The Influences of Compiler Optimization on Binary Files Similarity Detection

Hui Chen [1, a]

[1] School of Computer Science and Technology Shandong Yingcai University, Jinan, 250104, China

[a]email: chh2@163.com

Keywords: Clone Detection; Compiler Optimization; Binary File Comparison

Abstract. Binary files are generated after compilation. To analyze the similarity detection of binary files, the influences of compiler optimization should be considered. We tested generated binary files with similarity detection tools through the reclassification of clone types and the utility of different compiler optimization. The results show that the similarity detection is less influenced when benchmark files and object files adopting the same compiler optimization; and declined when adopting the different compiler optimization; in addition, the influence of compiler optimization is slightly different according to various clone types.

Introduction

Nowadays more mature software similarity detecting technologies are based on source code, but when the source code is unavailable, using binary files to have the comparison of the similarity is particularly important. Binary files are obtained from compilation. To analyze the similarity detection of binary files often needs decompilation [1]. Because of the asymmetry of compilation and decompilation, tiny changes of source code may lead to great changes of object code after compilation. Moreover, the optimization methods of compiler can also affect the forms of the object code. Even the same resource code may create different binary codes in different compiler environments. So the influence of the compiler cannot be ignored in the analysis of the similarity of binary files. All of the current technologies of similarity detection of binary files didn't consider this problem, and the insufficient comparison analysis it caused is one of the reasons of missing detection. So this paper studies the influence of the similarity detection of binary files from compiler optimization.

Binary File Similarity Detection Techniques

There are four general categories of technologies of similarity detection of binary files. The comparing technology based on binary bytes [2] is directly reading in binary data for comparison. The comparison of assembler instructions based on disassembling of binary files is the extraction of the operation codes and operands in assembler instructions for comparison. The graphical comparison based on similarities of instructions [3] is isomorphism matching of instruction graphics and the whole function structure, which begins to analyze from entry address to compare every instruction in the graphics. The comparison of structured signatures [4] is considering the whole executable file as a graphic and taking functions as basic logic units, which distributes a structured signature and uses the signatures to compare and identify the relationship between the functions.

Because the comparing technologies based on binary bytes and those of text based on disassembling of binary files all lack understanding of the overall logic of programs and are only applicable to a few changes, so there are no tools generally used. The graphical comparison based on similarities of instructions and those based on structured signatures have already been mainstream detecting technologies which has been realized and become adaptable tools. Among them, the representative tools of similarity detection of binary files are IDACompare based on the technology of signature scanning, Patchdiff based on Call Graph isomorphism comparing

technology, TurboDiff based on graphical comparing technology.

Compiler Optimization

Binary files are obtained from compilation. And code optimization plays an important role in the compiler system. Compiler optimization is improving the performance of programs through formal deductions to the programs. It utilizes various compiler optimization technologies to reduce the redundant computation of programs and to save storage space and energy consumption, or to make full use of the computing potential of target architecture [5]. Compiler optimization adopts the technologies of control flow analysis, data flow analysis, and dependence analysis and so on. Main types of the optimization are: redundant operations deletion, common subexpression elimination, unswitching, and reduction in strength, loop control conditions conversion and dead assignment deletion.

There are three different levels in optimization: local optimization, loop optimization and global optimization. Local optimization is applying expression precomputation and subexpression abstraction to the optimization transformation of small routine which only includes a few statements. Loop optimization contains reduction in strength and code unswitching, which is the optimization of the codes inside the loop. Global optimization is a large-scale applied to a program element, such as optimization transformation of a function or a process. Various changes to the codes provided by compiler optimization must comply with the following three principles. In equivalency principle, it shouldn't change the result of the program and must keep the meaning of the original program unchanged. In the principle of effectiveness, object code produced after optimization has shorter runtime and takes up less storage space. In economy principle, it achieves better effects at a lower cost as far as possible. Optimization is divided into intermediate codes optimization and object codes optimization according to the optimizing phases. Intermediate codes optimizing of them don't depend on specific computers, main jobs of which are common expression deletion, loop optimization (unswitching, reduction in strength, loop control conditions conversion, combination of known quantities and so on), copy propagation, dead assignment deletion and so on. The compiler optimization involved in this article is the intermediate codes optimization.

