Software Supply Chain Analysis Based on Function Similarity

Wenjie Sun, Zheng Shan*, Fudong Liu*, Xingwei Li, Meng Qiao, Chunyan Zhang
State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China
*Corresponding author’s e-mail: zzzhengming@163.com, lwfydy@126.com

Abstract. The supply of open source and open source components is growing at an alarming rate, while vulnerabilities in open source components are everywhere. Software supply chain analysis aims to discover third-party components and open source code used in a software, and analyze the software's dependence on components. In this paper, we propose a software component analysis method and a known vulnerabilities detecting method. By scanning the open source components of the binary file and conducting vulnerability analysis, the known vulnerabilities are detected. This paper mainly solves the problem of detecting known vulnerabilities in the supply chain of binary files. We conducted a case analysis and achieved good results.

1. Introduction
With the development of the open source community, more and more open source codes appear on the Internet, which can provide commonly used basic functions. In the process of software development, developers often search the Internet and open source code repositories to find existing open source code, and import existing functional codes into their own projects to improve development efficiency. There is even a lot of software that is assembled with third-party components and open source code. Third-party components and open source code provide the necessary software building foundation, allowing developers to quickly deliver software products and shorten time to market. This development method brings many benefits, but due to the lack of security audits of open source code and third-party components, the security performance of software products is affected by the quality of each component. The large-scale impact of vulnerabilities such as Heartbleed, Shellshock, Poodle, and Ghost highlight the impact of vulnerabilities in common open source component versions [1].

As today's software is more and more assembled from scattered open source code and third-party components, the vulnerabilities in these components have become a huge blind spot in software security. Components provide "free" function codes, but developers usually do not have a supply chain list of all the components they are using. Therefore, developers are rarely able to follow up on component security updates. In order to maintain the stability of the software, some applications even rely on some third-party components or basic libraries that have been eliminated.

Developers will not stop using third-party components for development efficiency, and users cannot rely on the open source community to keep the components in a state of no vulnerability. Therefore, for the application software we use, a clear understanding of all component components and supply chain relationships is conducive to scanning for known vulnerabilities and discovering whether there is a known vulnerability risk in the application software.

This paper aims to scan binary files and automatically identify the third-party components contained in the binary files and the multi-level supply chain relationship between the components. Further, based
on the differences in basic blocks between different versions of third-party components, we identify un-repaired vulnerabilities in the supply chain of binary files.

2. Related work
Frequent software supply chain security incidents in recent years have caused scholars to pay attention to software supply chain security. The current software supply chain security is mainly aimed at the detection and defense of supply chain pollution, and a lot of theory and case analysis have been carried out.

Liu Quan [2] proposed to carry out a software product safety review, to study and control the safety risk of the software supply chain. The key point is to check the risks of each link in the software life cycle and improve the transparency of the software supply chain. In addition, it also mentioned that a perfect supply chain management system should be established to complement the supply chain management shortcomings.

Abel Yeboah-Ofori et al [3] studied supply chain threats in network facilities and used the STIX threat model to model attacks. In the attack case against the smart grid case, they analyzed and evaluated the proposed model. The experiments used conditional probabilities to determine attack propagation and cascading effects, and the results show that their method effectively models threats.

3. The Proposed Method

3.1. Multi-Level Supply Chain Analysis
The composition of the software supply chain is reflected in the dependence on upstream code and third-party libraries, and the upstream code and third-party libraries may also have dependencies on other components, making the composition of the supply chain more complicated. For example, a third-party component of an application software may also use multiple third-party subcomponents, and each third-party subcomponent further references other third-party subcomponents to form a multi-level dependency relationship and supply chain network.

In order to solve the above problems, we propose a multi-level supply chain analysis method. As shown in Figure 1, given a binary file, we mainly obtain its multi-level supply chain dependencies through three steps: function extraction, function retrieval, and deep dependency analysis. The process of each step is described in detail below.

**Function extraction.** For the input binary file, first disassemble it to get the assembly code. Extract its basic block and control flow graph.

**Function retrieval.** Iterate through the control flow graphs of all functions in the input file, and search in the known function library in turn to check whether there is a similar control flow graph in the function library. If there is, it is considered that the binary file code uses a known component, Return the information of the function control flow graph, and proceed to the next deep dependency evaluation.

