Research on Android Intelligent Terminal for Code Obfuscation Evaluation Algorithm

Li-kai WANG¹,a,* and Zhi-dong ZHAO²

¹School of Electronic and Information, Hangzhou Dianzi University, Hangzhou, China
²Hangdian Smart City Research Center of Zhejiang Province, Hangzhou Dianzi University, Hangzhou, China

a506024987@qq.com
*Corresponding author

Keywords: Code obfuscation assessment, Android smart platform, Reverse engineering, BF function jump, Software security.

Abstract. Purpose: The code obfuscation is a software protection technique that can effectively increase the cost of reverse analysis and attack of attackers. However, the effectiveness evaluation and verification of the obfuscation algorithm have important problems to be solved in the study of code obfuscation. Currently, most researches on the effectiveness of code obfuscation are based on software complexity. Code obfuscation, as a technology to protect software security, needs to be evaluated from the perspective of reverse attack. Method: This paper introduces the idea of reverse engineering into the evaluation of code obfuscation algorithm. In this paper, a modified algorithm based on BF algorithm is proposed. The garbage instruction in the branch function (BF) algorithm is used to reposition the jump address of the BF function and hide the secondary jump address. The new algorithm improves the defects in BF algorithm and increases the obfuscation complexity of the BF algorithm for static disassembly. Results and Conclusions: The experiments are conducted on the Android smart terminal environment. The static disassembly tool IDAPro is used to test the obfuscation of BF algorithm and the obfuscation of the new algorithm.

Introduction

With the popularization and rapid development of mobile internet and mobile intelligent terminal, the number of mobile applications exploded. Among them, the Android platform intelligent terminal and its applications occupy the dominant position in the market. However, face to the growing application market, the situation of adding malicious code after the application has been pirated, and it is becoming more and more serious. Some crackers use these malicious programs to collect user privacy and tamper with data. It brings huge financial and security threats to users. The code protection and intellectual property protection for mobile applications need to be strengthened [1].

The development language for Android applications is Java. The Java source code is compiled to generate an intermediate code that retains a large amount of information about the program. The code that is decompiled can basically reconstruct the program. This puts Android applications at great risks for intellectual property protection. The code protection technology can prevent applications from being pirated and protecting intellectual property rights. The code obfuscation has its own advantages over other protection technologies [2]. On the one hand, the code obfuscation technology protects the code of the program. It doesn't change the functionality of the program implementation, even though the legitimate users can't
get important information from the obfuscated programs. On the other hand, the code obfuscation has a more active protection. It is difficult to solve the main algorithm and the data structure from the obfuscated program.

For applications, it is based on the analysis of the program structure that the crackers can pirate or add malicious code. Confusion about data and structure can't be very protective. The crackers can still reconstruct the program structure. Only by controlling flow code obfuscation and hiding or changing its control flow structure can protect the intellectual property rights of the application better [3]. Early studies for the code obfuscation mainly focus on constructing effective confusing algorithms. Collberg [4] divides code obfuscation techniques into four categories: Layout Code obfuscation, Control Code obfuscation, Data Code obfuscation and Preventive Code obfuscation. Among them, the research of controlling flow code obfuscation algorithm is relatively more, such as opaque predicate code obfuscation algorithm [5] and Cloakware Company [6]'s flat controlling flow code obfuscation algorithm. Currently, the researches on the construction of code obfuscation algorithms is are relatively mature. However, the validity evaluation of the code obfuscation algorithm lacks the corresponding theoretical supports. Barak [7] theoretically prove that code obfuscation technology can not completely guarantee the security of the information in the terminal operation without hardware auxiliary protection mechanism. However, Appel [8] has also proved that anti-aliasing is a NP hard problem. But between these two extremes, there is almost no valid theory to prove it. The extent to which code obfuscation slows down the analysis of programs lacks quantitative ways to measure its effectiveness. How to accurately measure the protection effect of code obfuscation and ensure the validity of the code obfuscation transformation is one of the most important issues in academia and industry.

