance of this method and other algorithms through mathematical simulation. The results show that this method has better advantages in detection time, storage cost, and software code size.

1. Introduction

With the rapid development of information technology, software system and its vulnerabilities have become a widespread concern for developers [1–4]. Software vulnerabilities are defects in software's specific implementation or system security policy, which can enable attackers to access or damage the system without authorization. It is mainly reflected in two aspects: (1) there are some inherent weaknesses or defects in the software system, and its execution can explicitly or implicitly cause damage to the system; (2) the worst unit to resist environmental disturbances in software systems. Case (1) refers to a defect in the software code itself. This type of defect can be used by some malicious users to cause various security issues, such as obtaining unauthorized permissions and interfering with the execution of key processes. Internet applications are more prominent. Various types of vulnerabilities have been discovered, such as SQL injection, buffer leaks, and format strings. Case (2) is for software systems running in harsh environments. Due to electromagnetic interference, voltage fluctuations, ultra-high and low-temperature environment, and other factors, even if the software system itself does not have any defects, unexpected situations may occur, most of

which are caused by errors introduced by environmental disturbances.

This paper mainly studies the software vulnerabilities under environmental disturbance. The occurrence of software faults can be avoided by detecting the data flow vulnerabilities which have two effects on the execution process of software system: one is that the system error is caused by the wrong signal input to the system by outside users when using the software, which may further change the internal state of the software. The other is that the internal state of the software is directly changed by the environment, and software faults may occur when the software continues to run in such an environment without error handling mechanism. To detect these faults accurately, it needs to analyze the data flow of the program, which is a tool to determine the relationship between the variable fixed value and the reference.

Software vulnerability refers to the safety-related design errors, coding defects, and operation faults in the software life cycle, and its cause is complex and difficult to analyze. As a key link in software vulnerability analysis, vulnerability positioning is very important for information security maintenance. Traditional information system software vulnerability assessment methods can be divided into rule-

based and model-based formalization methods [5–9]. Sanaz et al. [10] proposed a new paradigm for vulnerability discovery prediction based on code properties and have studied the impact of code properties on the vulnerability discovery trends by performing static analysis on the source code of four real-world software applications. Huang et al. [11] proposed a new automatic vulnerability classification model with term frequency-inverse document frequency and deep neural network. It is found that, compared to SVM and Naive Bayes, this model has achieved better performance in multidimensional evaluation indexes including accuracy, recall rate, and precision. Alhazmi et al. [12] used several major operating systems as representatives of complex software systems to analyze the data on vulnerabilities discovered in these systems. The results indicate that the values of vulnerability densities fall within a range of values and reveals that it is possible to model the vulnerability discovery using a logistic model that can sometimes be approximated by a linear model. Jeffrey et al. [13] compared models based on the software metrics and term frequencies and explored the role of dimensionality reduction through a series of crossvalidation and crossproject prediction experiments. The results showed that, in the case of software metrics, a dimensionality reduction technique based on confirmatory factor analysis provided an advantage when performing crossproject prediction, yielding the best F-measure for the predictions in five out of six cases. Georgios et al. [14] proposed a model with texts analysis and multitarget classification techniques, and it could estimate the vulnerability characteristics and subsequently and calculate the vulnerability severity scores from the predicted characteristics. Huang et al. [15] presented an analysis model which could express the security grades of software vulnerability and serve as a basis for evaluating danger level of information program or filtering hazardous weaknesses of the system and improve it to counter the threat of different danger factors, and it will be able to analyze the various degrees to which the vulnerability is affecting the security and this information will serve as a basis for future ameliorations of the system itself. Zhang et al. [16] described a novel approach to overcome the limitation of traditional dynamic taint analysis by integrating static analysis into the system and proposed a framework to detect software vulnerabilities with high code coverage.

However, the identification method of software data flow vulnerabilities has low prediction accuracy. Meanwhile, due to the complexity of the calculation process, the evaluation indexes cannot accurately reflect the tolerance capacity of the system in the process of processing. For this purpose, the identification method of software data flow vulnerabilities is established in this paper using the dendritic cell algorithm and the improved convolutional neural network on the basis of software data flow propagation model, and its effectiveness is finally verified through simulation experiments.

