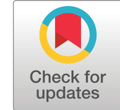
Journal of
Quantum Computing

Tech Science Press

DOI: 10.32604/jqc.2022.031312
Article



A Cross Language Code Security Audit Framework Based on Normalized Representation

Yong Chen[1,*], Chao Xu[1], Jing Selena He[2] and Sheng Xiao[3]

[1]School of Information Engineering, Nanjing Audit University, Nanjing, 211815, China
[2]Department of Computer Science, Kennesaw State University, Kennesaw, 30144-5588, USA
[3]Information Science and Engineering Department, Hunan First Normal University, Changsha, 410205, China
*Corresponding Author: Yong Chen. Email: chenyong@nau.edu.cn
Received: 14 April 2022; Accepted: 06 March 2023; Published: 15 May 2023

Abstract: With the rapid development of information technology, audit objects and audit itself are more and more inseparable from software. As an important means of software security audit, code security audit will become an important aspect of future audit that cannot be ignored. However, the existing code security audit is mainly based on source code, which is difficult to meet the audit needs of more and more programming languages and binary commercial software. Based on the idea of normalized transformation, this paper constructs a cross language code security audit framework (CLCSA). CLCSA first uses compile/decompile technology to convert different high-level programming languages and binary codes into normalized representation, and then uses machine learning technology to build a cross language code security audit model based on normalized representation to evaluate code security and find out possible code security vulnerabilities. Finally, for the discovered vulnerabilities, the heuristic search strategy will be used to find the best repair scheme from the existing normalized representation sample library for automatic repair, which can improve the effectiveness of code security audit. CLCSA realizes the normalized code security audit of different types and levels of code, which provides a strong support for improving the breadth and depth of code security audit.

Keywords: Code security audit; normalization; cross language; security vulnerabilities

1 Introduction

With the rapid development of information and intelligence, emerging technologies such as mobile Internet, big data, blockchain and artificial intelligence have been widely used in all aspects of people's production and life. Various software systems have become the main undertaker of business flow of enterprises and institutions. The security of business systems directly affects the security of economic activities. As a supervision mechanism, audit is responsible for timely early warning of the risks of economic activities of enterprises and institutions. How to timely find the security vulnerabilities of

This work is licensed under a Creative Commons Attribution 4.0 International License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

business systems for early warning will be an important aspect of audit in the new era. In addition, the audit work itself will also adapt to the trend of the times and change. Auditor General Hu Zejun of the National Audit Office pointed out: "We must adhere to strengthening audit through science and technology, innovate traditional audit methods, strengthen information infrastructure, make better use of Internet technology and information means to carry out audit, ask for resources from information and efficiency from big data, improve audit supervision, process control and decision-making support capabilities through information, digitization and networking, and actively promote full audit coverage." Big data audit and blockchain audit will become the trend [1–3]. In the future audit work, there will be many audit bases (i.e., audit evidence) And the audit process will depend on all kinds of audit systems. If the reliability and security of the audit system itself have problems and are attacked and used by hackers, it is bound to have a significant impact on the audit conclusion. Therefore, the audit for software system security, whether the audit object or the audit itself, is an urgent problem to be solved.

As we all know, software is generated by computer code. According to Gartner statistics, about 75% of the hacker attacks on software systems are attacks against code vulnerabilities. The security of code directly affects the reliability of software. For code security audit, researchers have formulated a series of coding rules for mainstream programming languages C and C++ [4,5], proposed targeted methods such as top-down, bottom-up and logical path coverage, and developed proxy security audit tools such as fortify SCA, find bugs, PMD and check style [6]. In recent years, with the rise of smart contracts, the code security audit technology for smart contract vulnerabilities has attracted extensive attention of scholars, and gradually proposed smart contract code security audit methods based on feature code matching, formal verification and symbolic execution [7]. In November 2020, the National Information Security Standardization Technical Committee issued the code security audit specification of information security technology, which stipulates the code security audit process, typical audit indicators and corresponding verification methods such as security function defects, code implementation security defects, resource use security defects and environmental security defects, so as to provide evidence for code security audit.