Code Obfuscation Algorithm Effectiveness Evaluation

Basic Idea of Effectiveness Evaluation

The purpose of a software attack is to collect as much information about the internal state of the program as possible [9]. The internal state information can be inherent in the program by static information. It also can be the dynamic information of the program, such as assembly instructions, control flow and data flow generated after disassembly techniques. To break the software, attackers often use reverse engineering techniques. For example, static analysis and dynamic tracking are used to analyze the programmer's thoughts and obtain confidential data and core algorithms. The specific steps are shown in Fig. 1.

First, disassembly techniques are used to convert binary byte code into easily understanding assembly instructions;

Then, the static analysis technology is used to analyze the control flow and data flow of the program effectively. At the same time, the efficiency and accuracy of analysis are improved through dynamic tracking.

Finally, the source code is obtained by decompiling technique.
However, in order to ensure that the state information inside the program is not available to the attacker, different code obfuscation algorithms have been proposed for the reverse analysis of attackers and the techniques adopted in each phase. In this paper, we use the code of reverse engineering to evaluate the effectiveness of the algorithm. The details are shown in Fig. 2 (P: A program that will be reversed, P\(^{-1}\): program that is generated after reverse analysis).

The attribute space is defined for binary executable file to characterize internal information of the programs. In this paper, the attribution space contains three attributes: instruction, controlling flow and data stream.

The reverse analyses on the procedures are performed before and after obfuscation. The key links are disassembly, controlling flow analysis and data flow analysis.

In the process of reverse analysis, as shown in Figure 2, the indexes that can describe the characteristics of program attributes are extracted. According to the assembly instructions generated after the disassembly, the evaluation index which can represents the characteristics of program instruction. It can be extracted from instruction execution rate. According to the result of controlling flow analysis, the
evaluation index which can characterize the flow of program control is extracted: cyclomatic complexity of controlling flow. According to the result of data flow analysis, the evaluation index which can represent the characteristics of program data stream is extracted: the complexity of fan in / fan out.

Comparing the changes of the three indexes before and after the obfuscation, the evaluation of the effectiveness of the code obfuscation algorithm was realized.

**Obfuscation Validity Evaluation Index**

In the evaluation of the effectiveness of code obfuscation, we introduce reverse engineering. From this view, three kinds of effectiveness evaluation indexes are mainly used [10]:

**Disassembly: Instruction Execution Rate.** When the attacker analyzes the assembly instructions, the object of dynamic analysis is the actual program execution path rather than the path that cannot be performed. Dynamic analysis provides results that are executed and must be accurate. The features extracted by static analysis are approximate features of the system when it is running. Therefore, the directive execution rate can be introduced to describe the characteristics of attribute instruction of the program:

Instruction execution rate (IE): The number of assembly instructions actually executed by taking the percentage of the number of assembly instructions generated after the disassembly. $I_s$ represents all instructions generated after data disassembly. $I_d$ represents the number of instructions that are actually executed during dynamic analysis.

$$IE = I_d / I_s.$$  \hspace{1cm} (1)

The Eq. 1 is the calculation formula used to describe the characteristics of a attribute instruction I of the program.

$$Character(x) = IE = I_d / I_s; x \in (i^{+1},(i)^{+1})$$ \hspace{1cm} (2)

**Controlling Flow Analysis: Cyclomatic Complexity of Controlling Flow.** The orders of execution in the instructions are important. The property controlling flow of a program is used to reflect these orders. The controlling flow graph (CFG) is the basis of computational complexity in McCabe [11]. Through the construction of CFG, the order of execution of the statement in the subroutine can be determined. To build CFG, the basic block of the subroutine must be determined first. A basic block corresponds to a node in CFG. It is the maximum set of consecutive statement. In this statement, the control can only start with first statements. It only can be transferred from the last statement by conditional branch or non-conditional branch. When the first sentence of the basic block is executed, all statements in the basis blocks will be executed. The cyclomatic complexity of controlling flow is $V(G)$, and the calculation formula is as follows:

$$V(G) = e - n + 2.$$ \hspace{1cm} (3)

where $e$ represents the number of edges in the controlling flow diagram, and $n$ represents the number of nodes in the controlling flow diagram. $V(G)$ is a formula for describing the characteristics of the attributes control flow of a program.