2. Software Data Flow Propagation Model

A software system usually consists of multiple interacting modules, and each module can contain multiple inputs and

outputs and is connected by parameter passing, message passing, memory sharing, etc. Therefore, when the input signal or intermediate signal goes wrong, it will be passed among modules along the path of signal propagation, which will thus lead to software errors. If the input signal i of the module A goes wrong, the error probability $PEP_{i,j}^{A \rightarrow B}$ of the output signal j from module A to module B is the error permeability of the signal i to the signal j . The calculation formula is as follows:

$$PEP_{i,j}^{A \rightarrow B} = P \left(\frac{\text{error occurred when outputting the signal } j}{\text{error occurred when inputting the signal } i} \right). \quad (1)$$

The permeability based on signal errors can be used to analyze the effect of a signal on system output, but the signal errors can be propagated to the system output through many paths. As a result, the following data propagation tree is constructed to locate the data flow vulnerabilities quickly and conveniently:

- (1) A system output signal is selected as the root node of the data propagation tree.
- (2) It needs to determine which system module the system output signal comes from and take each input of the module as a subnode of the root node.
- (3) If the signal corresponding to each subnode is not the input signal of the system, it needs to continue to track the input signal of each subnode and then continue to execute step (2) to find the next subnode of each subnode until all the leaf nodes of the propagation tree are found. If the corresponding signal is the input signal of the system, the node does not have any subnode and becomes a leaf node of the propagation tree.
- (4) If there are other system output signals, it needs to return to step (1) and continue to generate the propagation tree.

By generating a propagation tree, a corresponding propagation tree is generated for each output signal. When an error occurs in an output signal, all vulnerabilities where the signal could go wrong can be found along the propagation tree. In the tree structure, the root node represents the output of the system, the leaf node represents the input of the system, and the intermediate node represents the internal output of the system. Each path has a corresponding signal error permeability which can be judged by calculating the weight of each path in the tree structure, that is, the larger the path weight is, the higher the signal error permeability will be and the more data flow vulnerabilities will be on this path. In the actual calculation, the error propagation rate of each path is calculated as follows:

$$PEP_{\text{all}} = \prod_n PEP_{i,j}^{A \rightarrow B}. \quad (2)$$

In the actual signal propagation, the input signal and intermediate signal input may be correct, but the output signal is wrong. This is because there is a signal impact

factor in the system, which makes the correct input signal go wrong and then leads to the output signal error. The impact factor of input signal or intermediate signal S on the output signal O_{sys} of the system is calculated as follows:

$$S \longrightarrow O_{\text{sys}} \prod_k (1 - \omega_k), \quad (3)$$

where ω_k is the weight on the path k from the signal S to the system output O_{sys} . The larger the signal impact factor, that is, the more complex the environment is, the higher the error probability in signal propagation will be. Formula (3) only considers one system output, but when there are multiple system outputs, the impact factor will be calculated using formula (3) according to the importance of system output, that is, the impact factors of the output signals on multiple paths are multiplied by the output importance, and then multiplied by each other.

In addition to signal error propagation, there is module error propagation. If the module A has m input signals and the module B has n output signals, then the error permeability $\text{PEP}^{A \rightarrow B}$ of module A to module B is calculated as follows:

$$\text{PEP}^{A \rightarrow B} = \frac{1}{m \cdot n} \sum_i \sum_j \text{PEP}_{i,j}^{A \rightarrow B}. \quad (4)$$

In addition to input errors among modules, there may also be correct input but incorrect output under the influence of external environment. Therefore, the leakage rate PEL of the module is defined to calculate the error probability in this case. Specific calculation is as follows:

$$\begin{aligned} \text{PEL}_s^X &= P(s \neq s' \mid Z_X \neq Z_X'), \\ \text{PEL}^X &= \frac{1}{n} \sum_k \text{PEL}_k^X, \end{aligned} \quad (5)$$

where PEL_s^X is the probability that the output signal s of the module X varies with the internal state conditions, that is, the leakage rate of the module X . y and y' are the outputs of the module X , and Z_X and Z_X' are the internal states of the module X . PEL^X is the overall leakage rate of the module X when it has multiple outputs, and n is the number of outputs of the module X .

A module may be leaked in the running state, and each module has its specific running time. Assuming that in a normal operation of the software, the probability that the module D is in the running state is called the activity ratio PA^D of the module D :

$$\text{PA}^D = \frac{T_D}{T_{\text{all}}}, \quad (6)$$

where T_D is the running time of software module D and T_{all} is the total running time of the software.