However, most of the existing code security audit methods are aimed at the source code of one or several common high-level programming languages. Even the code security audit specification of information security technology only expounds the relevant provisions of code security audit from the two common high-level programming languages, C language and Java language. However, the source code based security audit method has the following disadvantages.

a) There are many programming languages for writing software code. For each high-level programming language, we need to design the corresponding code security audit software separately. The development workload of code security audit related tools is large and the universality is not high. With the continuous expansion of software application scope, software developers will choose the most appropriate programming language for development according to specific business needs, such as smart contract in blockchain: bitcoin is developed in bitcoin script or ivy, Ethereum is developed in Solidity, Serpent, Mutan or LLL, and Hyperledger is developed in Go, Java or Nodejs. At present, there are more than 2500 high-level programming languages, and there will be more and more programming languages in the future. If each programming language designs a special code security audit tool, its workload will be huge.
b) Due to the different syntax of different high-level programming languages, it is difficult to synthesize different high-level programming languages for comprehensive analysis. Big data technology has been successful in many fields, which has greatly promoted the process of human intelligence. In the field of code security audit in the future, big data technology

will also become an innovative technology. However, big data technology is a data-driven technology, and the basic premise is to have as many sample data as possible. Although the open source movement of software enriches the software code samples, researchers can obtain the source code and development process information of some projects from hosting websites (such as Google Code, GitHub, SourceForge, etc.), but most of which is still the source code of mainstream programming languages. For the minority languages or new programming languages, the sample source code is very limited. Therefore, if the code security audit model based on big data technology, these new programming languages and minority languages will be greatly limited.

c) It is difficult to analyze with commercial application software. Limited by business privacy, it is generally difficult for people to obtain a large number of source codes in practical business applications. However, these non open source codes are often the core business of enterprises. If there are security vulnerabilities in this part of the code, it will be more destructive. In fact, code security audit should pay more attention to this part. However, the current source code based code security audit methods can only conduct independent audit by signing relevant confidentiality contracts, which is difficult to integrate similar products of different commercial companies for comprehensive audit, so the audit results will also be limited.

In fact, the code written in each programming language is not run directly on the computer, but through the processing of software construction program, and finally form the software that can run on the computer. For software builders, such as compilers, it is usually divided into front-end and back-end. The front end can convert different high-level programming languages into a unified intermediate representation with semantic equivalence. If we conduct code security audit based on this unified intermediate representation, we will obtain a more universal code security audit tool. Therefore, guided by the idea of program normalization transformation, this paper constructs a cross language code security audit framework CLCSA (cross language code security audit framework), which provides strong support for the implementation of broader and more universal code security audit.

The outline of this paper is as follows. In the next section, we describe the background and related work, and Section 3 describes our proposed CLCSA approach. We conclude the paper and highlights future directions in the last section.

2 Background and Related Work

Code Auditing provide objective, reproducible and quantifiable measurements about a software product, and researches have studied it for a long time. Daskalantonakis categorize software metrics based on their intended use as product, process and project metrics [8].

2.1 Product Metrics Prediction

The product metrics focuses on the attributes of code size and intrinsic complexity of source code. The underlying assumption is that program modules with higher code size or complexity have a higher probability of containing defects inside. Researchers first used lines of code (LOC) to measure. Akiyama [9] gives the relationship between the number of defects (D) and LOC (L): $D = 4.86 + 0.018L$. However, the measurement element is too simple to measure the complexity of software system reasonably. Subsequently, researchers gradually considered Halstead's scientific metrics [10] and McCabe's cyclomatic complexity [11]. Halstead's scientific metrics measure the reading difficulty of code by counting the number of operators and operands within the program. Its hypothesis is that the higher the difficulty of reading the code, the higher the probability of containing defects. The

main measurement elements involved in Halstead metrics include the length, capacity, difficulty and workload of the program. McCabe loop complexity is concerned with the control flow complexity of the program. Its assumption is that the higher the complexity of the program's control flow, the more likely it is to have defects.

With the popularity of object-oriented development methods, Chidamber et al. proposed the CK metric element [12], which considers the characteristics of inheritance, coupling and cohesion in object-oriented programs. Based on some medium-sized information management systems, Basili et al. [13] first verified the relationship between CK metrics and defects in program modules. Then, Subramanyam et al. [14] further validated the findings of Basili et al. based on eight industrial projects. Zhou et al. [15] also conducted an in-depth analysis of the correlation between metrics based on object-oriented programs and program module defects. Furthermore, they found that class size metrics have potential mixed effects in analysis and will influence the performance of defect prediction model [16]. Therefore, they proposed a method based on linear regression to try to remove this mixed effect. Finally, they conducted an in-depth analysis of the correlation between the package-modularization metrics proposed by Sarkar et al. [17] and the cohesive metrics based on program slices and program module defects [18].