$$Character(x) = V(G) = e - n + 2; x \in (c^{+1},c^{+1})$$ \hspace{1cm} (4)

**Data Flow Analysis: The Complexity of Fan in/Fan out.** Data flow analysis is
an indispensable stage in the process of reverse analysis [12]. Typically, an attacker collects information of data flow D about a program's runtime. The relationship between data objects in the program will be analyzed to get the specific algorithm of the program. Data flow analysis focuses on the usage, definition and dependencies of data in a program. These indexes are important for determining the logical components of the system and their interactions. Data flow analysis is much more complex than controlling flow analysis. For example, the controlling flow analysis only needs to analyze the possibility of the loop, while the data flow must determine the variation of the variables in the loop. Data flow analysis can also obtain the information with lower requirement of the level of abstraction, such as process dependencies, dependencies between variables and data related to the modification of code segments. In order to increase the difficulty of the data analysis, a variety of data flow obfuscation algorithms are introduced. Such as array dimension change, function splitting and class splitting and merging. Henry proposes a method for measuring the data flow complexity by using the fan input/fan out of the module [13].

Fan-in: The fan-in of a module refers to the sum of data flow into the module; Fan-out: The fan-out of a module is the sum of the data flow that the module outputs. The formula of fan input/fan out complexity is shown below:

$$DC = (fan-in \times fan-out)^2$$

(5)

DC is the description of calculation formula of data flow attribute D in Eq. 1:

$$Character(x) = DC = (fan-in \times fan-out)^2; x \in (d^{-1}, (d)^{-1})$$

(6)

An Improved Algorithm Based on BF Algorithm

Principle and Algorithm Designation

In the original BF algorithm [14], the garbage instruction block uses invalid instruction and random data to finish the confusion of data disassembly. The introduction of garbage orders presents an important overhead problem. These instructions increase the file size of the program itself. A serious shortage in the BF algorithm is that all CALL instructions that replaced with jumps to the BF instructions will jump to a fixed address, which is the entry address of the BF. When BF returns, it does not return to the next line of the CALL instructions. These abnormal behaviors may be detected in the process of disassembly to locate BF function's location and internal logic. This allows the attacker to believe that the address returned by the BF function must be the same of the code actually started.

Obviously, the solution to this problem is simply to replace the BF function address of the CALL instruction with an opaque expression. In this way, the static disassembly cannot get the specific address of the jump, so that the internal logic of BF function cannot be known. The idea of a secondary jump [15] is based on the assumption that the BF function will always go straight to the correct code execution address where it returns. In this way, the data disassembly will find the address for the correct disassembly through such a loophole. Now that the data disassembly considers that the execution address is correct, and jumping to the junk instruction block raises the difficulty of the data disassembly. After the BF function returns, it does not jump directly to the instruction address that the “JMP” instruction should jump to, but jumps to the secondary jump address in the garbage instruction. Because of the instruction's address is dynamically set and jump, the usage deformation of the
secondary jump instruction effectively improves the difficulty of locating the BF function address of the data disassembly.

Implementation of Algorithm Design

In the implementation of the algorithm, it is important to determine the location and number of “JMP” instructions that need to be replaced with BF function. This concerns the intensity of obfuscation and the magnitude of performance intensity. It is important to ensure that the insertion of the garbage instructions and the controlling flow of the program can be effectively obfuscated. The overall framework of this algorithm is shown in Fig. 3. The garbage command block is inserted between the “CALL BF” command and the actual jump address “addr”. The beginning and ending addresses of the garbage instructions are “addr1 and addr2”. The garbage instruction size is the difference between “addr1 and addr2”. It can be divided into three parts according to the different implementation functions.