Experiments Settings

Experimental Data Selection. Reference [6] concludes clone transformation means into four types. From Type I to Type IV, the plagiarism means adopted are gradually complex and the difficulty of source code analysis and detection also gradually increase. During the process of source files compiled to binary files, some clone transformation means are eliminated. Thus such clone transformation means are excluded in the experimental data selection. In order to obtain more universal experimental data, this paper reclassifies clone transformation means into four type according to the influence on basic blocks. The experiment is constructed with the four concluded clone transformation means, i.e. (Table 1). Type I mainly adopts means of renaming; Type II mainly adopts means of redundant codes; Type III mainly adopts means of changing branches; Type IV mainly adopts means of changing loop statements.

Table 1 Clone Types

Clone Types	Operation Description
Type I	Renaming the function name
	Renaming the parameter name
	Renaming the variable name
	Constants replacement
	Renaming self-defining data type name
Type II	Adding redundant variables

	Adding redundant statements
	Adding redundant parameters in functions
	Adding redundant empty function, return no value, never call
	Adding redundant empty function, return value, never call
	Adding redundant empty function, return value, call
Type III	Conditional expressions changing to if statements
	If statements equivalently transforming into conditional expressions
	Multiple if statement consolidated into conditional expression and located in while determining condition
	After the determining condition changes, the contents of if and else exchange
Type IV	While statement equivalently transforming into for statement
	While statement equivalently transforming into do…while statement
	For statement equivalently transforming into while statement
	For statement equivalently transforming into do…while statement
	Do…while statement equivalently transforming into for statement
	Do…while statement equivalently transforming into while statement

Compiler Optimization Settings. In the experiment, four types of source codes of object files are designed according to the classification in Table 1. The binary benchmark files and object files are compiled with the following five methods with MS Visual C++ 6.0. Method 1: The benchmark and object files both adopt default compiler option, namely compiler optimization is not conducted. Method 2: The benchmark and object files both adopt code optimization settings, and choose Maximize Speed to generate the quickest code. Method 3: The benchmark files adopt default compiler option and compiler optimization is not conducted; the object files adopt code optimization settings, and choose Maximize Speed to generate the quickest code. Method 4: The benchmark files adopt code optimization settings, and choose Maximize Speed to generate the quickest code; the object files adopt default compiler option, and compiler optimization is not conducted. Method 5: the benchmark and object files adopt different code optimization option, among which the benchmark files adopt Maximize Speed to generate the quickest code and the object files adopt Minimize Size to generate programs of minimum size.

Decision Function. The experiment adopts binary file similarity detection tool such as IDACompare, Patchdiff and TurboDiff to detect the benchmark files and object files. The three selected tools all can offer the function numbers of the detected files and the corresponding function similarity is also offered. Thus, the decision function [7] this paper chooses is as follows:

$$S_k(A,B) = \frac{|C_k|}{m_k \vee n_k} \tag{1}$$

Where C_k indicates the collection of similar functions between benchmark file A and object file B detected by testing tool k; m_k and n_k indicates the function numbers in benchmark file A and object file B detected by testing tool k; $S_k(A, B)$ indicates the similarity between binary benchmark file A and object file B detected by testing tool k; and similarity S_k meets $S_k(A, B)= S_k(B, A)$.

Experimental Results and Analysis

Situation 1: Comparing the detected similarity of optimized benchmark files and object files or not optimized benchmark files and object files. The similarity detected with same clone transformation means with Method 1 and Method 2 in Table 2 is compared. When clone transformation Type I and Type IV are adopted, the detected file similarities of optimized compiler and not optimized compiler are basically the same. When clone transformation Type II is adopted, the file similarity of optimized compiler is slightly higher than the file similarity of not optimized compiler detected with tools PatchDiff and TurboDiff. When clone transformation Type III is adopted, the file similarity of optimized compiler is slightly lower than the file similarity of not

optimized compiler detected with tool IDACompare.

Situation 2: Comparing each of the benchmark file and object file being optimized and the other not being optimized. The similarity detected with same clone transformation means with Method 3 and Method 4 in Table 2 is compared. The similarities detected are basically the same, but the similarities detected are 2.13%-6.11% lower than the similarities detected with Method 1 and Method 2. When each of the benchmark file and object file adopt compiler optimization and the other doesn't, the total function numbers detected in both binary files are different and the similar function numbers are small.

Situation 3: Comparing both benchmark file and object file being optimized, but the similarities are detected with different optimization option. The similarities detected with Method 5 are 0.48%-3.06% higher than those detected with Method 3 and Method 4, which adopt the same clone transformation means, but they are 1.66%-5.33% lower than those detected with Method 1 and Method 2. When both benchmark file and object file adopt compiler optimization, the function numbers detected in both binary files are the same, but the similar function numbers are smaller than those detected with Method 1 and Method 2.