Zimmermann et al. predict [19] the number of defects after module release by analyzing the dependencies in low level programs. They analyze data dependencies and calling dependencies between binaries and modeling subsystems as dependency graphs. Experiments have shown that there are correlations between the dependency graph and the number of defects.

2.2 Process Metrics Prediction

Software defects are not only related to the internal complexity of the program module, but also closely related to code modification features, developer experience and dependencies between modules. In recent years, with the development of software history warehouse mining, it is possible to analyze software development process and design metrics related to the above factors.

In code modifications, Nagappan et al. [20] present a technique for early prediction of system defect density using a set of relative code churn measures that relate the amount of churn to other variables such as component size and the temporal extent of churn. Experiments show that their set of relative measures of code churn is highly predictive of defect density.

3 CLCSA Framework

CLCSA framework is mainly based on programming language theory, and integrates software reverse engineering, machine learning and other technologies to provide strong support for more universal and accurate code security audit results. The framework mainly includes three modules: software code normalization processing module, code security audit model construction module, and code security audit countermeasure suggestion model construction module, as shown in Fig. 1.

First, with the help of programming language theory, using program analysis, pattern matching and other technologies, the source code and binary programs of various languages are transformed into normalized representation based on compiler intermediate representation (IR); Then the normalized code is analyzed by graph neural network. According to the two eight rule of the program, unsupervised learning method is used to extract multi-dimensional program defect features, and then integrated learning is used to build a domain adaptive defect prediction model. At the same time, for each defect feature, according to the results of code clustering analysis, hash index and heuristic search methods are used to construct the normalized repair template and scheme selection strategy. Finally,

with the help of reverse engineering, it is converted into the source program or binary repair patch of the corresponding language. So as to realize the self-repair of software defects.