![Figure 3. Overall frame diagram of the algorithm.](image)

Initialization of Garbage Instruction Block and Transformation Parameter

In the garbage command block, the location of the “JMP” instruction for the secondary jump can be defined according to the actual requirement. Generally speaking, the garbage instructions are required before and after the jump instruction. As shown in Fig. 4, the secondary jump instruction is located in the garbage instruction, which causes the second jump instruction to be wrongly parsed and unable to get the real jump address correctly. The program can locate the instruction address of the secondary jump correctly when it is actually executed. The information for a secondary jump can be hidden effectively.
The ROA (return offset address) of the BF function can be obtained after determined the offset of the “JMP” instruction to the start position of the garbage instruction block.

\[ \text{ROA} = \text{offset} \quad (7) \]

The value of the transfer parameter of the BF function is associated with two variables. These two variables are ROA and RV (return value) (the “addr” shown in Figure 4). The function of the variable ROA is to change the return address of the BF function, so that the function can jump directly to the address of the secondary jump. The variable RV is the destination address of the secondary jump. The value is defined as an integer type (4 bytes). The structure of the value is shown in Fig. 5.

The value is divided into two low-order bytes and two high-order bytes (represented by \( l_{\text{value}} \) and \( h_{\text{value}} \)). These two low-order bytes and two high-order bytes are the values of RV and ROA, respectively. In order to save space and protect data information, the corresponding relationship between them is represented through the completely minimal hash function \[16\].

\[
RV = \text{hash}(l_{\text{value}}) \quad (8)
\]

\[
\text{ROA} = \text{hash}(h_{\text{value}}) \quad (9)
\]

The value will be passed to BF function through the PUSH instruction when the initialization is completed.

**Implementation of the New BF Function**

The new BF function implements three functions: Parsing the transfer parameter value;
computing the new return address of the function; transmitting RV. The add function is used to parse the two high-order bytes and the two low-order bytes after the parameter values are passed to the BF function. The ROA and the RV are obtained, respectively. The assemble code is used to calculate the real secondary jump address and pass the RV as a function return value after obtaining the ROA and RV variables. The key implementation assemble code is shown in Fig. 6.

The instruction “I1” saves the value of the offset ROA in the position of “esp + 4”. The instruction “I2” swaps the values of the “eax” and the value on the top of the stack, so that the initial address of the garbage instruction is saved in “eax”. The instruction “I3” adds the starting address to the offset ROA. The real address of the secondary jump will be obtained and saved at the location of “esp + 4”. The instruction “I4” is used to restore the initial value of “eax”. The instruction I6 function is used to return. In this way, the BF function jumps to the garbage instruction block. The controlling flow begins to execute the instructions for a secondary jump.

**Deformation and Concealment of Secondary Jump Instructions**

The address of the secondary jump is determined by the RV that passed back from the BF function. This dynamic approach can effectively count static disassembly. In order to use the garbage command block efficiently, the “JMP” instructions of the secondary jump are hidden by deformation. It is restored to the “JMP” instruction in the process of program execution. As shown in Fig. 7.

The key assemble code is shown in Fig. 8 as follow.

![Figure 6. Key assembly code results of new BF function.](image)

![Figure 7. Deformation and concealment of the secondary jump.](image)

![Figure 8. Key assembly code results of new BF function.](image)
The instruction “I1” pushes the RV into the stack after the BF function completes the return value. Since the sixteen hexadecimal code of the push “eax” is “0x50”, the instruction “0x50” and “0xB9” can be XOR calculated by the address “addr3” and the address “addr4”. That is, the “JMP” instruction in the sixteen hexadecimals codes, and the “JMP” instruction is converted to the “I5” instruction. In this way, the instruction will turn into “JMP” instruction at instruction “I5”. The instruction “I3” assigns the address of the secondary jump to “JMP” instructions. The instruction “I4” pops up the value on the top of the stack to keep the stack balanced. In this way, both the “JMP” instruction and the address of the jump are hidden. It improves the error rate of disassembly effectively combined with the use of random data in garbage instruction.

