Towards Discovery of Differences of Source Code Through Semantic Difference Checking

Qingdi Han† and Wu Gao†

1 College of Information Science and Engineering, Ocean University of China, Qingdao, Shandong, 266100, China
2 Shandong Taian Zero Distance Learning Center, China University of Petroleum, Qingdao, Shandong, 266580, China
*Corresponding author’s e-mail: hqd@stu.ouc.edu.cn

Abstract. Source code difference analysis plays an important role in software development, maintenance and regression testing. Many existing research methods are text-based, however, they are subject to code movement and alignment limitations, resulting in imprecise difference analysis. In addition, some methods are lexical and syntax-based, which ignore the key semantic information of the code and make the analysis results imprecise. In this paper, we propose a source code difference analysis technique based on semantic difference checking to improve the problem of inaccurate difference analysis. We implement the prototype, and exploit static analysis and textual difference detection tools to pro-process the programs, then perform semantic difference checking on the code with textual differences, and define the semantic differences. We conduct experiments on two C programs with differences and the results show that our approach can effectively and accurately detect code differences.

1. Introduction
Software maintenance has been recognized as the most difficult, costly and long-lasting phase in the software development life cycle, where more than 50% of all maintenance costs arise from changing software[1][2][3][4]. Changes may come from new requirements of user, fixing bugs or faults, change requests and so on. However, partial changes present high risks. They will inevitably have some unexpected side effects, may introduce new bugs, and may lead to more instability, rather than improve the software[4]. The source code difference analysis technique detects differences in different versions of the same software and identifies the parts of the code that have changed. It has an important role in software development, maintenance and regression testing.

In traditional source code difference analysis, which includes techniques such as text comparison based on lines of code[5][6] and code difference analysis based on abstract syntax trees[7][8], it can be affected by code movement, making the difference analysis inaccurate due to misalignment of the compared code during the comparison process. In addition, the traditional lexical and syntax-base source code difference analysis techniques[9][10][11][12] often ignore the key semantic information of the code, such as control flow information and data dependency information, making the analysis results less accurate.

In order to alleviate the above problems, this paper proposes a source code difference technique based on semantic difference checking. In this paper, we take function as the research granularity, and in order to improve the efficiency of the analysis, we first pro-process the two programs to get the functions
with textual differences, and then perform semantic difference checking on the functions with textual differences. If there exists an instruction with semantic difference between the two functions, it means that the two functions are different. We performed an experimental study on C program. The results show that the proposed technique can accurately and efficiently check the differences between the two programs.

The remainder of the paper is organized as follows: Section 2 gives an overview of our method and the detailed design of our technology. Section 3 presents the experimental study we performed and discusses the results we obtained. Finally, we give in section 4 a conclusion and some future work directions.

2. Approach overview and design
In this section, we first give an overview of our approach, and then describe the details of the four phases in our approach.

2.1. Approach overview
To mitigate the above issues, we propose a new source code difference analysis technique that using semantic equivalence check to find the differences. Unlike the syntax-based difference detection method, this method detects code differences from the semantic point of view and improves the accuracy of code differences. To reduce the time to analyze redundant information, the method first finds the functions with textual differences and analyzes only the functions with differences.

The overall architecture of our approach as shown in figure 1. It mainly consists of four phase, i.e., the pro-process phase, the extract functions phase, the semantic difference check phase and the visualize the differences phase.

2.2. Pro-process phase
This phase is to pro-process the two programs (original program and modified program) by a static LLVM-based analysis method and a textual difference checking tool. If there are textual differences in the two programs, then extract the functions with text difference. And then perform semantic difference detection on these functions, and finally visualize the difference. On the contrary, if there is no textual difference, then visualize the difference directly. If there are differences between the two programs, the added functions, deleted functions and modified functions are marked with different colors.
redundant work. Therefore, we use text differences tool to process programs. When the comparison results are the same, go to the next step to check the text differences, otherwise modify the two programs until the comparison results are the same. If there are textual differences in the two programs, then go to the next phase to extract the functions with textual differences.