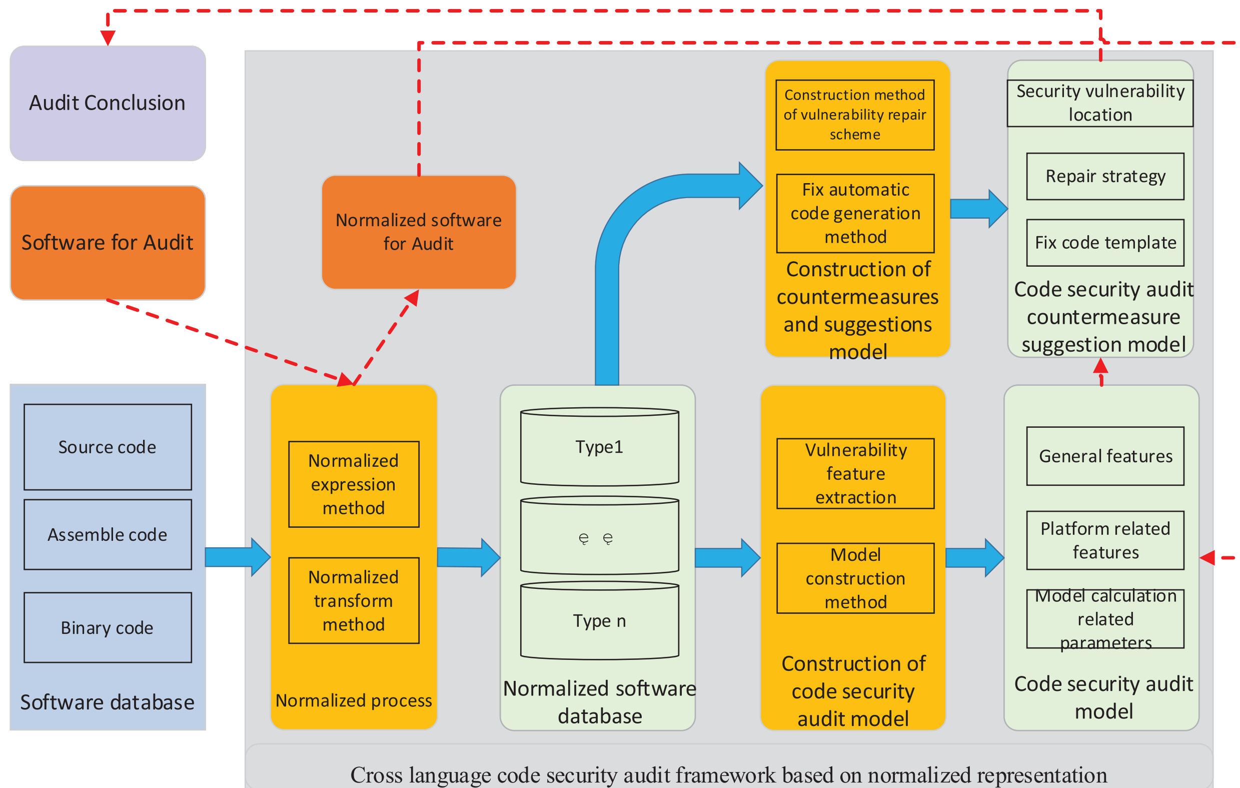


Figure 1: The outline of the CLCSA

3.1 Software Code Normalization Processing Module

The software code normalization processing module is to realize the normalization processing of the code, which is used to normalize the codes of different languages and different levels into the same level and the same form of code, and provide unified input for the code security audit model. In order to bring a large number of closed source commercial software into the scope of code security audit, the software code normalization processing module can not only convert the source code of different high-level programming languages into normalized representation, but also use decompilation technology to convert the closed source binary code into normalized representation. At the same time, due to different software application fields, the focus of code security audit may also be different. Therefore, the result code of software code normalization processing will be classified and stored according to the application field of software.

The software code normalization processing module is the basis of CLCSA framework, which mainly includes two aspects: cross language software normalization representation method and cross language software normalization conversion method. The normalized representation method of cross language software is a normalized form used to represent different levels of programming languages such as high-level programming language, assembly language and binary program on the premise of maintaining program semantics. The normalization conversion method of cross language software is the process of realizing the conversion from different levels of code to normalized code. Because the

front end of the compiler can well convert the source programs of different high-level programming languages into the intermediate representation of the compiler, and there are disassembly tools that can convert binary code into assembly code, which is generated by the intermediate representation of the compiler. Therefore, CLCSA framework first designs the normalized code representation based on the current mainstream LLVM IR. Then, aiming at the normalized representation, the high-level programming language is extended by LLVM compiler; For binary code, CLCSA framework converts binary program into assembler through disassembly technology, and then generates preliminary normalization conversion results based on pattern matching according to normalization conversion template and domain characteristics. Because it is the intermediate code generated directly from assembly, and the assembly program is mainly based on register and memory operation, there will be some redundant code in the preliminary conversion. Therefore, for the preliminary conversion results, we will normalize and merge again to simplify the expression for the convenience of subsequent code security audit. The specific scheme is shown in Fig. 2.

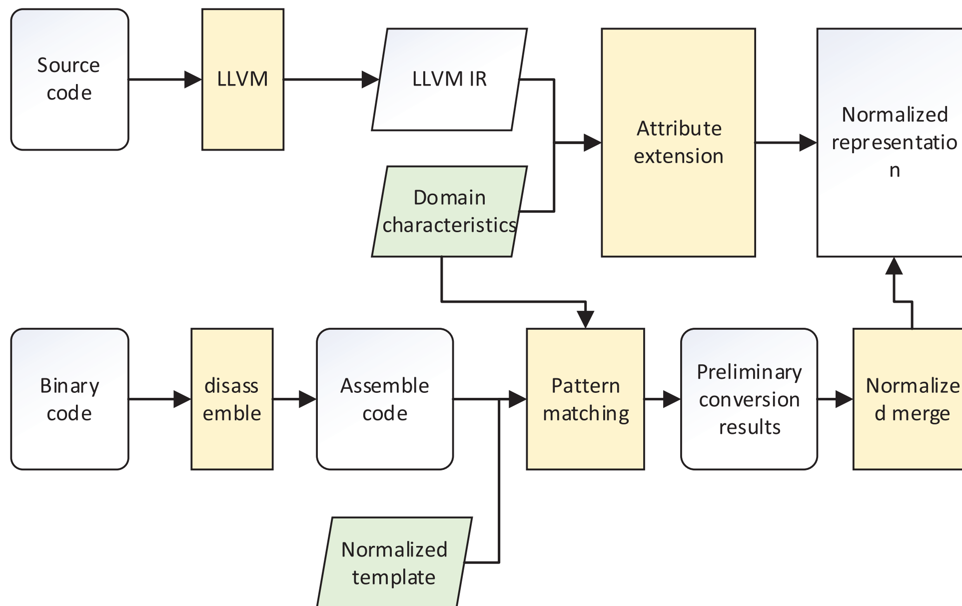


Figure 2: Normalization processing method of software based on semantic analysis and pattern matching

3.2 Code Security Audit Model Building Module

Code security audit model building module is the key module of code security audit. It provides the evaluation of code security risk level. Because its input is the code after the normalization of software code normalization processing module, it eliminates the influence of code syntax on code security audit model and is conducive to extracting more essential code security risk information.

