Journal of Information Hiding and Privacy Protection
DOI:10.32604/jihpp.2021.027280
Article

A Survey on Binary Code Vulnerability Mining Technology

Pengzhi Xu[1,2], Zetian Mai[1,2], Yuhao Lin[1], Zhen Guo[1,2,*] and Victor S. Sheng[3]

[1]School of Cyberspace Security (School of Cryptology), Hainan University, Haikou, China
[2]Key Laboratory of Internet Information Retrieval of Hainan Province, Haikou, China
[3]Department of Computer Science Texas Tech University, Texas, USA
[*]Corresponding Author: Zhen Guo. Email: guozhen@hainanu.edu.cn
Received: 13 January 2022; Accepted: 02 March 2022

Abstract: With the increase of software complexity, the security threats faced by the software are also increasing day by day. So people pay more and more attention to the mining of software vulnerabilities. Although source code has rich semantics and strong comprehensibility, source code vulnerability mining has been widely used and has achieved significant development. However, due to the protection of commercial interests and intellectual property rights, it is difficult to obtain source code. Therefore, the research on the vulnerability mining technology of binary code has strong practical value. Based on the investigation of related technologies, this article firstly introduces the current typical binary vulnerability analysis framework, and then briefly introduces the research background and significance of the intermediate language; with the rise of artificial intelligence, a large number of machine learning methods have been tried to solve the problem of binary vulnerability mining. This article divides the current related binary vulnerabilities mining technology into traditional mining technology and machine learning mining technology, respectively introduces its basic principles, research status and existing problems, and briefly summarizes them. Finally, based on the existing research work, this article puts forward the prospect of the future research on the technology of binary program vulnerability mining.

Keywords: Binary; vulnerability mining; stain analysis; symbolic execution; fuzzing testing; machine learning

1 Introduction

With the rapid development of the Internet, computers have been integrated into our daily lives, and computer software has brought great convenience to our lives. Taking software business as an example, the data [1] released by the Ministry of Industry and Information Technology shows that in 2020, there will be more than 40,000 enterprises above designated size in the software and information technology service industry across the country. The areas involved include daily office, life services, digital finance and many other categories. However, Exploitable vulnerabilities pose a potential threat to the safe operation of computer systems and affect user information security.

Security vulnerabilities [2] refer to defects that occur intentionally or unintentionally in the process of demand, design, implementation, configuration and operation of information technology, information products, and information systems. Once these deficiencies are used by malicious entities, they will affect the operation of normal services built on the information system, and cause serious damage to the confidentiality, integrity and availability of the information system. Therefore, the study of security vulnerabilities is one of the core contents of cyberspace security research.

Although propelled by the rapid development of programming language design and program analysis, the source code-oriented software vulnerability mining technology has made significant progress [3], but there are still some shortcomings. First, out of the protection of commercial interests and intellectual property rights, many businesses have not disclosed the source code of the program. Second, the source code will eventually be compiled into binary code. Loopholes may also occur in the process of compiling the source code. These vulnerabilities are difficult to detect at the source code level. Vulnerability mining research for binary code does not need to understand the compilation process of the source code, or even the source code of the program and its language.

With the increase in software complexity, the traditional vulnerability mining methods face high labor costs, path explosion and other issues have become increasingly prominent. Machine learning relies on its powerful ability to analyze code data to achieve automated research on security vulnerabilities. Therefore, it has gradually been widely used in binary vulnerability mining.

2 Binary Code Vulnerability Analysis Framework

In terms of vulnerability mining, although many more mature methods have been developed, these methods are often used alone and cannot fully combine the advantages of different methods. With the improvement of computer performance, it becomes possible to integrate various excellent vulnerability analysis techniques into a unified framework. Within the same framework, the complementary advantages of technology can be well formed, and the experimental data can be saved. There are existing binary vulnerability analysis frameworks.

2.1 Angr Framework

The angr binary analysis framework [4] includes the following modules: Intermediate representation module (IR), which translates binary codes into intermediate language. Binary program load module (CLE), which loads a binary program into the analysis platform. Program state representation module (SimuVEX), which represents the state of the program, these states can be specified by the user. The data model module (Claripy) provides an abstract representation for the value stored in the SimState register or memory. The flow chart of angr processing is shown in Fig. 1.

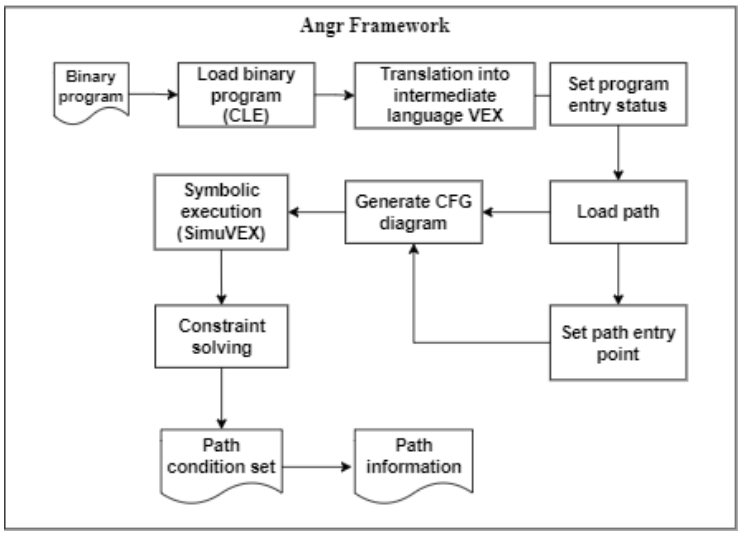

Figure 1: The processing flow of the angr framework

Although the angr framework supports cross-platform and cross-architecture, it has good compatibility, and has greatly improved analysis capabilities by introducing a powerful symbolic execution engine. However, angr still needs manual assistance for subsequent analysis because it only provides the path information of the program, and has not integrated other more typical analysis

techniques. At present, it is not possible to conduct a complete vulnerability analysis. Therefore, angr needs further research in automation and integration of other analysis technologies.