There are module impact factors among modules, which are used to indicate the error probability of the module. If the module D has i outputs, then the module impact factor is defined as follows:

$$D \longrightarrow O_{\text{sys}}(i) = 1 - \prod_k (1 - \text{PEL}_k^D \cdot \text{PA}^D \cdot (O_D(k) \longrightarrow O_{\text{sys}}(i))). \quad (7)$$

3. Identification Method of Software Data Flow Vulnerability

Dendritic cell algorithm is an algorithm based on innate immunity without dynamic learning, and pattern matching is not required in the detection process. Although the detection speed is fast and the detection accuracy is high, data flow vulnerabilities will be identified using the improved dendritic cell algorithm as there are flaws in signal selection and processing. The main steps of the algorithm include initialization stage, updating stage, and comprehensive evaluation stage. The running characteristics of the software are analyzed in this paper, and the interaction among indexes is also taken into account, and the identification indexes of software data flow vulnerabilities are abstracted as follows:

- (1) Module impact factor: if a module has a higher module impact factor, it indicates that once an error occurs in the module, it will have a high probability of causing damage. Wrong data from this module will cause more harm to the system, and it is easier to generate abnormal data points.
- (2) Buffer overflow rate: if the input data exceeds the maximum amount of data that the buffer can hold, a buffer overflow occurs. The overflowed data flows into the memory area adjacent to the buffer, which would overwrite and modify the value in the adjacent memory area.
- (3) Module error permeability: the higher the module error permeability is, the lower the fault-tolerant ability of the module will be, which will lead to the probability of subsequent module errors.

Assume that a program a of the software C is divided into several modules, and each module has at least one input and output. The location of the assignment statement in the program is called a fixed value point which is represented by a binary group (P, L) , in which P is the variable to be valued and L is the number of program module where the fixed value point is located. The location of the reference variable in the program is called a reference point. The improved dendritic cell algorithm is now used to track the fixed value variables and search for data flow vulnerabilities. However, only the software vulnerability identification method based on the dendritic cell algorithm has an unsatisfactory convergence speed, a poor generalization ability, and a high false positive rate. Therefore, this article would identify vulnerability with an improved convolutional neural network and a dendritic cell algorithm.

Specific algorithm flow is as follows:

Step 1: the parameters are initialized. The input parameters PAMP, DS, etc. of the algorithm are initialized, and relevant indexes are mapped as follows: (i)

PAMP: pathogen-associated molecular pattern, which shows that the system is in an unhealthy state, and there are abnormal data points, and data passing cannot be performed normally. The occupancy of each software module and the software monitoring state are abstracted as PAMP signals, that is, $PAMP = \{<P, L>\}$. (ii) DS: danger signal, which is used to describe the possibility of data anomalies in software, $DS = \{\text{module impact factor, PEP, PEL}\}$. n modules are collected, and each module C_i contains m attributes, i.e., $C_i = (P_1, P_2, \dots, P_m)$, and the normal level of the attribute is measured by the attribute mean of n antigens. The abnormality T_j of the j^{th} attribute of the module i is calculated as follows:

$$\bar{P} = \frac{\sum_{j=1}^m (P_j - (\sum_{i=1}^n P/n))}{m},$$

$$T_j = \begin{cases} P - \bar{P}, & (P > \bar{P}), \\ 0, & (P \leq \bar{P}). \end{cases} \quad (8)$$

Set DS as the safety signal, which shows that the probability is higher, and the software is in a healthy state, $DS = \{P, L, \text{module impact factor, PEP, PEL}\}$. Input signal concentrations C_{PAMP} , C_{DS} , and C_{SS} are abstracted from dataset.

Step 2: the improved abstracted input signals C_{PAMP} , C_{DS} , and C_{SS} are trained using an improved convolutional neural network model. The improved convolutional neural network model training includes two parts: forward propagation process and back-propagation process.

Step 2.1: forward propagation training.

- (i) Training is performed in batches in the system. Each training randomly selects a fixed-size batch from the preprocessed input signal set as an input through two convolutional layers and uses the convolutional layer, and the multiconvolution kernel performs convolution operations and feature extraction on all feature values of the initial input:

$$y_j^l = \sigma \left(\sum_{i \in M_j} y_j^{l-1} w_{ij}^l + b_i^l \right), \quad (9)$$

where y_j^l represents the result obtained by processing each convolution layer through a convolution kernel, l represents the number of convolution layers, j represents the serial number of the convolution kernel, σ is an activation function, w represents weight, and b is offset.

- (ii) Processed by the crosslayer aggregation module, for each convolution kernel output feature of the convolutional layer, it will be down sampled by the pooling layer to reduce the dimension and extract the region-maximized feature of the data. The pooling calculation formula is

$$y_j^l = \beta(w_j^l \text{down}(y_j^{l-1}) + b_j^l), \quad (10)$$

where down represents the down-sampling operation on matrix elements and β is a pooling function.

- (iii) Perform a merge operation on the output results to generate a dataset, and perform data classification at Softmax layer.

Step 2.2: backpropagation training.