Vulnerabilities in different fields may take different forms. For example, in the traditional desktop program code, functional vulnerabilities may be the main part of code security vulnerabilities, but in mobile embedded systems such as mobile app, energy consumption, performance and privacy can not be ignored. In real-time control applications, timing is another condition to be met. Therefore, according to the characteristics of different fields, we need to extract different code security vulnerability features and set different feature parameters in order to build a more accurate defect prediction

model. Specifically, CLCSA framework mainly extracts the corresponding vulnerability characteristics based on machine learning, and then constructs the corresponding code security audit model, as shown in Fig. 3.

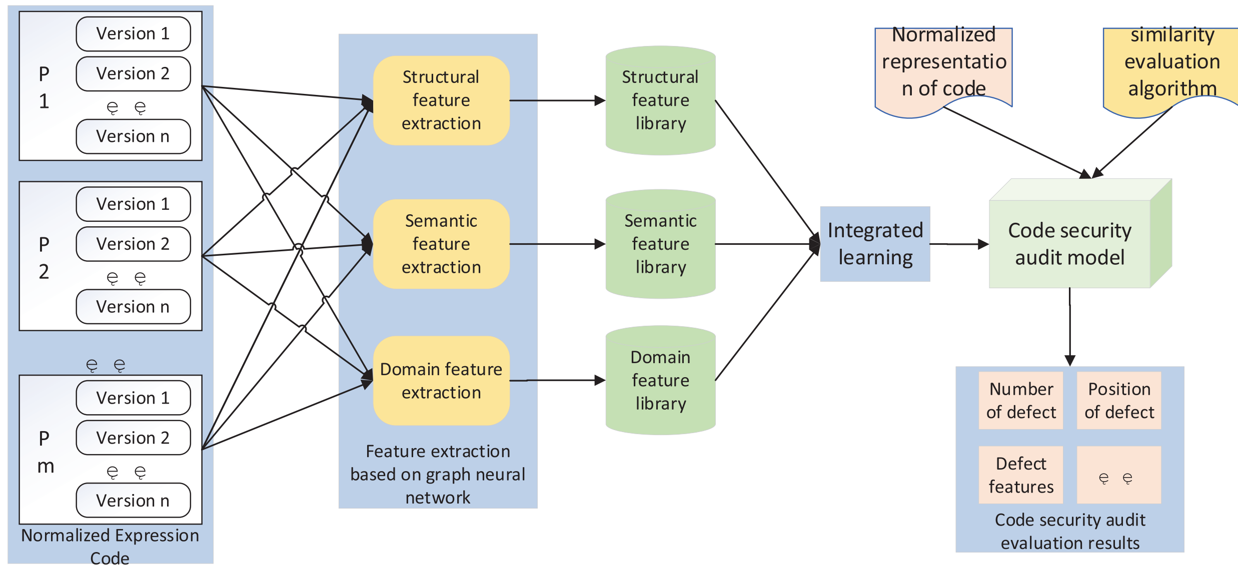


Figure 3: Code security audit model based on graph neural network and ensemble learning

Different from the direct analysis of high-level languages, the normalized representation has avoided the interference at the syntax level of various programming languages. When extracting vulnerability features, a graph model can be constructed based on the normalized expression according to the information such as control flow and data flow, and then the features can be extracted automatically with the help of the learning ability of graph neural network. However, the characteristics of different defects are different: some vulnerabilities may be expressed in the structural dimension (such as the lack of processing of a conditional branch); Some vulnerabilities may be manifested in the semantic dimension (such as incorrectly modifying iteration variables within the loop); There may also be domain related vulnerabilities (such as energy consumption defects in mobile app, etc.). Therefore, when building the code security audit model, CLCSA will extract the vulnerability features from multiple dimensions such as structural features, semantic features and domain features, and then use the integrated learning technology to comprehensively analyze the features extracted from a large number of code samples, determine the corresponding influencing factors of each feature, and form a code security audit model with domain adaptive ability.