2.2MBVA Framework

MBVA is a binary program vulnerability analysis tool, which checks program specifications through a combination of abstract program modeling and model checking, so as to accurately detect and analyze the vulnerabilities in binary programs. The flow chart of MBVA framework is shown in Fig. 2.

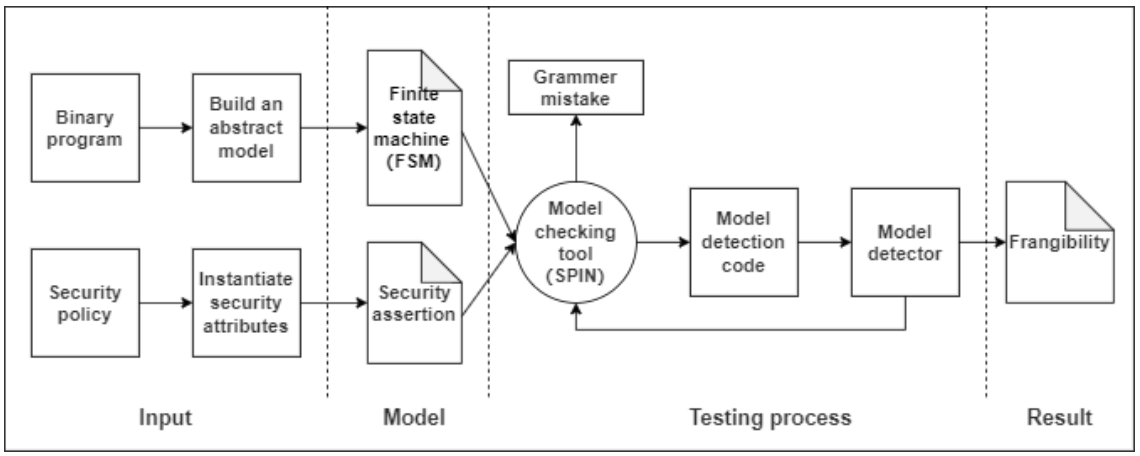

Figure 2: The processing flow of the MBVA framework

3 Background and Significance of Intermediate Language Research

Vulnerability analysis technology for binary programs has the difficulty of complex underlying instruction set and lack of corresponding semantic and type information. Researchers proposed to use intermediate language [5] to represent binary codes. This method is to convert binary codes into intermediate codes with semantic information to facilitate subsequent analysis and research. After analyzing the intermediate language with rich semantic information, we can obtain the required program CFG diagram, and then obtain the information flow of the program. Therefore, obtaining the intermediate language translated from the binary code is of great significance to the follow-up work.

There are currently some methods and tools used to convert binary to intermediate language. Jiang et al. [6] proposed an intermediate representation method called VINST, which follows the basic principle of keeping the frequently executed parts efficient and keeping other parts correct. This method is relatively simple in form but very inefficient. Zhang et al. [7] developed the dynamic binary translation technology TCG, which uses basic blocks as the translation unit to translate binary codes. The dynamic translator will cache the translation and optimization results, which can provide more space for dynamic translation.

However, the current vulnerability analysis technology based on intermediate language also faces some problems. Some information may be lost in the process of translation into an intermediate language. The intermediate language translates one instruction into several instructions, which may reduce efficiency. Regarding the efficiency and accuracy of the translation process, firstly, you can set translation rules to delete redundant and redundant segments. Secondly, try to ensure the integrity of semantic information during the deletion process.

4 Traditional Binary Vulnerability Analysis Technology

There are many existing traditional binary vulnerability analysis techniques [8]. For example, from the perspective of software operation, it can be divided into dynamic analysis, static analysis, and dynamic and static analysis; from the perspective of openness of software code, it can be divided into black box testing, white box testing and gray box testing; from the perspective of the research object form,

it can be divided into Intermediate language and vulnerability analysis technology based on the underlying instruction set. Among the typical technologies are stain analysis, symbolic execution and fuzzing testing. These three technologies almost cover these classification standards.

4.1 Stain Analysis Technology

Stain analysis was first proposed by Denning [9] in 1976. At present, taint analysis techniques are widely used in information leakage detection, vulnerability detection, and reverse engineering. The taint analysis technology marks the data in the system or application as tainted or non-stained. When the tainted data can affect the non-stained data according to the information flow dissemination strategy, the label of the non-stained data is modified as a tainted. When the tainted label is finally propagated along with the data to the designated storage area or information leakage point, it is considered that the program has a security risk. The treatment process of stain analysis technology can be divided into three stages: identification of stain source, stain propagation analysis, and harmless treatment. The basic process is shown in Fig. 3.

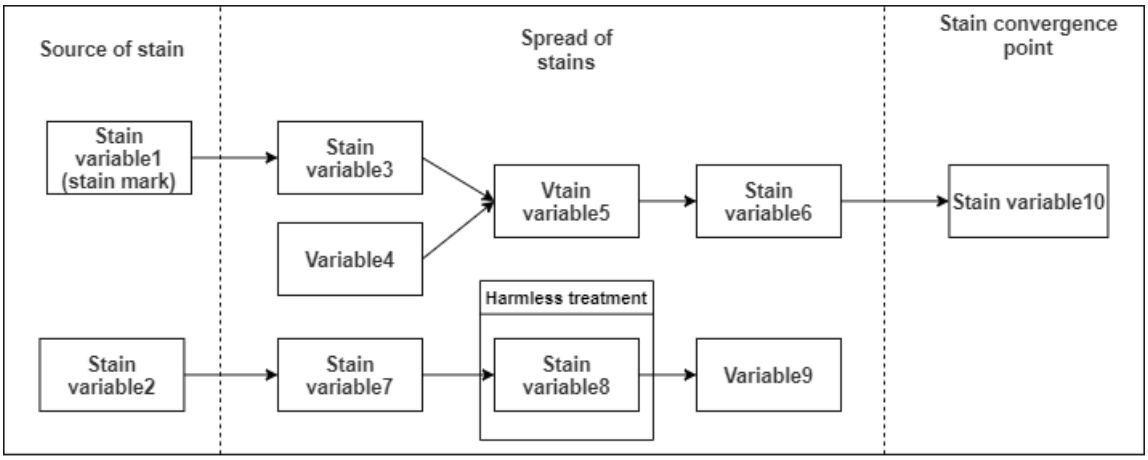

Figure 3: The process of stain analysis

Static stain analysis mainly uses methods such as lexical and grammatical analysis to analyze the data and control dependencies between variables offline. During this process, neither the target program is run nor the code needs to be modified. However, because the target program is not run and additional information cannot be obtained when the program is running, static stain analysis technology still has the problem of insufficiently accurate analysis results.