Getting the number of callees and callees of the programs through a LLVM pass. The method is described in Algorithm 1. The LLVM IR of program as the input of the Algorithm 1, and the outputs include function names, the number of callees and the callees of each functions. The variable “count” counts the number of callees to each function (line1). Firstly, traverse the functions of the program and output the functions (line2-3). Secondly, traverse instructions in each basic block in the function (line5-6). If it is the call instruction, the “count” adds 1 and output the callees of the function (line7-8). When the function is traversed, output the “count” (line12).

Algorithm 1 Get the NOC and callees for each function of the program P.

| Input: LLVM IR of the program, e.g. ProgA.bc; |
| Output: functions & the number of callees of everyone function & callees; |
| 1. count = 0; // initialize count |
| 2. for functions of the P do //each function |
| 3. return functions; |
| 4. for Basicblock &bb:F do //each basicblock |
| 5. for Instruction &ii:bb do //each instruction |
| 6. if &ii is the call instruction then |
| 7. count++; |
| 8. return callees; |
| 9. end if |
| 10. end for |
| 11. end for |
| 12. return count; |
| 13. end for |

2.3. Extract functions phase
The phase is to extract these functions that have text differences. There are some special cases, for example, the header file issues required for function execution, the function calls other functions, and the processing of the global variables in the function. The difference check is performed at the granularity of the function, and only focuses on the functionality of the function, so the conditions should be the same. For example, if the two function call the same function, then treat the callee function as the same value, and regardless of whether the functionality of the callee is the same. We only detect the semantic difference of the current function. The detailed handling of these three cases is as follows.

- Case1: header files. Add the header files that the function depends on to the source program of the function so that the function can execute normally.
- Case2: call other functions. When the function calls another function, treat the return value of the callee function as a constant with the same type as its function (e.g., int add(), treat the return result of add() as a constant "1"; bool flag(), treat the return result of flag() as a determined value "true").
- Case3: global variables. When the function includes global variables, treat the global variable as a constant with the same data type.

2.4. Semantic difference check phase
The phase is to check the semantic difference of the two functions. Specifically, as long as the semantic difference is detected, then the two functions are different.

For functions with the same input, it is not precise to only pay attention to whether the output value is the same to detect the semantic difference of the two functions, just like the black-box method. Because some functions have no output or return values. And checking inside the black box has been
used to expose faults complementing the traditional black box output checking approach[14]. Therefore our idea is to compare the change sequence of addresses and values of the variables when the function pairs are executed with the same inputs.

Executing the two functions with the same inputs, and logging the changes of the addresses and values of the variables. The reason for comparing the change of the memory address of the variable is to avoid the difference caused by the different variable name with same behavior. For example, function add defines two variables a=1, b=2; function add' defines two variables c=1, d=2. The memory addresses of the variables (e.g. a and c, b and d) in these two functions are the same. And it mainly records the changes of variables that determine the functionality of the function, rather than recording the changes of all variables. Comparing the change sequence of the record variables. If the record change sequence of the variable values and memory addresses of the function A is \( t_1, t_2, \ldots, t_n \), and the function B is \( t'_1, t'_2, \ldots, t'_n \). We define a suitable notion of semantically difference:

**Definition 1.** Function A is semantically difference to function B if there is a change sequence of the variables value and memory addresses different, when execution of A is \( t_1, t_2, \ldots, t_n \), then when the B is executed with the different sequence of the variables value that is \( t'_1, t'_2, \ldots, t'_n \).

According to Definition 1, we can conclude that if these two change sequences are different, then the two functions are different. Our goal is to find the differences between the two functions, so it is to be marked as long as there is a difference. Conversely if there is no difference, it does not mean that the two functions are semantically equivalent.

For functions with loops, we use the black box method to record changes of variable values. Like the example in [15], for loop X and loop Y, we record the changes of the values of variables i, x and k, and only record the values of variables before entering the loop body and after exiting the loop body. That avoids the situation where there are too many loop iterations, and too many the changes of variables need to be recorded.