3.3 Code Security Audit Countermeasure Suggestion Model Construction Module

The code security audit countermeasure suggestion model construction module is mainly used to build the audit conclusion countermeasure suggestion model. When it is found that there are security risks in the code, it gives the code security risk location and avoidance methods as accurately as possible, so as to provide support for repairing code security vulnerabilities in time. The module mainly includes two aspects: the construction method of vulnerability repair scheme library and the automatic generation method of repair code.

For the construction of vulnerability repair solution library, CLCSA mainly obtains from the historical update information of the software. A large amount of software code update information

has been stored in the software normalized code base. How to quickly find the repair history of corresponding vulnerabilities is the key problem to be solved in the construction of vulnerability repair scheme base. To this end, CLCSA will be based on the hash index technology, take the normalized representation of each code vulnerability as the index, and place each vulnerability repair scheme in its Hash list. Each time we select a vulnerability repair scheme, we can quickly determine its possible repair scheme according to the normalized representation of the vulnerability. In the process of selecting the vulnerability repair scheme, CLCSA will use heuristic search to find the most matching repair scheme according to the domain of the vulnerability, development language, frequency of use and other information.

For the automatic generation of repair code, CLCSA uses a code reverse generation algorithm. The algorithm takes the normalized expression as the initial language, designs the corresponding conversion template for each conversion language (including codes expressed in various high-level languages and binary programs), and then combines the vulnerability repair scheme library to complete the generation of multi-language repair template library by using the principle of pattern matching. Finally, according to the determined repair scheme, the repair code is integrated into the program to realize the automatic repair of code vulnerabilities. The basic flow of this method is shown in Fig. 4.

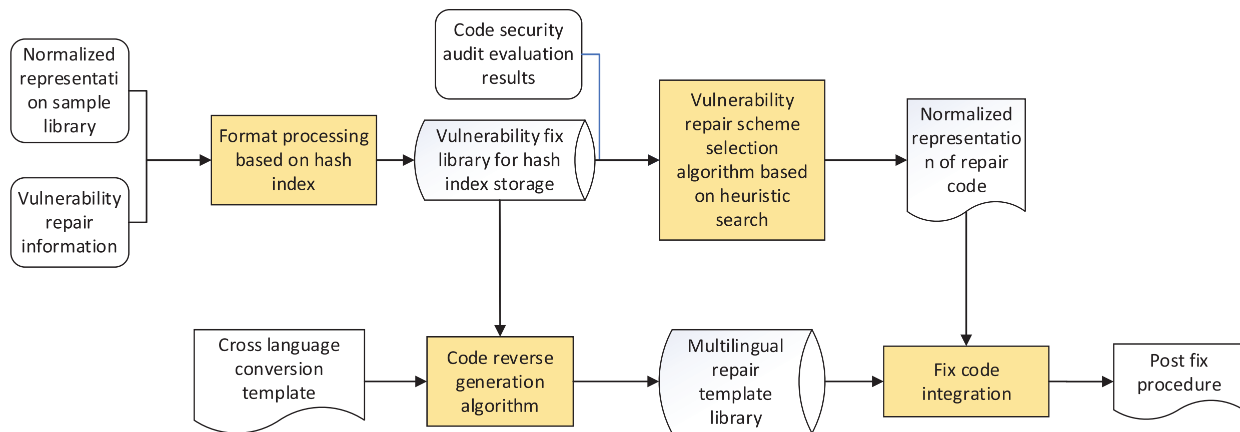


Figure 4: Code security audit countermeasure suggestion model based on heuristic search and reverse engineering

3.4 Safety Analysis

CLCSA audits the cross-language code security based on the static analysis of program by normalized representation. In specific applications, its security and effectiveness will be limited by the following two aspects.

a) Because CLCSA is mainly based on static analysis, some runtime information of the program is difficult to be accurately expressed, and there may be some false positives. However, CLCSA has combined the dynamic profile information to obtain the corresponding matching pattern and added it to the normalized software database, which can weaken the impact of this aspect.

b) The detection ability of CLCSA is largely limited by the normalized representation. If the normalized representation misses some key defect detection information, it will have a great impact on the application effect of the framework. However, the intermediate representation used in CLCSA is an extension of the intermediate representation of the compiler, so in most

cases, especially for the source code, it can maintain complete semantic information. Therefore, more complete matching pattern information can be obtained.

