Research on Detection of Abnormal Software Code

Li Xufang[1, 2, a]

[1]School of Management, Shanghai University of Engineering Science, Shanghai, 201620, China

[2]Management Science and Engineering Postdoctoral Station, University of Shanghai for Science and Technology, Shanghai, 200093, China

[a]lucylxf@163.com

Keywords: Code quality; Abnormal pattern; Anomaly detection; Null pointer dereference.

Abstract. The code structure of large-scale software is complex, so it is difficult to analyze and debug errors. Software code reliability has attracted wide attention. In order to improve the quality of the code, this paper summarizes the common abnormal patterns in large-scale software, and designs the framework for anomaly detection by using the commercial software Coverity to analyze linux-3.12. It is realized successfully to identify and detect the abnormal pattern of null pointer dereference. And through experimental verification, the results are more accurate, can be used in large-scale software.

Introduction

Software reliability refers to the ability of software to not cause system failure under specified conditions and time. The application itself has higher and higher reliability requirements for system operation. In some key application such as defence and aerospace, its reliability is particularly important. Coverity, a development and testing service provider, scans a large number of open source and closed-source projects each year to assess its quality and find defects. Coverity once scanned and analyzed more than 700 samples of open source C/C + + programs and closed source enterprise software projects, then found that mean defect density of open source C/C + + project was 0.59, mean defect density of closed source enterprise software projects was 0.72. At the same time, there are still a large number of high-risk vulnerabilities, including resource leakage, memory conflict, illegal memory access, uninitialized variables and so on.

The traditional method of improving code reliability is formalized and software testing, but the two methods also have drawbacks. There is a new way to improve code quality, which is to insert logs in the code at the abnormal point that can cause a software failure. Logs can provide some useful information to users or programmers to trace the root cause of software errors conveniently. The premise to log in is to know where the abnormal points are in the software code. This paper studies anomaly detection, which provides important support for adding logs in source code.

Large-scale Software Anomaly Detection

Anomaly Pattern Analysis. Use Coverity to run linux-3.12 source : cov-analyze --dir / home /ywx /server /test /linux -3.12. The following shows some basic information after running the kernel source by using Coverity.

Analysis summary report:

```
    --------------------------------------------------------------------
    Files analyzed: 13163
    Total LoC input to cov-analyze: 8252232
    Functions analyzed: 281082
    Paths analyzed: 21108373
    Time taken by Coverity analysis: 02:42:57
    Defect occurrences found: 6148 Total
```

From the above, we can see that after nearly three hours of scanning, detection and analysis, all the files of linux-3.12 were run, and 281,082 functions were analyzed, and 6148 anomalies were detected.

For each abnormal pattern in the source code, Coverity can provide its occurrence frequency and impact level. According to the results of Coverity, the common anomaly patterns in large-scale software are summarized by considering the factors such as frequency and level of abnormal patterns, as shown in Table 1.

Table 1 Abnormal patterns information list

Abnormal pattern	Impact level	Occurrence frequency
Null Pointer Dereference	high	649
Memory Management Error	high	380
Out of Buffer/String Bounds	high	820
Command Injection	medium	265
Path Operation	medium	268
Type Mismatch	high	639
Double Free	high	347
Uninitialized	high	524
Resource Leak	high	219
Negative Array Index	high	127
Malloc Anomaly	high	514
memset() error	high	128
Dead Code	medium	795
Infinite Loop	high	117

Set Up Anomaly Detection Framework. Almost all of the abnormal patterns are memory-related, and almost all of the memory problems are caused by the exception of the pointer. The lifecycle of a pointer generally undergoes the following processes, declaration, allocation, initialization, using, and release. In the process of analysis, the operation of the pointer in the code is mapped to the above five elements. Using the principle of finite state automata, the state of pointer type variables is maintained and the state migration is checked to meet the safe use of memory.

Anomaly detection framework is shown in Fig.1. The framework is mainly for the following memory-related abnormalities: use after free, type mismatch, resource leak, uninitialized point, null pointer dereference, out of buffer/string bounds, negative array index, allocation size mismatch, double free, and unused variable. In the construction of the framework, all the above abnormalities are divided into two categories: one occurs during the life cycle advancement process, the other occurs during some period of the life cycle.