The principle of dynamic taint analysis technology is to mark data from untrusted sources, track and record its propagation process during program execution, and detect the illegal use of tainted data to achieve the purpose of detecting vulnerabilities.

Dynamic binary instrumentation analysis focuses on the real behavior of the program when it is running. It directly monitors the process during program execution and inserts the instruction stream on the basis of not destroying the original logic of the target program. Taking the Pin [10] platform as an example, the dynamic binary instrumentation framework is shown in Fig. 4.

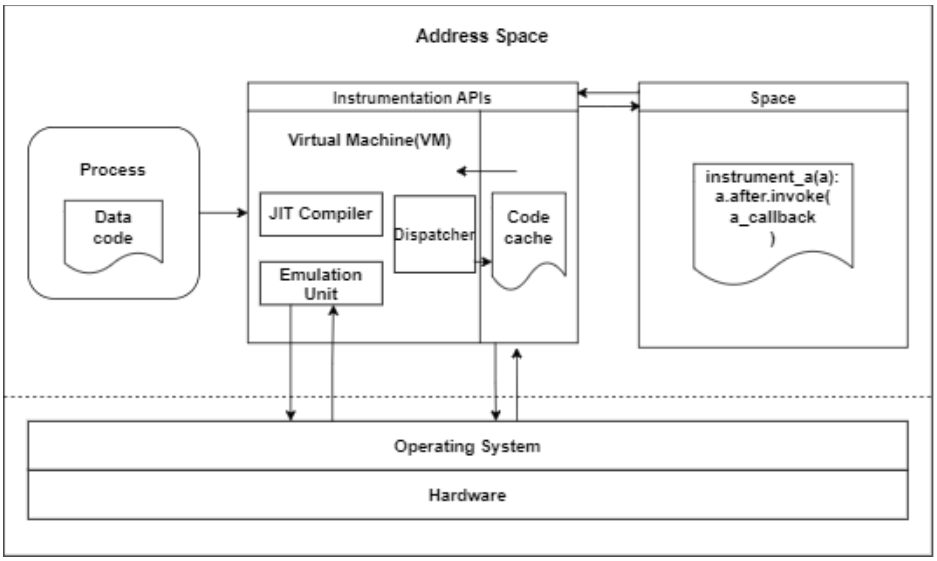

Figure 4: The dynamic binary instrumentation framework

4.1.1 Research Status

There are already many tools that use taint analysis techniques to mine binary vulnerabilities. Dong et al. [11] proposed a dynamic taint analysis system ODDTA for binary vulnerability mining. The system expands the taint status definition in the taint identification process, and refines the taint status attributes defined in the taint marking process. In the taint detection stage, the security detection rules are further expanded, and different response strategies are designed according to the different levels of the security rules. Yin et al. [12] proposed an extended platform TEMU based on the QEMU virtualizer. TEMU uses a system-wide perspective to analyze the interaction between activities in the kernel and multiple processes, and conduct in-depth analysis in a fine-grained manner.

In order to overcome the shortcomings of dynamic stain analysis and static stain analysis, some researchers have combined the two. This analysis method can not only compensate for the loss of information caused by insufficient path coverage in dynamic analysis, but also provide some accurate operating information that cannot be obtained by static analysis.

4.1.2 Existing Problems

(1) Implicit flow analysis problem

The core problem faced by static implicit flow analysis is that accuracy and efficiency cannot be achieved at the same time. Path-sensitive data flow analysis tends to cause path explosion problems, resulting in unacceptable overhead. In order to reduce the cost, a simple method of statically propagating the taint of branch statements is to mark all the statements whose control depends on it. However, this method will cause some variables that do not carry private data to be marked, leading to the occurrence of excessive pollution.

The primary concern of dynamic implicit flow analysis is how to determine the range of sentences that need to be marked under taint control conditions. Current research mostly uses offline static analysis to assist judgment. Clause et al. [13] proposed to use the post-dominate relationship between control flow graph nodes obtained by offline static analysis to solve the implicit flow labeling problem in dynamic taint propagation. The second problem is the underreporting of potential security issues due to the leakage of some tainted information. Vogt et al. [14] added offline static analysis on the basis of traditional dynamic taint analysis to track the control dependency in the dynamic execution process and mark the variables in all assignment statements within the control range of the taint branch. But there will still be pollution. The third problem is how to choose the appropriate taint marking branch for taint propagation.

The selection strategy of the taint marking branch can be designed quantitatively according to the different scope of information leakage.

(2) The problem of lack of semantic information

Because the binary code lacks necessary semantic and type information, the application of dynamic stain analysis technology in binary program analysis is limited. Zhuge et al. [15] proposed dynamic stain analysis technology based on type perception and symbolic execution technology oriented to type variables. This technology promotes dynamic stain analysis in binary program analysis to variable granularity.

(3) Analyze overhead problem

The operating efficiency of dynamic stain analysis directly affects user experience. Therefore, it is necessary to reduce the cost of analysis as much as possible. One of the current solutions is to selectively control the number of instructions that need to be tainted, and combine other means to achieve better optimization results.

4.2 Symbolic Execution Technology

The idea of symbolic execution was first proposed by King et al. [16] in 1975. Its basic principle is to use abstract symbols to replace program variables. The output of program calculations is expressed as a function of input symbol values, and the execution space of the program is traversed according to the semantics of the program, so as to carry out relevant analysis. Symbolic execution, as an important formal method and software analysis technique, plays an important role in software testing and program verification, and can be applied to program vulnerability detection [17]. When performing vulnerability analysis on a binary program through symbolic execution, it is first necessary to obtain the intermediate representation of the program. Then perform symbolic execution and constraint solving to obtain the vulnerability analysis result. The analysis process is shown in Fig. 5.