**Depth dependency evaluation.** For the control flow graph retrieved by the function, further check its third-party library dependencies. Then find the source code corresponding to the function, detect the references and calling relationships present in it, list the other third-party components and basic library functions it references, and return the result.

---

Fig. 1. Software supply chain analysis process
Through the above three steps, for a given binary file, the function composition can be automatically disassembled, and the multi-level supply chain relationship can be retrieved and returned.

### 3.2. Software defect analysis

Security databases such as CVE and NVD can maintain a public list of vulnerabilities, providing detailed information on known vulnerabilities. Although some known vulnerabilities only have a minor impact, however, vulnerabilities such as Heartbleed are triggered by known vulnerabilities that have been published in components or basic libraries. When a version of a component is included in the vulnerability database list, it means that the component or the code that references the component is at risk of being attacked until the developer fixes the vulnerability and releases a new version of the code.

The upstream code base will release the bug-fixed code in the latest version, but it cannot affect the code that has been previously used by developers. Therefore, the application software used by users is more likely to exist vulnerabilities in their software supply chain and have been disclosed. According to Ponemon research [4], 58% of cyber attacks involve at least one known and unfixed vulnerability.

When a code of a component release version has a vulnerability, the developer will fix the vulnerability and release a new version. After compilation, the code changes will be reflected in the assembly code and reflected in the differences in the control flow graph. For functions with vulnerabilities, find the version with vulnerabilities and the version that has been repaired, and compare the differences between the set of basic blocks to find the basic blocks that may cause vulnerabilities.

![Fig 2. Vulnerability patch example](image)

Figure 2 shows an example of a security patch in C language. Where + and - represent adding or deleting a line of code. In this example, the developer believes that there is a buffer overflow problem in the original code, so a check of the length of the str parameter was added to lines 3 to 5 in the figure. In the original function, the incoming parameter str of any length will be accepted. When the length of str exceeds the buffer space, it will cause a buffer overflow problem. After adding the length check, the input space of the incoming parameter str is limited. So as to ensure that the length of str is within a safe range. When the length of str is greater than MAX_LEN, it will cause the function to return an error code -1, preventing the code from further execution. In the case of normal input, the patch will not affect the operation of the program and return results.

Figure 2 shows the control flow diagrams of this code compiled before and after patching. The left picture is the control flow diagram before patching, the right picture is the control flow diagram after patching. Blue in the figure indicates that the original basic blocks in original function, red indicates the basic blocks deleted in the original function, and green indicates the basic blocks newly added in the original function. As can be seen from the figure, the check of the parameter length introduces the red conditional jump branch in the figure. The source code modification is reflected in the addition, deletion, and modification operations of the basic block. Compare the differences between different versions of the same software, we need to compare the basic block differences between the two versions of the
software. For a specific version of the code, we can determine whether the version of the code has been patched by analyzing whether it contains a basic block for bug fixes.

Before repair

After repair

Fig 3. The difference between the vulnerability version and the patch version in the genetic map

Based on the above ideas, this section proposes a method for detecting known vulnerabilities in the supply chain to analyze whether a software contains known vulnerabilities and defects. As shown in Figure 3, the method is mainly divided into three parts: vulnerability information collection, basic block difference analysis between versions, and basic block analysis of functions.

Function A
Function B
Function C

Vulnerability library

Fig 4. Detection methods for known vulnerabilities in the supply chain

Vulnerability information collection. Find vulnerabilities in open source components from open public vulnerability libraries such as CVE, save the version of the vulnerability, the function that caused the vulnerability, and other information, and map it to the control flow graph in the function library.

Analysis of basic block differences between versions. For the library functions with vulnerabilities, compare the basic block set \( G_v \) of the vulnerabilities version and the basic block set \( G_p \) of the version after the repair to determine the changes of basic block additions and deletions. The basic block added in the function control flow is \( G_{add} = G_p - (G_v \cap G_p) \), the basic block deleted in the function control flow graph is fixed is \( G_{del} = G_v - (G_v \cap G_p) \). The two basic block sets \( G_{add} \) and \( G_{del} \) can represent the basic block differences between the two versions.