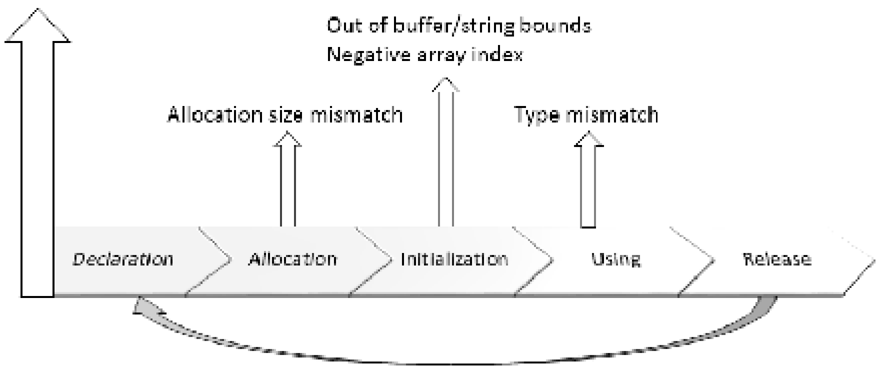


Fig. 1 Anomaly Detection Framework

From Fig.1, the following information can be got. If the pointer does not allocate space after the declaration, the abnormal pattern of null pointer dereference occurs, that is to say, after the declaration section, the allocation section has been passed directly to the initialization section or the using section.

If the pointer is not initialized after allocating the memory, then the abnormal pattern of uninitialized pointer/variable occurs, that is to say, the initialization section has been passed directly to the using section. If the same pointer from the release section to the release section, the abnormal pattern of release twice occurs. While the abnormal pattern of type mismatch occurs during the using section, and the abnormal pattern of the allocation size mismatch occurs during the allocation section, and so on.

Anomaly Detection of Null Pointer Dereference

Using the API provided by Clang, consider the impact level and occurrence frequency of the abnormal pattern, aiming at one specific exception mode: null pointer reference, design the appropriate algorithm, and implement the anomaly detection.

Anomaly Detection Algorithm. In the process of code implementation, the definition and acquisition of null Pointers are mainly considered as follows. The first type is a direct display of the null pointer when defined (or assigned). The second type is to get the null pointer through a function call. There are also two main types of null pointer dereferences. The first is to assign pointers directly. The second is a null pointer as a function call parameters.

The discovery of the null pointer dereference anomaly requires a certain control flow analysis. Clang provides a control interface for controlling the flow graph. A control flow graph can be generated for each function to help traverse the path of each program and find the exception.

During the anomaly detection process, each time a function definition is scanned, the CFG corresponding to the function is generated and the entire CFG is traversed from the first block. If the definition or assignment of a null pointer is found during the traversal process, the node is recorded. Then the node as the root node, traverse each path, in the path to detect whether there is a null pointer dereference anomaly occurred. If an anomaly is found, the location is recorded and the corresponding anomaly reminder is given.

For empty pointers that are obtained through function calls, special handling is required. There are two cases: one is obtained through the definition, such as int * a = foo (), foo () function return null in its definition. For such a statement, Clang provides an interface called DeclStmt to represent , Which can be called the DeclStmt node; another case is obtained by assigning a pointer, such as a = foo (), the same, foo () function return null in its definition, this statement can be called the BinaryOperator node.

For the first case, whenever an DeclStmt node is encountered, such as int * a = foo (), get the object it defines (i.e. a). Next, get its first child node, that is, the object that was initialized at definition (i.e. foo ()) and determine if it is a function call node called as CallExpr. If the current CallExpr node corresponding function return null in the definition, the object that the DeclStmt node defines is stored as the object that the null pointer defines, for the follow-up work of null pointer dereference judgment.