Table 2 Results of Similarity Detection

Tools	Clone Types	Method 1	Method 2	Method 3	Method 4	Method 5
IDACompare	Type I	100%	100%	95.68%	95.68%	97.10%
	Type II	92.55%	92.45%	89.38%	89.38%	90.57%
	Type III	93.75%	91.82%	89.38%	89.38%	89.94%
	Type IV	92.50%	92.45%	89.34%	89.38%	89.94%
PatchDiff	Type I	100%	100%	94.12%	94.12%	94.67%
	Type II	97.97%	98.97%	92.86%	93.40%	94.92%
	Type III	96.92%	96.91%	94.36%	94.36%	94.85%
	Type IV	97.95%	97.94%	93.85%	93.33%	94.85%
TurboDiff	Type I	95.32%	95.27%	92.98%	92.98%	93.53%
	Type II	96.55%	97.01%	91.37%	92.42%	92.89%
	Type III	95.92%	95.90%	93.37%	93.37%	93.85%
	Type IV	96.94%	97.95%	93.37%	93.37%	93.85%

The identifiers such as function names, parameter names and variable names self-defined by the users are transformed into token in the compiler scanning process, which won't influence the program execution effect. Thus, the noise brought by clone transformation Type I is eliminated after compilation. In Table 2, the similarities of binary files of Type I detected with Method 1 and Method 2 are higher, while the similarities of binary files of Type I detected with Method 3, Method 4 and Method 5 are lower than those detected with Method 1 and Method 2. Thus, the detected similarities of clone transformation Type I are influenced when object file and benchmark file adopt different compiler optimizations.

The compiler optimization normally deletes the redundant codes or chooses quicker but larger code sequences to create quicker or smaller binary executable files. Type II mainly adopts means of redundant codes. In Table 2, the similarities of clone transformation Type II are lower than those of other clone transformation Types with Method 3 and Method 4, and are also lower than those detected with Method 1, Method 2 and Method 5 for the called redundant functions can change the division of basic blocks, add meaningless blocks or function signatures can be disturbed. Thus, the detected similarities of clone transformation Type II are greatly influenced by compiler optimizations.

The compiler optimization can eliminate common subexpressions, decrease calculation intensity, optimize circulation and jump and transform the equivalent expressions into unified forms, but meanwhile the optimization settings may cause code rearrangements and the same function codes in different forms. In Table 2, the similarities of Type III and Type IV detected with Method 3, Method

4 and Method 5 are lower than those detected with Method 1 and Method 2. Thus, the detected similarities of clone transformation Type III and Type IV are influenced by compiler optimizations.

Conclusion

This paper researches the influences of compiler optimization on the similarity detection of binary files. In the experimental results, the similarity detection is less influenced when the benchmark and object files adopt the same compiler optimization; the detected similarity decreases and the influences on different clone transformation types are slightly different when the benchmark and object files adopt different compiler optimization schemes. Due to the limitations of the author's knowledge and ability and the experiment range, the conclusion may not be comprehensive. The future research should further study how to decrease the influence of compiler optimization on the similarity detection of binary files.

Acknowledgement

This research is supported by research grants from Yingcai #12YCYBZR04.

References

[1] I. J. Davis M. W. Godfrey, Clone detection by exploiting assembler, Proceedings of the 4th International Workshop on Software Clones. ACM, 2010， p.77-78.
[2] A. Schulman, Finding binary clones with opstrings and function digests, Dr Dobb's Journal-Software Tools for the Professional Programmer, 2005, p.64-70.
[3] T. Dullien, R. Rolles, Graph-based comparison of executable objects, SSTIC '05, 2005, p.1-3.
[4] H. Flake, Structural Comparison of Executable Objects, In Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment, 2004, p.161–173.
[5] R. Allen, K. Kennedy, Optimizing compilers for modern architectures, San Francisco: Morgan Kaufmann, 2002.
[6] C.Roy, J.Cordy, R.Koschkeb, Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach, Science of Computer Programming, 2009, p. 470-495.
[7] H. Chen, T. Guo, B. J. Cui, Comparison and Analysis on Binary File Similarity Detection Technique, Computer Engineering and Application, vol.48, 2012, p.79-84