Experimental Results and Analysis

Experimental Platform Construction

In this study, application crawler technology is applied to several major application storages. After analyzing the application sample library, the application files were divided into different levels according to the code quantity [17]. The applications were tested according to the distribution of the application code quantity. Code obfuscation is applied on different levels of code quantity. The obfuscation tool is used to test the functionality of the application after its obfuscation. The functional testing tool of the IDAPro [18] focuses on whether the functions of application are consistent or not before and after obfuscation. In this paper, six different code levels and hundreds of different kinds of applications were tested. The sample selection is performed according to the code level distribution of the application in the sample library. As shown in Table 1, the sample applications have the same function before and after obfuscation. The experimental results show that the confusion tool does not change the original semantics of the application. The obfuscation of controlling flow can be implemented in functions.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>programming language</th>
<th>assembly</th>
<th>platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>C</td>
<td>gcc</td>
<td>ubuntu</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>C</td>
<td>gcc</td>
<td>ubuntu</td>
</tr>
<tr>
<td>429.mcf</td>
<td>C</td>
<td>gcc</td>
<td>ubuntu</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>C</td>
<td>gcc</td>
<td>ubuntu</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>C</td>
<td>gcc</td>
<td>ubuntu</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>C</td>
<td>gcc</td>
<td>ubuntu</td>
</tr>
</tbody>
</table>

The data for functional test can only explain the possibility of obfuscation. The static disassemble tool IDAPro is used to test the obfuscation of the BF algorithm and the obfuscation of the new algorithm. The degree of obfuscation, obfuscation intensity and performance cost are tested, respectively.
Algorithm and Evaluation Results

When generating experimental data, each test program will perform seven times of obfuscation and generate experimental reports. The highest and lowest values in the data will be removed and take the average of the remaining five. The three evaluation indicators described in Section 2 of this article will be used in the test. The BF algorithm and the above mentioned algorithm are used to obfuscate the testing program. The experimental results of these two algorithms are compared after the obfuscation.

The first is the comparison of test data on the degree of obfuscation of the testing program. The results are shown in Table 2.

Table 2. Test data comparison of the degree of obfuscation.

<table>
<thead>
<tr>
<th>program</th>
<th>BF degree of obfuscation</th>
<th>improved algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td>80.05</td>
<td>82.36</td>
</tr>
<tr>
<td>bzip2</td>
<td>83.12</td>
<td>84.56</td>
</tr>
<tr>
<td>mcf</td>
<td>75.36</td>
<td>80.47</td>
</tr>
<tr>
<td>gobmk</td>
<td>73.14</td>
<td>79.58</td>
</tr>
<tr>
<td>hmmer</td>
<td>86.56</td>
<td>85.69</td>
</tr>
<tr>
<td>sjeng</td>
<td>84.25</td>
<td>87.12</td>
</tr>
<tr>
<td>average</td>
<td>80.41</td>
<td>83.30</td>
</tr>
</tbody>
</table>

The second is the test data comparison of the testing program’s potency of obfuscation. The results are shown in Table 3.

Table 3. Test data comparison of the potency of obfuscation.

<table>
<thead>
<tr>
<th>program</th>
<th>BF potency of obfuscation</th>
<th>improved algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td>56.58</td>
<td>73.25</td>
</tr>
<tr>
<td>bzip2</td>
<td>63.25</td>
<td>85.65</td>
</tr>
<tr>
<td>mcf</td>
<td>72.57</td>
<td>79.14</td>
</tr>
<tr>
<td>gobmk</td>
<td>65.23</td>
<td>75.89</td>
</tr>
<tr>
<td>hmmer</td>
<td>54.41</td>
<td>84.71</td>
</tr>
<tr>
<td>sjeng</td>
<td>73.45</td>
<td>79.56</td>
</tr>
<tr>
<td>average</td>
<td>64.24</td>
<td>79.70</td>
</tr>
</tbody>
</table>

The finally is the test data comparison of the testing program’s performance overhead. The results are shown in Table 4.