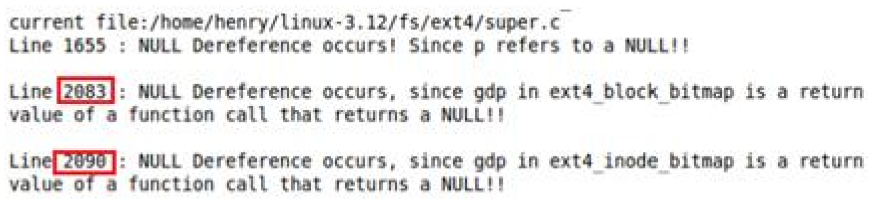
For the second case, whenever a BinaryOperator node (such as a = foo ()) is encountered, it is determined whether its right value is a CallExpr. If so, continue to determine whether the function corresponding to CallExpr return null in its definition. If the result of the judgment shows that the function corresponding to CallExpr returns null, the left value of the function is recorded.

Finally, when judging whether a null pointer is misused, it is need to determine whether the pointer is null before the reference.

Test Results. After a smooth test of linux-3.12 source code, it took 99m15.007s to find the first type of null pointer dereference occurred 273 times, and the second type of null pointer dereference occurred 351 times, a total of 624 times.

Anomaly detection implement the null pointer recognition and detection in code successful. In addition, by analyzing the ext4 module of the file system, Coverity found that the number of null pointer dereferences in the first type is much less than the second type, which is similar to the result of our anomaly detection. This is also true in the actual code writing process. If the programmer directly

defines a null pointer, the pointer will be judged before be used. If the null pointer is returned by a function, it is far more likely to forget to judge the pointer is not null than the first situation. Comparing the anomaly detection with Coverity, the test result is relatively accurate. And the results show that the anomaly detection can be applied to large-scale software.
The following is a screenshot of the abnormal test results, which successfully identified the exceptions of the 2083 lines and the 2090 null-pointer references in the program.



```
current file:/home/henry/linux-3.12/fs/ext4/super.c
Line 1655 : NULL Dereference occurs! Since p refers to a NULL!!

Line 2083 : NULL Dereference occurs, since gdp in ext4_block_bitmap is a return
value of a function call that returns a NULL!!

Line 2090 : NULL Dereference occurs, since gdp in ext4_inode_bitmap is a return
value of a function call that returns a NULL!!
```

Fig.2 Test Results

Conclusions

At present, the reliability of large-scale software is paid more and more attention. The following three tasks are mainly completed in the paper:
(1) Using the linux-3.12 source code as the target code, and using Coverity to scan the code, and based on the detection results, analyzed and summarized several abnormal patterns common in large-scale software.
(2) Based on abnormal mode analysis, the basic framework of anomaly detection is set up.
(3) For the abnormal mode of "null pointer dereference", the anomaly detection of large software is realized. Compared with the results of Coverity, the test result is relatively accurate.
In order to improve the quality of software code, adding some log information to the source code is a more mainstream and effective method. When a system failure occurs, the recorded error or warning message can often be used as the primary basis for helping to assess and diagnose the root cause of the failure. Automatically add logs, enhance the log code and log analysis are all worth further study of the direction.

Acknowledgements

This work was financially supported by the National Natural Science Youth Foundation of China (71503163), and Philosophy Social Science Youth Foundation of Shanghai (2015EGL007).

References

[1] Dominic Fandrey. Clang/LLVM Maturity Report[J].In proceedings of the Summer 2010 Research Seminar. June 2010,(9):55-56.
[2] V. R. Basili, and R. W. Sellby. Comparing the Effectiveness of Software Testing[J]. IEEE Transactionson Software Engineering. 1987, SE-13 (12) : 1278 - 1296.
[3] University of Applied Science in Rapperswil. LLVM Architecture of clang Analyze an open source compiler based on LLVM Christopher[R],2011,06.
[4] Dimitrov Martin, Zhou Huiyang.Unified architectural support for soft-error protection or software bug detection[C].PACT 2007:73-82.
[5] Murtaza, etc.. Total ADS: Automated software anomaly detection system[J]. Proc. - IEEE Int. Working Conf. Source Code Anal. Manip., SCAM,2014,12:83-88.
[6] Fang Wenhua,etc..Object detection in low-resolution image via sparse representation[C]. MMM2015:234-245.