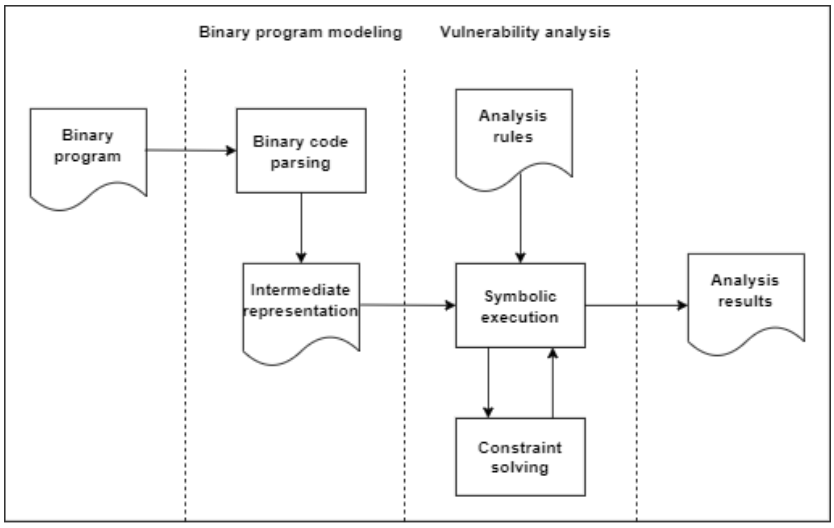

Figure 5: The process of symbolic execution

Symbolic execution can be divided into classic symbolic execution and dynamic symbolic execution. The classical symbolic execution is based on the analytical procedure, and the execution is simulated by the symbolic value. Due to the lack of program dynamic runtime information in the classic symbols, program execution is difficult to be fully simulated, and the analysis is not accurate enough. To solve this problem, related researchers have proposed dynamic symbolic execution technology. Dynamic symbolic execution combines the advantages of specific execution and classic symbolic execution, which improves the accuracy of analysis.

4.2.1 Research Status

Symbolic execution was proposed in the 1970s [18]. After a period of development, Bush et al. [19] proposed a vulnerability mining tool Prefix based on static symbolic execution, which uses path sensitivity and process analysis to improve the accuracy rate. So symbolic execution technology can be applied in the field of vulnerability mining.

The static symbolic execution technology is difficult to be completely simulated due to its lack of relevant information when the program is dynamically running. The analysis is not accurate enough. To solve this problem, researchers proposed to use dynamic symbolic execution for vulnerability analysis. Since the dynamic symbolic execution technology [20] was proposed, it has been widely used in the field of vulnerability mining and has produced many related projects and tools, such as KLEE [21], Mayhem [22], SAGE [23–24], S2E [25], etc. Among them, KLEE was developed by Cadar [21]. It is an open source tool that uses symbolic execution technology to construct program test cases. While analyzing the program to construct test cases, it also uses symbolic execution and constraint solving techniques at key program points. Analyze the value range of the symbol. If it is found that the value of the symbol cannot meet the safety regulations during the analysis, it is considered that there is a corresponding loophole in the program.

Researchers have also developed many tools for vulnerability analysis at the binary code level, such as BitBlaze [26], SmartFuzz [27], etc. BitBlaze [26] is a binary analysis platform, which integrates mainstream binary analysis techniques, among which Vine, TEMU and Rudder correspond to static analysis, dynamic analysis, and dynamic symbolic execution functions. SmartFuzz [27] is a tool based on the Valgrind binary instrumentation platform, which can be used to find integer vulnerabilities in binary programs under Linux systems.

4.2.2 Existing Problems

(1) Path explosion problem

The path explosion problem is the main factor restricting the application of symbolic execution in real-world program analysis. Because in the analysis process of symbolic execution, at each branch node, symbolic execution will derive two symbolic execution instances, and the number of program branch paths increases exponentially with the number of program branches. The main ideas to alleviate the path explosion problem are as follows:

1) Use heuristic search method to search the program path space. Sen et al. [28] used a hybrid random symbol search strategy in CUTE and jCUTE to improve the breadth and depth of the test.

2) State consolidation. In 2014, Avgerinos et al. [29] proposed the concept of Veritesting, which reduces the state space of the program through state fitting and improves the usability of dynamic symbolic execution.

3) Redundant path pruning. In the process of program analysis, some paths are redundant. The redundant paths are determined through analysis and pruned. The challenge is that the judgment of redundant paths is more complicated, it is difficult to judge them comprehensively. The misjudgment of redundant paths may lead to the final analysis of the target.

(2) Constraint solving problem

The efficiency of symbolic execution in program analysis largely depends on the efficiency of constraint solving. The core problem of constraint solving is to convert the arithmetic constraints of path conditions into basic solver problems.

1) Non-linear integer constraints often make path conditions unsolvable, and the solvability of constraint sets with nonlinear constraints is generally undecidable [30].

2) The solver cannot handle the external library function calls included in the path constraint conditions [31].

Researchers have also proposed some optimization methods for constraint solving: irrelevant

constraint elimination technology and cache solution strategy [20].

1) Elimination of irrelevant constraints. The purpose of irrelevant constraint elimination is to reduce the number of constraint items through analysis. The reason is that a program branch usually only depends on a small part of the program variables, and the program variables that the branch depends on may be independent of the variables contained in other constraints on the path.

2) Cache solution strategy. The constraint information between adjacent executions is only slightly different, so theoretically, there are only minor differences in the solution results. Ramos et al. [32] proposed a lazy constraint solution strategy in 2015. The core idea is that only when the branch judgment reaches the target location, the query solver verifies the reachability of the path and generates test cases.

4.3 Fuzzing Technology

Fuzzing technology was first proposed by Miller et al. [33] in 1989. Fuzzing is an automatic software testing technology based on defect injection. It uses a large amount of semi-valid data as the input of the application, and uses the abnormality of the program as a sign to discover possible security vulnerabilities in the application.