Our method capture the differences between the two functions accurately. By comparing the value change sequences of the recorded variables, it can figure out at which step the variable value is changed. So in this respect it is more accurate than the black box method. For the result that the input is the same, and the output is different of the two functions, there are several cases are as follows:

1) **Change instructions:** in this case, the instructions of the two functions have changed. And the black box method can only conclude that the two functions are different by observing the output value, and cannot figure out where the two functions are different, but our method can do this.

2) **Change the order of instructions:** in this case, the instructions have not changed, but the order of the instructions has changed. For example, there are two instructions in A: \( a=a^3, a=a/3 \); and two instructions in B: \( a=a/3, a=a^3 \). We know that if two variables are of integer type, the "/" is an integer division operation. Thus the order of the instructions is very important. For the B function, the value of the variable "a" will lose precision, which will cause the final result to be different. Our method can detect this difference, while the black box method cannot. It also applies to addition and subtraction of unsigned integers. For unsigned integer operations, there is no problem if you add first and then subtract, but sometimes it will be wrong to subtract first and then add.

3) **Add or remove instructions:** in this case, the new version of the program adds or removes instructions. For an simple example, in function A: \( a=1, a=a-1 \); while in the function B: \( a=1, a=a-1, a=a+3 \). The sequence of changes in the value of variable “a” of the function A is \( \{1, 0\} \). And the sequence of changes in the value of variable “a” of the function B is \( \{1, 0, 3\} \). Obviously, function B adds an instruction for variable “a”.

For the above three cases that cause different output for the same inputs, our method can locate the difference more accurately than the black box method.

2.5. **Visualize differences phase**

The phase is to visualize the differences between the two programs. Our method is to take two programs as input and output the visual differences between the two programs. As mentioned in Section 1, our method is based on function granularity for program comparison, so the differences include function
addition, removal and modification in the new version program. Based on the difference results from the first and third phases, the method will display the differences in function units. And the instruction to call the difference function will be displayed.

3. Experimental

In this paper, we implement the source code difference analysis method based on semantic differences, and experiment with the two C programs in figure 2.

![Figure 2](image_url)

The functions of the two programs and the callees of each function are obtained separately by static analysis, as shown in figure 3. This information shows that the `mul` function has been removed from the new version of the program and the `squ` function has been added. Removing the two functions from the two programs. And a text difference check of the two pro-processed programs shows that there is a difference in the `sub` function of the two programs. Then semantic difference detection of `sub` function of both programs and the results are shown in table 1. To unify the conditions, we turn off the address space layout randomization strategy (ASLR). The two columns of “Memory address” in table 1 indicate the change of memory addresses of variables when the two programs are executed. When the memory address in the two programs and the value stored in that memory address are not the same for each change of the same variable, then the variable has changed, just like the variable “b” in the table 1. The sequence of changes of variable “b” in the two programs is different. Obviously, the sequence of change of variable “b” in `sub` is \{3,6\}, while the sequence of change of variable “b” in `sub’ is \{3,9\}. Therefore, it can be concluded that the sub function has changed in both programs.

![Figure 3](image_url)
In summary, the \textit{mul} is removed and the \textit{squ} is added in the pro-processing stage, and the \textit{sub} is modified in the semantic difference checking stage. Visualize these differences separately, as shown in figure 4. The modified functions and their affected statements are marked in blue, the added functions and their affected statements are marked in red and the removed functions and their affected statements are marked in green.

| Variable | Memory address | Value | Variable | Memory address | Value |
|----------|----------------|-------|----------|----------------|-------|
| a        | 0x7fffffff9bc  | 2     | a'       | 0x7fffffff9bc | 2     |
| b        | 0x7fffffff9b8 | 3     | b'       | 0x7fffffff9b8 | 3     |
| b        | 0x7fffffff9b8 | 6     | b'       | 0x7fffffff9b8 | 9     |
| sum      | 0x7fffffff9c4 | -6    | sum'     | 0x7fffffff9c4 | -9    |

Figure 4 Visualization of the difference results between the two programs.