Basic block analysis of functions. Perform a supply chain analysis on the binary file. When there is a function similar to the control flow graph of the vulnerability function in the file to be detected, all the basic blocks of the function are extracted and further analyzed. Check whether the \( add_{pv} \) basic block and \( del_{pv} \) basic block exist in the function control flow graph. If there is an \( add_{pv} \) basic block and there is no \( del_{pv} \) basic block, it means that the current function version has deleted the basic block that may trigger the vulnerability, and by adding the basic block of vulnerability patches, it is considered that the version of the function has been repaired and is safe, otherwise it is considered that there may be an unfixed vulnerability and it is unsafe.

4. Experiment

4.1. Analysis case of software supply chain

In order to verify the software supply chain analysis method proposed in this article, we conducted a case study on the open source project SafeBoardMessenger [5] of GitHub. In order to construct a function library data set that can analyze the software supply chain of the SafeBoardMessenger project, according to the source code of the SafeBoardMessenger project, we downloaded all third-party source code that may be included in the project from GitHub, compiled and stored it in the function library.
Compile and disassemble SafeBoardMessenger into a function control flow graph, and analyze its supply chain to obtain SafeBoardMessenger's software supply chain relationship as shown in Figure 4. The code of the software contains the source code of the three open source projects OpenSSL, libstrophe and libstrophe. For further in-depth dependency analysis, the project relies on 66 basic libraries such as atomic and thread.

The above experiment proves that when the third-party function data resources in the function library are rich enough, for any binary file to be detected, the supply chain composition structure of the binary file can be completely obtained.

4.2. Software defect analysis case

This experiment verified the effectiveness of the software defect analysis method in this paper through an example of a vulnerability in the SafeBoard Messenger project.

On September 10, 2019, the OpenSSL team submitted the OpenSSL vulnerability CVE-2019-1547 [low severity] in the CVE vulnerability library. The vulnerability could cause the ECDSA signed key to be recovered by an attacker. The vulnerability is caused by the EC_GROUP_set_generator function in the crypto/ec/ec_lib.c file in the OpenSSL project. The OpenSSL official has fixed this defect in the three versions of 1.1.1d, and added the code to fix the vulnerability in the function that caused the vulnerability, but the code of the OpenSSL-1.1.1-1.1.1c version is still affected by this defect. By comparing the basic blocks of 1.1.1d (fixed version) and 1.1.1c (defective version), the EC_GROUP_set_generator function reduces a total of 6 basic blocks, denoted as $G_{del}$, and adds 8 basic blocks, denoted as $G_{add}$.

Comparing the basic block set from OpenSSL in SafeBoardMessenger, it is found that in the control flow graph matching the function with the vulnerability, there are 6 basic blocks in the $G_{del}$ set, and there are no 8 basic blocks in the $G_{add}$ set, indicating that the OpenSSL code imported in this project is a version with CVE-2019-1547 vulnerability. Therefore we conclude that the SafeBoardMessenger project will be affected by the known vulnerability CVE-2019-1547.

The above experiments prove that the method proposed in this paper can effectively detect the known loopholes in the supply chain.
5. Discussion
This paper analyzes the composition of components in the software through function similarity comparison; collects the vulnerabilities and versions of known open source code; by analyzing the basic block differences between versions, judges the component versions imported by the software and evaluates whether there are known risk of vulnerabilities. Experiments prove that the method is effective.

This paper provides a solution for how to identify, manage, and evaluate third-party components, and understand the security risks associated with their supply chain. It solves the unknown risks that software may face from the perspective of the software supply chain.

Acknowledgment
This work was supported by the National Natural Science Foundation of China (Grant Nos.61802435).

References
[1] BLACK DUCK. (2017) Open source security and risk analysis report. https://www.blackducksoftware.com/open-sourcesecurity-risk-analysis-2017
[2] Quan Liu, Chao Wang. Suggestions for strengthening the security guarantee of software supply chain[J]. China Information Security,2018(11):64-66.
[3] Yebo A, Islam S. Cyber Security Threat Modeling for Supply Chain Organizational Environments[J]. future internet, 2019, 11(3)
[4] OWASP Top 10: Using Components with Known Vulnerabilities Security Vulnerability Practical Overview[A/OL]. [2018-08]https://www.immuniweb.com/blog/OWASP-using-components-with-known-vulnerabilities.html
[5] OpenSSL Vulnerabilities[A/OL]. https://www.openssl.org/news/vulnerabilities.html.