- (i) Softmax layer was used to classify the training set and samples to calculate the overall error parameter value.
- (ii) Backpropagation is performed based on the error parameter value. Iteratively adjust the weights and offsets of the layers of the network according to the error between the actual output value and the ideal output value until the model achieves a good convergence effect. Here, the loss function $C(w, b)$ is established according to the Stochastic Gradient Descent method:

$$C(w, b) = \frac{1}{2n} \sum_x y^{(x)} - a^2, \quad (11)$$

where n is the number of training input data and a represents the vector output when the input is x .

Step 3: the evaluation factor ω of the subsequent module to the current module is calculated to adjust the data tracking direction:

$$\omega(n) = \frac{\sum_{i=1}^n \text{Aff}(n, n+i)}{n}, \quad (12)$$

$$\text{Aff}(n, m) = \sqrt{\sum_{i=1}^n T_n(A_i) - T_m(A_i)},$$

where k denotes the extension of k modules backward, $T_n()$ denotes the attribute of the n^{th} module, and $\text{Aff}(n, m)$ denotes the similarity between the n^{th} module and the m^{th} module.

Step 4 (signal collection and processing): DC collects antigens in the tree according to the established signal transmission tree model, processes three kinds of input signals, i.e., PAMP, DS, and SS, which is trained with the convolutional neural network, and calculates their output signals. The output signal is calculated as follows:

$$C_{\text{out}} = \frac{(W_{PAMP} \times C_{PAMP}) + (W_{ss} \times C_{ss}) + W_{DS} \times C_{DS}}{|W_{PAMP}| + |W_{ss}| + |W_{DS}|}, \quad (13)$$

where C_{PAMP} , C_{DS} , and C_{SS} denote input values and W_{PAMP} , W_{DS} , and W_{SS} denote the weights corresponding to output values.

Step 5 (comprehensive algorithm evaluation): MCAV is defined to finally evaluate the data flow information and calculated as follows:

$$\text{MCAV} = \frac{\text{Count}}{\text{Total}}, \quad (14)$$

where Count denotes the number of anomalies in data flow and Total denotes the total number of times the data flow is detected. The larger the MCAV value is, the higher the abnormal degree in the data will be.

Step 6: if there is a new module, then it needs to go to Step 1, or the algorithm is terminated.

4. Simulations

Our algorithm is now compared with the results of the traditional detection method to verify the effectiveness of data vulnerability identification method proposed in this paper. Several commonly used software are selected for testing, and the number of software vulnerabilities extracted before and after using this method is counted, respectively (see Table 1, for test results). The hardware environment used in the test is Windows 7, 4 GB memory, Pentium Dual-Core CPU, and MATLAB is adopted for statistical analysis.

As can be seen from Table 1, the number of data flow vulnerabilities identified by our algorithm has increased significantly so that the scope of code analysis is greatly reduced for vulnerability explorers, and the analysis objectives are more clear, and the efficiency of vulnerability mining is improved, and the software is well modified in the process of testing and running to greatly improve the development efficiency. Classical Error Propagation Analysis (EPA) is compared with our method in this paper. Figure 1 shows the comparison of detection time between EPA and our algorithm in each time period, in which the detection time period is 4 min, that is, each time period is 4 min (see Figure 1, for comparison results). As can be seen from Figure 1, the detection time of our algorithm is lower than that of EPA algorithm, which greatly improves the detection efficiency. In addition, it can be seen from the figure that as the time goes on, the detection time also increases. This is because the data propagation path is random, so the data tracking process takes longer and longer in the later stages.

Secondly, Figure 2 shows the comparison of storage cost between EPA and our algorithm. The detection time period in Figure 2 is 4 min, that is, each time period is 4 min. As can be seen from Figure 2, the memory space occupied by our algorithm is much less than that of EPA algorithm, which will greatly improve the CPU's operation mechanism and make the detection more efficient and accurate. In addition, EPA algorithm will constantly consume the memory space over time, which will increase the memory burden and easily lead to data overflow. It can be seen that our algorithm has higher superiority and can greatly reduce the detection cost.

Finally, the memory overflow detection rate is defined here as the success rate to detect a memory overflow in the system. Figure 3 shows the relationship between the memory overflow detection rate and storage of the two algorithms. As can be seen from Figure 3, with the increase of storage, the detection rate of memory overflow shows a decreasing trend. This is because the storage is increased, the amount of data

TABLE 1: Plug-in running results.

Software	Version	Before use	After use
Excel.exe	15.0.4420.1017	1920	7532
QQ.exe	8.7.19113.0	2041	9275
JisuPdf.exe	2.1.2.1001	401	1226
VISIO.exe	11.0	593	2618
360zip.exe	3.2.0.2110	102	295

FIGURE 1: Comparison of detection time.