In this paper, we use these two C programs as experimental examples to get the differences between the two programs precisely and efficiently. This method not only gets the functions with differences, but also detects the part of the code affected by these differences, such as the instructions that call the modified functions.

4. Conclusions

In this paper, we study the limitations of accuracy in the existing source code difference analysis methods. To mitigate these limitations, we propose a novel source code difference technique based on semantic difference checking, which first detects the functions with textual differences between two programs, and then detects the semantic differences between them. The first step is to improve the analysis efficiency, and the second step is to improve the accuracy of the analysis results. We implement the approach and experiment with C programs. The results show that our method is effective in detecting differences in the source code of the two programs.
Acknowledgments
First of all, I would like to thank my supervisor, Mr. Qu. He has helped me a lot in my research and life. Secondly, I would like to thank my classmates for giving me helpful suggestions during the paper writing. Finally, I would like to thank my friends for accompanying me through this period of research.

References
[1] Lee, M., Offutt, A. J., Alexander, R. T. (2000) Algorithmic analysis of the impacts of changes to object-oriented software. In: Proceedings of the 34th International Conference on Technology of Object Oriented Languages and Systems (TOOLS 34). Santa Barbara, USA. pp. 61–70.
[2] Li, W., Henry, S. (1995) Maintenance support for object-oriented programs. In: Journal of Software Maintenance: Research and Practice; 7(2):131–147.
[3] Schneidewind NF. (1987) The state of software maintenance. IEEE Transactions on Software Engineering; 13(3):303–310.
[4] Li, B., Sun, X., Leung, H., Zhang, S. (2013) A survey of code-based change impact analysis techniques. Softw. Test. Verif. Reliab., 23: 613-646.
[5] Roy, C. K., Cordy, J. R.(2008) NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization, In: 16th IEEE International Conference on Program Comprehension, Amsterdam, Netherlands, pp. 172-181.
[6] Lee S., Jeong I.(2005) SDD: High performance code clone detection system for large scale source code. In: Proc. of the Companion to the 20th Annual ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications. ACM. 140–141.
[7] Wei H, Li M.(2017) Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proc. of the 26th Int’l Joint Conf. on Artificial Intelligence. AAAI Press. 3034–3040.
[8] White M, Tufano M, Vendome C, Poshvyvanyk D.(2016) Deep learning code fragments for code clone detection. In: Proc. of the 31st IEEE/ACM Int’l Conf. on Automated Software Engineering. 87–98.
[9] Wang P.(2018) CCAligner: A token based large-gap clone detector. In: Proc. of the 40th Int’l Conf. On Software Engineering. ACM. 1066–1077.
[10] Li L, Feng H, Zhuang W, Meng N, Ryder B. (2017)CCCLearner: A deep learning-based clone detection approach. In: Proc. of the Int’l Conf. on Software Maintenance and Evolution (ICSME). IEEE. 249–260.
[11] Raheja K, Tekchandani R.(2013) An emerging approach towards code clone detection: Metric based approach on byte code. In: Int’l Journal of Advanced Research in Computer Science and Software Engineering,3(5).
[12] Hotta K, Yang J, Higo Y, Kusumoto S.(2014) How accurate is coarse-grained clone detection: Comparison with fine-grained detectors. In: Proc. of the Electronic Communication of the European Association of Software Science and Technology. 63.
[13] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Pâsăreanu. (2008) Differential symbolic execution. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. Association for Computing Machinery, New York, NY, USA, 226–237.
[14] Tao X., Notkin, D. (2004) Checking inside the black box: regression testing based on value spectra differences, In: 20th IEEE International Conference on Software Maintenance, Proceedings., Chicago, IL, USA, pp. 28-37.
[15] Sharma R., Schkufza E., Churchill B., and Aiken A. (2013) Data-driven equivalence checking. In: Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications (OOPSLA ’13). Association for Computing Machinery, New York, NY, USA, 391–406.