4 Conclusion

To meet the needs of more and more high-level programming language code security audit, improve the breadth and depth of code security audit. This paper proposes a cross language code security audit framework based on normalized representation (CLCSA). Using the software normalization module, the framework transforms high-level programming language, assembly language and binary code into the intermediate representation of the compiler, which not only makes it possible for the comprehensive audit of different high-level programming language codes, but also provides strong support for the security audit of binary code related to trade secrets; On the basis of software normalization, machine learning technology is used to adaptively build code security audit models for different fields, so as to effectively improve the pertinence of code security audit; At the same time, in order to build effective audit conclusions and give targeted audit suggestions, CLCSA framework also integrates the code security audit countermeasure suggestion model construction module to give timely and accurate code security vulnerability repair suggestions to realize the combination of audit and governance.

Funding Statement: This work was supported by the Universities Natural Science Research Project of Jiangsu Province under Grant 20KJB520026; the Natural Science Foundation of Jiangsu Province under Grant BK20180821.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] C. Xu and Y. Chen, "Research on the application of big data technology and method in audit supervision," *The Journal of Quantitative & Technical Economics*, vol. 48, no. 5, pp. 135–153, 2021.
[2] O. Desplebin, G. Lux and N. Petit, "To be or not to be: Blockchain and the future of accounting and auditing," *Accounting Perspectives*, vol. 20, no. 4, pp. 743–769, 2021.
[3] Y. Xu, J. Ren, Y. Zhang, C. Zhang, B. Shen *et al.,* "Blockchain empowered arbitrable data auditing scheme for network storage as a service," *IEEE Transactions on Services Computing*, vol. 13, no. 2, pp. 289–300, 2020.
[4] W. James, M. Robert, C. Seacord, "Secure coding standards," *The Journal of Defense Software Engineering*, vol. 20, no. 3, pp. 9–12, 2007.
[5] R. Seacord, "Secure coding in C and C++," *Pearson Schweiz Ag*, vol. 4, no. 6, pp. 74–76, 2006.
[6] L. Xiang and L. Zhi, "An overview of source code audit," in *Proc. ICIICII*, Wuhan, China, pp. 26–29, 2015.
[7] A. Javier and L. Pedro, "On the impact of smart contracts on auditing," *The International Journal of Digital Accounting Research*, vol. 21, pp. 155–181, 2021.
[8] M. K. Daskalantonakis. "A practical view of software measurement and implementation experiences within Motorola," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 998–1010, 1992.
[9] F. Akiyama, "An example of software system debugging," in *Proc. IFIP Congress*, Ljubljana, Yugoslavia, USA, pp. 353–359, 1971.
[10] M. H. Halstead, "Elements of Software Science," in *Elsevier Computer Science Library*, North-Holland, New York, NY: Operational Programming Systems Series, 1978.

[11] T. J. Mccabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1977.
[12] S. R. Chidamber and C. F. Kemerer. “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.
[13] V. R. Basili, L. C. Briand and W. L. Melo, “A validation of object-oriented design metrics as quality indicators,” *IEEE Transactions on Software Engineering*, vol. 20, no. 10, pp. 751–761, 1995.
[14] R. Subramanyam and M. S. Krishnan, “Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects,” *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297–310, 2003.
[15] Y. M. Zhou, B. W. Xu and H. Leung, “On the ability of complexity metrics to predict fault-prone classes in object-oriented systems,” *Journal of Systems & Software*, vol. 83, no. 4, pp. 660–674, 2010.
[16] Y. Y. Zhao, Y. Yang, H. M. Lu, Y. M. Zhou, Q. B. Song *et al.,* “An empirical analysis of package-modularization metrics: Implications for software fault-proneness,” *Information & Software Technology*, vol. 57, no. 1, pp. 186–203, 2015.
[17] S. Sarkar, A. C. Kak and G. M. Rama. “Metrics for measuring the quality of modularization of large-scale object-oriented software,” *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 700–720, 2008.
[18] Y. Yang, Y. Zhou, H. Lu, L. Chen, Z. Chen *et al.,* “Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? An empirical study,” *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 331–357, 2015.
[19] T. Zimmermann and N. Nagappan, “Predicting subsystem failures using dependency graph complexities,” in *Proc. ISSRE*, Trollhattan, Sweden, pp. 35–47, 2007.
[20] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *ICSE*, New York, NY, USA, pp. 284–292, 2005.