The workflow of fuzzing [8] roughly goes through three basic stages. The preprocessing stage collects target-related information and develops a fuzzing test strategy to make necessary preparations for monitoring the running status of the target in the test. After entering a data link to obtain a large amount of data in the testing phase, filter the data through the input selection link to filter out invalid input data. The judgment part is mainly to design appropriate experiments and evaluate and judge the fuzz test according to the end conditions. The workflow of the fuzzing test is shown in Fig. 6.

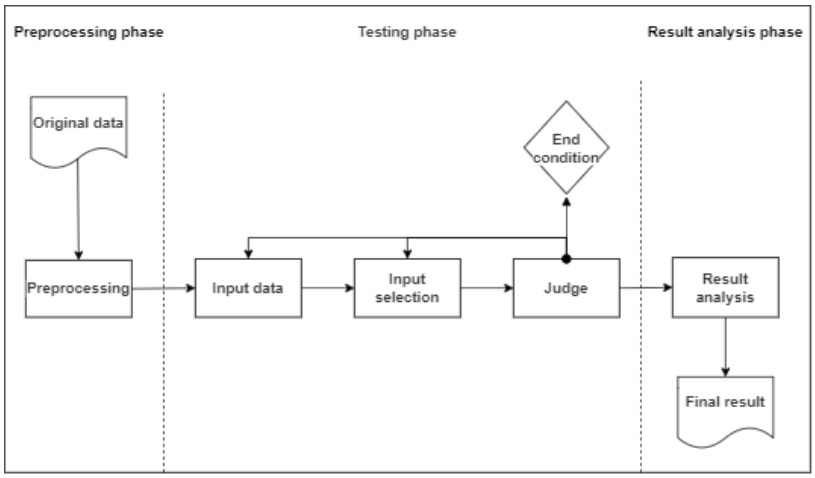

Figure 6: The workflow of the fuzzing test

According to the magnitude of the analysis of the internal structure of the program, the fuzz testing technology can be divided into white box, black box, and gray box fuzz testing. Divided by the way of sample generation, the test input of fuzz testing can be divided into two ways: mutation-based and generation-based [8]. The mutation-based fuzzing method uses the technology of mutating existing data to create new test cases. The generation-based fuzzing method is to generate test cases from scratch by modeling the specific program to be tested.

4.3.1 Research Status

Fuzzing can be traced back to 1989, and its main purpose was to try to use unconventional data to detect the robustness of the target. In 1999, the protocol testing project team of the University of Oulu in Finland developed the network protocol security testing software PROTOS [34], which used the gray-box testing method and applied the fuzzing test to the test of the network protocol for the first time.This is the

beginning of fuzzing technology as a practical tool.

In 2004, IOACTIVE released the famous cross-platform fuzzing framework Peach [35], which can fuzz test almost all common objects, such as file structure, network protocol, API interface, etc. Peach can manually define the data model used to generate input data by the user. It is an early application of the idea of generating input based on grammar. In the same year, Dave released the open source fuzzer SPIKE [36], which implements a block-based method and has the ability to describe variable-length data blocks. In 2008, Godefroid [37–38] of Microsoft Research introduced the idea of white box testing in the fuzzing method, and at the same time applied symbolic execution and constraint solving methods to the fuzzing test, which improved the coverage of the fuzzing test.

In response to the limitations of fuzz testing when encountering checksum protection mechanisms, in 2010, Wang of Beijing University [39] combined symbolic execution and fine-grained dynamic stain propagation technology. He proposed a method to bypass verification and developed the corresponding tool TaintScope. Remove the obstacles for the application of fuzz testing to find deep vulnerabilities. American fuzzy lop (AFL) [40] fuzzing tool that appeared in 2013 is a gray box fuzzing test that uses a small amount of internal information to carry out fuzzing ideas. AFL is a coverage-oriented fuzzing testing tool. Through the method of instrumentation, the edge coverage corresponding to the input data is collected as a measure of the selection of the fuzzing test seed.

4.3.2 Existing Problems

(1) Optimization problem of test case generation strategy

Traditional fuzzing tests often blindly mutate a certain part of normal test cases when generating test cases, which causes the scale of test cases to explode and the test effect is bad. The current improvement methods are:

1) The introduction of annealing genetic algorithm [41–42] to improve the test rules, so that the smallest set of test cases can cover the largest code execution path, in order to discover those hidden software vulnerabilities.

2) You can also consider combining fuzz testing with other binary vulnerability analysis techniques to learn from each other.

(2) The degree of automation needs to be improved

Fuzzing is a blind injection method, which often requires manual participation in determining the constraints of input data and generating test cases, which affects its practical value. Therefore, the focus of current research is mostly on automated and intelligent fuzzing testing methods. On the one hand, genetic algorithms can still be considered to improve the degree of automation of testing. On the other hand, with the rapid development of machine learning, if you find a suitable combination of the two, you can maximize the advantages of the two.

5 Machine Learning Technology

Machine learning [43] refers to the study of computer simulations or the realization of human learning behaviors in order to acquire new knowledge or skills, and reorganize the existing knowledge structure to continuously improve the performance of the computer system itself. In recent years, with the rapid development of machine learning, researchers have begun to use machine learning techniques to alleviate some bottlenecks in the field of software vulnerability mining. Through the use of machine learning technology, help corresponding vulnerability mining tools to train massive vulnerability data and generate models to classify and predict samples. Finally, improve the accuracy and efficiency of software vulnerability mining.

In machine learning, first, the data acquisition module collects a large amount of software program-related data used for training and evaluation, and applies them to the training phase and the detection phase respectively. Experiment [44] shows that: in the software defect prediction method based

on machine learning, data preprocessing is more important than the choice of classifier. Therefore, in the training phase, the training data set needs to be balanced and preprocessed through a series of sampling methods. In the model training module, the vector obtained from the data characterization module is used as input, and the feature expression extracted by the model is used as output. After several trainings, a classifier model for the detection stage is obtained. In the detection phase, the target code data set is first characterized by the same method. Then, the vector obtained by the characterization is sent to the classifier model obtained in the training phase to obtain the classification or prediction result of the target code. Finally, evaluate and tune the model. The basic process of machine learning is shown in Fig. 7.