Table 4. Test data comparison of the performance overhead.

<table>
<thead>
<tr>
<th>program</th>
<th>BF volume size</th>
<th>improved algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td>23.14</td>
<td>23.98</td>
</tr>
<tr>
<td>bzip2</td>
<td>15.36</td>
<td>85.65</td>
</tr>
<tr>
<td>mcf</td>
<td>26.78</td>
<td>16.32</td>
</tr>
<tr>
<td>gobmk</td>
<td>13.56</td>
<td>30.12</td>
</tr>
<tr>
<td>hmmer</td>
<td>19.74</td>
<td>20.15</td>
</tr>
<tr>
<td>sjeng</td>
<td>24.32</td>
<td>28.24</td>
</tr>
<tr>
<td>average</td>
<td>20.48</td>
<td>22.37</td>
</tr>
</tbody>
</table>

Analysis of Experimental Results

We can see that the BF algorithm is a very efficient algorithm according to the comparison of the above three tables. It can achieve the ideal obfuscation degree and the potency of obfuscation on the premise of the average volume of files increases by 20.48%. The improved algorithm in this paper is not greatly improved in the degree of...
obfuscation, because it is mainly treated as a kind of reinforcement obfuscation for the controlling flow (“JMP” instruction). The potency of obfuscation increases from an average of 64.24% in BF to about 80%. The size of the program file only grows about 2%. Due to the different internal logic, the results of the experiments are different in the same test conditions. For example, the test program “bzip2”, which implements compression and improves the potency of obfuscation about 23% by the improved BF algorithm. This is because the jump instructions in the program are heavily used. It makes the obfuscation of particles denser. In this article, the optional part of adding conditional control statements in the BF function is mentioned. This part is not necessary for the algorithm, but just as a complement and reinforcement of the algorithm. In the actual test, this part is added, and a comparison test was made. The test results are expressed only on the average due to a limited space. As shown in Table 5, it can be seen that the addition of conditional control improves the controlling flow obfuscation from 79.70% to 85.13%. However, the increase of the volume of program files is also notable. So, whether to add this part needs to be chosen according to the actual situation.

Table 5. Comparison test before and after adding conditional control statements.

<table>
<thead>
<tr>
<th>metrics</th>
<th>the potency of obfuscation /%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>before</td>
</tr>
<tr>
<td>$N_p$</td>
<td>83.30</td>
</tr>
<tr>
<td>$N_r$</td>
<td>79.70</td>
</tr>
<tr>
<td>$C_S$</td>
<td>22.37</td>
</tr>
</tbody>
</table>

Conclusion

With the popularity of Mobile Internet and the rapid development of mobile intelligent terminals, as well as the increasingly serious piracy, code protection and intellectual property rights of mobile applications need to be strengthened. For Android applications, only obfuscation about the controlling flow can truly protect them. Meanwhile, the Android mobile applications are encoded in the Java language. The strong type properties of Java byte code and the fact that its code and data cannot be mixed, make it easier to analyze the controlling flow of Android applications. It lays the foundation for controlling flow obfuscation of Android mobile applications.

On the basis of this background, a new evaluation model is adopted to evaluate the effectiveness of the code obfuscation algorithms. The purpose of this research is to investigate the effectiveness of the obfuscation techniques on different software by observing the changes of program attribute values in reverse processes. From this perspective, an improved obfuscation algorithm based on BF algorithm is proposed. The algorithm is mainly for the two main methods of disassembly in reverse, linear scanning and recursive reciprocation. Because of the particularity of the jump behavior of the BF algorithm, it is easy to identify and localize to the disassembly data. By using garbage instruction block to finish the secondary jump of address, not only the resynchronization of linear scanning can be delayed, but also the instruction information of secondary jump can be hided. In the last experiment of Android code obfuscation experiment platform, the comparison between the improved algorithm and the original BF algorithm is given under the three evaluation criteria. The results show that the algorithm improves the degree and potency of obfuscation under the premise of guaranteeing performance overhead.
References