FIGURE 2: Comparison of storage.

that can be received in the buffer is increased, and the possibility of data overflow is reduced, which shows that the detection rate of memory overflow is reduced. In addition, when the storage is small, the difference between the memory overflow detection rates of the two algorithms is small, and as the storage space is increased, the memory overflow detection rates of the two algorithms are greatly different. This further validates the effectiveness of the algorithm in this paper.

FIGURE 3: Relationship between memory overflow detection rate and storage.

5. Conclusions

A method for identifying software data flow vulnerabilities is proposed based on the dendritic cell algorithm and the improved convolutional neural network to effectively solve the transmission errors in software data flow. In this method, we first gave the software data flow propagation model and constructed the data propagation tree structure. Secondly, we analyzed the running characteristics of the software, took the interaction among indexes into account, and identified data flow vulnerabilities using the dendritic cell algorithm and the improved convolutional neural network. Finally, we conducted an in-depth study on the performance of this method and other algorithms through mathematical simulation. The results show that this method has better advantages in detection time, storage cost, and software code size. In the follow-up study, we can consider combining multiple performance indexes to improve the data flow vulnerability identification model.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported by the Philosophy and Social Sciences Planning Project of Zhejiang Province under Grant no. 20NDJC230YB and Foundation of Yiwu Industrial and Commercial College under Grant no. ZD2020JD331-01.

References

- [1] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.
- [2] X. Wang, R. Ma, B. Li, D. Tian, and X. Wang, "E-WBM: an effort-based vulnerability discovery model," *IEEE Access*, vol. 7, pp. 44276–44292, 2019.
- [3] S. S. Murtaza, W. Khreich, A. Hamou-Lhadj, and A. B. Bener, "Mining trends and patterns of software vulnerabilities," *Journal of Systems and Software*, vol. 117, pp. 218–228, 2016.
- [4] J. Ruohonen, S. Rauti, S. Hyrnsalmi, and V. Leppänen, "A case study on software vulnerability coordination," *Information and Software Technology*, vol. 103, pp. 239–257, 2018.
- [5] V. Smyth, "Software vulnerability management: how intelligence helps reduce the risk," *Network Security*, vol. 2017, no. 3, pp. 10–12, 2017.
- [6] H. Holm, M. Korman, and M. Ekstedt, "A Bayesian network model for likelihood estimations of acquirement of critical software vulnerabilities and exploits," *Information and Software Technology*, vol. 58, pp. 304–318, 2015.
- [7] D. Zhang, D. Liu, C. Csallner, D. Kung, and Y. Lei, "A distributed framework for demand-driven software vulnerability detection," *Journal of Systems and Software*, vol. 87, pp. 60–73, 2014.
- [8] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "Tracing known security vulnerabilities in software repositories - a Semantic Web enabled modeling approach," *Science of Computer Programming*, vol. 121, pp. 153–175, 2016.
- [9] M. Kimura, "Software vulnerability: definition, modelling, and practical evaluation for e-mail transfer software," *International Journal of Pressure Vessels and Piping*, vol. 83, no. 4, pp. 256–261, 2006.
- [10] S. Rahimi and M. Zargham, "Vulnerability scrying method for software vulnerability discovery prediction without a vulnerability database," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 395–407, 2013.
- [11] H. Huang, Y. Li, Q. Wang, J. Ren, Y. Cheng, and X. Zhao, "Automatic classification method for software vulnerability based on deep neural network," *IEEE Access*, vol. 7, pp. 28291–28298, 2019.
- [12] O. H. Alhazmi, Y. K. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *Computers & Security*, vol. 26, no. 3, pp. 219–228, 2007.
- [13] J. Stuckman, J. Walden, and R. Scandariato, "The effect of dimensionality reduction on software vulnerability prediction models," *IEEE Transactions on Reliability*, vol. 66, no. 1, pp. 17–37, 2017.
- [14] G. Spanos and L. Angelis, "A multi-target approach to estimate software vulnerability characteristics and severity scores," *Journal of Systems and Software*, vol. 146, pp. 152–166, 2018.
- [15] C.-C. Huang, F.-Y. Lin, F. Y.-S. Lin, and Y. S. Sun, "A novel approach to evaluate software vulnerability prioritization," *Journal of Systems and Software*, vol. 86, no. 11, pp. 2822–2840, 2013.
- [16] R. Zhang, S. Huang, Z. Qi, and H. Guan, "Static program analysis assisted dynamic taint tracking for software vulnerability discovery," *Computers & Mathematics with Applications*, vol. 63, no. 2, pp. 469–480, 2012.