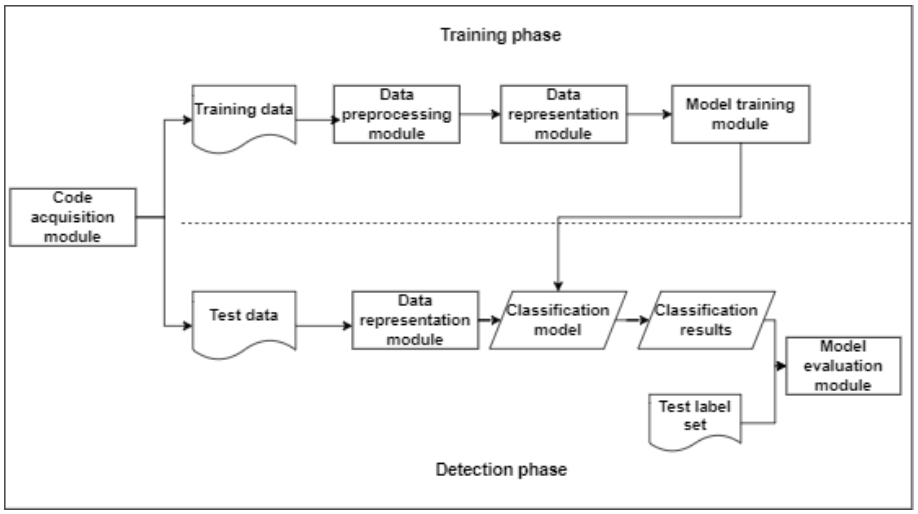

Figure 7: The basic process of machine learning

5.1 Research Status

In recent years, researchers have conducted in-depth research on the principles and conditions of loopholes, and used machine learning algorithms to mine loopholes, and achieved good results. The work of using machine learning to mine software vulnerabilities can be divided into the following categories [45].

(1) Vulnerability prediction based on software measurement

Software measurement [46] is a process of continuous quantitative analysis of products in the software development process. Software metrics are quantitative representations of specific software attributes, including metrics such as complexity, coupling, and cohesion. The software measurement method only needs to use the software measurement as the feature set, and then use the supervised learning method in machine learning to train the model to predict the vulnerabilities of the software component.

(2) Vulnerability prediction based on anomaly detection

Anomaly detection refers to the process of discovering data that does not conform to normal behavior patterns. In 2010, Gruska et al. [47] proposed a cross-project anomaly detection model, which extracts all possible function call sequences and related information in a specified function, and finally establishes a finite automata model of the function. In 2013, Yamaguchi et al. [48] proposed a system that combines machine learning and static code analysis to check for missing checks in the source code. By using the nearest neighbor technology to find the neighbors of the specified function, then map the function and its neighbor functions to the vector space, calculate the vector center after the mapping, and mark the function far from the center as anomalies. Its advantage is that it can automatically find software vulnerabilities caused by improper use of API and logic vulnerabilities caused by neglected conditions and missing inspections.

(3) Vulnerability prediction based on vulnerability pattern recognition

Use machine learning methods to extract vulnerable code patterns from vulnerable code samples, and then use pattern matching technology to detect and locate vulnerabilities in software. In 2012, Yamaguchi et al. [49] used information retrieval technology to propose a vulnerability extraction method to assist software security audits. Two years later, Yamaguchi et al. [50] merged the concepts of abstract syntax tree, control flow graph and program dependency graph in classic program analysis into a code attribute graph, and then carried out vulnerability modeling in the form of graph traversal.

In 2014, Pewny [51] first proposed a preliminary study of using text analysis techniques to predict vulnerable software components. In 2017, Young et al. [52] proposed a deep learning-based assembly code representation method. Based on Word2Vec, they proposed an Instruction2vec method suitable for assembly instruction vector representation, and then adopted a convolutional neural network model and loop the neural network model is used to predict software defects, and the results show that the effect is very good.

Now, most of the research is on source code, and only a few are on binary software. Binary program function recognition is the basis of binary analysis. However, because the binary code lacks information in high-level language programs, it is often difficult to recognize functions. Bao proposed the ByteWeight scheme, which uses machine learning algorithms to realize the recognition of binary program functions. First, a weighted prefix tree is used to learn the signature of the function, and the function is identified by the way the signature matches the binary fragment. Then, the value set analysis and incremental control flow recovery algorithm are used to realize the identification of the function boundary.

5.2 Existing Problems

(1) Data collection

To apply machine learning to vulnerability mining, we must first build a unified and standardized vulnerability data set. Only a normative data set can scientifically evaluate existing research in an all-round way. However, there is currently no publicly available vulnerability data set that can be used as a benchmark. For binary programs, since they do not contain syntactic and semantic information, they often need to be converted into intermediate language before using machine learning models for training. Therefore, to introduce machine learning into the vulnerability mining of binary programs, an open and capable intermediate language data set containing relevant semantic information needs to be established.

(2) Feature selection

For machine learning models, how to select features that can fully represent the data set is the key to constructing a model with superior performance. This requires comprehensive consideration of the model itself, vulnerability information, and program operating environment. In a binary program, static features can be obtained from call graphs, control flow graphs, data flow graphs, etc., but there is a problem of a large amount of calculation for the established graph-based structure. Dynamic features can be executed by instrumentation to track actual function calls and the method of calling parameters is captured, but the overhead of instrumentation is also very large. It may even affect the space layout of the memory when the program is actually executed.

(4) Model selection

The application of machine learning to vulnerability research, especially the vulnerability mining of binary programs, is still in its infancy. How to use the powerful performance of the machine learning model to conduct vulnerability mining and vulnerability assessment is a difficult problem. Some machine learning models are often only suitable for specific scenarios, and even in the same scenario, the effects of different models will be quite different.

6 Expectation

In future research, mature technologies can be combined to build a comprehensive binary vulnerability mining platform, so as to analyze the vulnerabilities in binary programs. (1) When directly exploiting vulnerabilities in the binary code, build a corresponding binary information database for

subsequent analysis; (2) When using an intermediate language for vulnerabilities mining, ensure that the binary information is complete. (3) Before analyzing the program using taint analysis, symbolic execution and fuzzing testing, machine learning can be used to pre-train the data and generate models to guide the generation of high-quality test input samples. (4) Using taint analysis technology to track the bytes that affect the conditional branch in the test input to reduce the number of paths; At the same time, it combines symbolic execution technology and fuzz testing to conduct vulnerability mining. The process of binary vulnerability analysis framework is shown in Fig. 8.

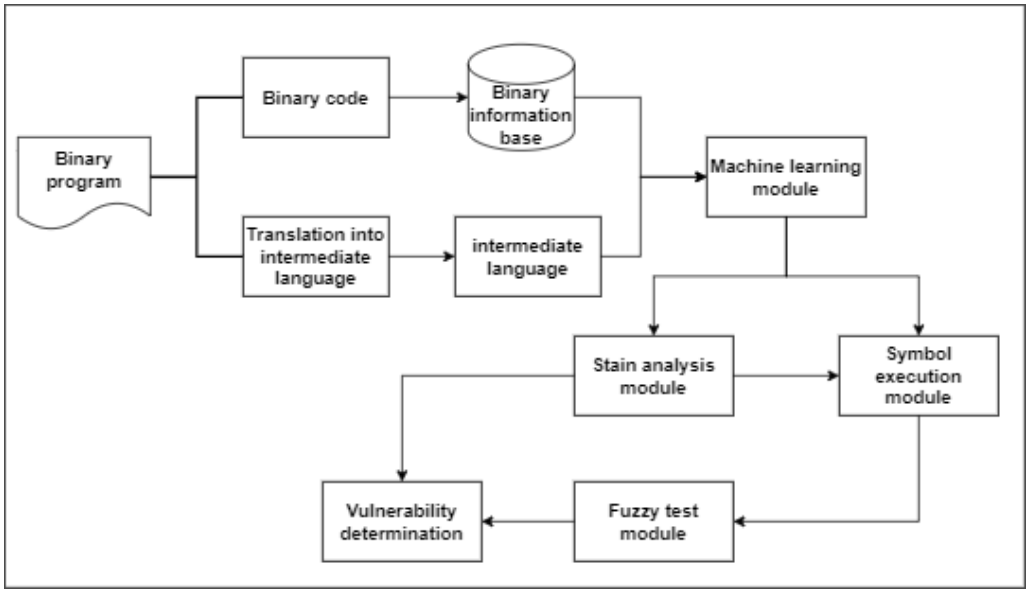

Figure 8: Binary vulnerability analysis framework process

7 Conclusion

Vulnerability mining is an emerging topic. Due to its complexity, it usually requires a combination of multiple fields and multiple technologies for research. In the future, it is a relatively important research direction to combine various vulnerability mining technologies and learn from each other to make up for the shortcomings faced by a single vulnerability analysis technology.

This article summarizes the current research status of binary vulnerability mining by investigating the current typical binary vulnerability analysis framework, as well as introducing the principle, current situation and existing problems of traditional binary code vulnerability mining technology and machine learning vulnerability mining technology. Finally, a binary vulnerability analysis framework combining traditional techniques and machine learning is proposed.

Acknowledgement: The authors would like to thank the partners for their hard work, as well as the reviewers for their detailed review and valuable comments.

Funding Statement: This paper is based on the funding of the following two projects: Research on Key Technologies of User Location Privacy Protection and Data Integrity Verification under Mobile P2P Architecture, Project No. (619QN193); Research on Security Vulnerability Detection Technology of Open Source Software Based on Deep Learning, Project No. (ZDYF2020212).

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

References

[1] Ministry of Industry and Information Technology. [Online]. Available: https://www.miit.gov.cn//, 2021.

[2] Q. X. Liu, Y. Q. Zhang, Y. F. Gong and H. Wang, "Vulnerability identification and description specification," National Information Security Standardization Technical Committee. GB/T28458-2012.

[3] T. L. Wang, "Research on key techniques of vulnerability mining for binary programs," Ph.D. dissertation, Beijing University, Beijing, 2011.

[4] Y. Shoshitaishvili, R. Y. Wang, C. Salls, N. stephens and G. Vigna, "The art of war: Offensive techniques in binary analysis", in *2016 IEEE Sym. on Security and Privacy*, San Jose, USA, pp. 138–157, 2016.

[5] N. Li and J. M. Pang, "Binary translation intermediate code optimization method based on intermediate representation rule replacement", *Journal of National University of Defense Technology*, vol. 43, no. 4, pp. 156–162, 2021.

[6] L. Y. Jiang, A. L. Liang and H. B. Guan, "Intermediate representation in dynamic binary translation," *Computer Engineering*, vol. 35, no. 9, pp. 283–285, 2009.

[7] X. C. Zhang, X. Y. Guo and L. Zhao, "TCG: Research on dynamic binary translation technology," *Computer Applications and Software*, vol. 30, no. 11, pp. 35–41, 2013.

[8] S. Z. Wu, G. Tao, G. W. Dong and P. H. Zhang, *Software Vulnerability Analysis Technology*, Beijing, China: Science Press, 2014.

[9] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[10] C. K. Luk, R. Cohn, R. Muth, H. Patil and K. M. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[11] G. L. Dong, L. Zang, H. Li, L. Gan and Y. K. Guo, "Binary program vulnerability detection based on taint analysis," *Computer Technology and Development*, vol. 28, no. 3, pp. 138–142, 2018.

[12] H. Yin and D. Song, "Temu: Binary code analysis via whole-system layered annotative execution," *Electrical Engineering and Computer Sciences*, 2010.

[13] J. Clause, W. Li and A. Orso. "DYTAN: A generic dynamic taint analysis framework," in *Proc. of the 2007 Int. Sym. on Software Testing and Analysis*, ACM Press, USA, pp. 196−206, 2007.

[14] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *Proc. of the NDSS*, San Diego, USA, 2007.

[15] L. W. Zhuge, L. B. Chen and F. Tian, "Dynamic stain analysis based on type," *Journal of Tsinghua University (Natural Science Edition)*, vol. 52, no. 10, pp. 1320–1321, 2012.

[16] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[17] R. S. Boyer, B. Elspas and K. N. Levitt, "Select-a formal system for testing and debugging programs by symbolic execution", in *Int. Conf. on Reliable Software*, pp. 234–245, 1975.

[18] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler, "EXE: Automatically generating inputs of death," in *Proc. of the 13th ACM Conf. on Computer and Communications Security*, pp. 322–335, 2006.

[19] W. R. Bush, J. D. Pincus and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software-Practice and Experience*, vol. 30, no. 7, pp. 775–802, 2000.

[20] P. Godefroid, N. Klarlund and K. Sen, "DART: Directed automated random testing," in *Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Chicago, USA, pp. 213–223, 2005.

[21] C. Cadar, D. Dunbar and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Sym. on Operating Systems Design and Implementation*, San Diego, California, pp. 209–224, 2008.

[22] S. K. Cha, T. Avgerinos, A. Revert and D. Brumley, "Unleashing mayhem on binary code," in *Proc. of 2012 IEEE Sym. on Security and Privacy*, pp. 380–394, 2012.

[23] P. Ggoefroid, M. Y. Levin and D. A. Molnar, "Automated white boxfuzz testing," in *Proc. of NDSS*, San Diego, USA, pp. 151–166, 2008.

[24] E. Bounimova, P. Godefroid and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proc. of the 2013 Int. Conf. on Software Engineering*, pp. 122–131, 2013.

[25] V. Chipounov, V. Kuznetsov and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[26] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager *et al.*, "BitBlaze: A new approach to computer security via binary analysis," in *4th Int. Conf. on Information Systems Security*, Hyderabad, India, Springer, pp. 1–25, 2008.

[27] D. Molnar, X. C. Li and D. Wagner, "Dynamic test generation to find integer bugs in x86 binary Linux programs," in *18th Conf. on USENIX Security Sym.*, Montreal, Canada, pp. 67–82, 2009.

[28] K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path Model-Checking tools," in *Int. Conf. on Computer Aided Verification*, pp. 419–423, 2006.

[29] T. Avgerinos, A. Rebert, K. C. Sang and D. Brumley, "Enhancing symbolic execution with veritesting," in *Int. Conf. on Software Engineering*, pp. 1083–1094, 2014.

[30] H. B. Enderton, M. Davis, "Computability, unsolvability. Hilbert's tenth problem is unsolvable," *American Mathematical Monthly*, pp. 233–269, 1973.

[31] X. Xiao, T. Xie, N. Tillmann and J. de Halleux, "Precise identification of problems for structural test generation," in *Proc. of the 33rd Int. Conf. on Software Engineering*, pp. 611–620, 2011.

[32] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *USENIX Conf. on Security Symposium*, pp. 49–64, 2015.

[33] B. P. Miller, L. Fredriksen and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[34] G. Shu and D. Lee, "Testing security properties of protocol implementations-a machine learning based approach," in *27th Int. Conf. on Distributed Computing Systems*, 2007.

[35] P. Oehlert, "Violating Assumptions with fuzzing," *IEEE Security and Privacy Magazine*, vol. 3, no. 2, pp. 58–62, 2005.

[36] D. Aite, *The Advantages of Block-Based Protocol Analysis of Security Testing*, Immunity Inc., New York, 2002.

[37] P. Godefroid, A. Kiezun and M. Y. Levin, "Grammar-based whitebox fuzzing," in *2008 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Tucson, USA, pp. 206–215, 2008.

[38] P. Godefroid, M. Y. Levin and D. Molnar, "Automated whitebox fuzzing," in *Proc. Network Distributed Security Sym. (NDSS)*, San Diego, USA, 2008.

[39] T. L. Wang, T. Wei, G. F. Gu and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *IEEE Sym. on Security and Privacy*, Oakland, USA, pp. 497–512, 2010.

[40] M. Zalewski, "American fuzzy lop," [Online]. Available: https://lcamtuf.coredump.cx/afl/.

[41] V. Ganesh, T. Leek and M. Rinard, "Taint-based directed whitebox fuzzing," in *Int. Conf. on Software Engineering*, pp. 474–484, 2009.

[42] A. Lanzi, L. Martignoni, M. Monga and R. Paleari, "A smart fuzzer for x86 executables," in *Third Int. Workshop on Software Engineering for Secure Systems*, 2007.

[43] I. H. Witten, E. Frank and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann, 2016.

[44] A. Agrawal and T. Menzies, "Is better data better than better data miners? On the benefits of tuning smote for defect prediction," in *Proc. of the 40th Int. Conf. on Software Engineering*, pp. 1050–1061, 2018.

[45] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Computing Surveys*, vol. 50, no. 4, pp. 1–36, 2017.

[46] D. H. Xing, J. D. Cao and H. C. Wang, "Overview of software metrology," *Computer Engineering and Applications*, vol. 27, no. 1, pp. 17–19, 2001.

[47] N. Gruska, A. Wasylkowski and A. Zeller, "Learning from 6,000 projects: Lightweight cross-project anomaly detection," in *Int. Sym. on Software Testing and Analysis*, pp. 119–130, 2010.

[48] F. Yamaguchi, C. Wressnegger, H. Gascon and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Conf. on Computer and Communications Security*, pp. 499–510, 2013.

[49] F. Yamaguchi, M. Lottmann and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Computer Security Applications Conf.*, pp. 359–368, 2012.

[50] F. Yamaguchi, N. Golde, D. Arp and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," *Security and Privacy*, pp. 590–604, 2014.

[51] J. Pewny, F. Schuster, L. Bernhard, T. Holz and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Computer Security Applications Conf.*, pp. 406–415, 2014.

[52] J. L. Young, C. Sang-Hoon, C. Kim, S. H. Lim and K. W. Park, "Learning binary code with deep learning to detect software weakness," in *Int. Conf. on Internet*, pp. 245–249, 2017.