binOb+: A Framework for Potent and Stealthy Binary Obfuscation

Byoungyoung Lee, Yuna Kim, and Jong Kim
Department of Computer Science and Engineering
Pohang University of Science and Technology (POSTECH)
Hyoja-dong, Nam-gu, Pohang, Republic of Korea
{override,existion,jkim}@postech.ac.kr

ABSTRACT

Reverse engineering is the process of discovering a high-level structure and its semantics from a lower-level structure. In order to prevent malicious use of reverse engineering against binaries, various techniques have been developed called binary obfuscation. Obfuscated binary is a transformed binary which retains original binary's executing behavior while its outer representation obstructs the reverse engineering. In this paper we propose three novel approaches to improve the binary obfuscation. First we propose a generalized binary obfuscation algorithm that covers any specific or whole part of a binary code by using confusing code and redirecting control-flow using exceptions. Second, we employ a data-mining method to make our obfuscated binary look like a normal binary. And third, we address the issue that the previous techniques could not be applied to Windows binaries by designing a new exception hooking mechanism in Windows. Experimental results show that our obfuscated binary can hide 60-90% of the original instructions from reverse engineering tools, while its execution slows down a little, and moreover the obfuscated binary's stealth can be guaranteed.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement - Reverse Engineering; I.2.2 [**Automatic Programming**]: Program transformation

General Terms

Security

Keywords

Binary obfuscation, reverse engineering, exception handling, Windows SEH, stealth

ASIACCS'10 April 13–16, 2010, Beijing, China.

1. INTRODUCTION

Software applications are often distributed in an executable binary form, which makes it difficult to understand the internal behavior without source code. Nevertheless, many attackers take advantage of distributed binaries by exploiting vulnerabilities, removing copy protection, circumventing access restrictions, and more. For example, commercial software is frequently cracked by attackers who discover its vulnerable parts from binaries, and well-known internet worms can be created fundamentally based on vulnerabilities found in software distributed with IIS web server or SQL server [28, 29]. Furthermore, there are many good-quality decompiling tools that facilitate the analysis of binaries by translating binaries into source code in a high-level language.

In order to stop distributed binaries from being broken down so easily, there has been a lot of research for developing binary obfuscation techniques [5,6,14,19,30]. Binary obfuscation is a technique for altering the original code's structure without the source code, and maintaining its original functionality. A binary code that is obfuscated well is difficult to decompose into its original semantics. Obfuscation is growing more important due to newly developed software that can be easily decompiled and examined since they are mostly created in object-oriented cross platform programming languages (ex. Java, Visual Basic, C#, Python, etc.).

State-of-the-art approaches to binary obfuscation are based on the following three techniques: 1) inserting, after an unconditional control transfer instruction, 'junk bytes, which are unreachable at runtime but make disassemblers confuse alignment of instructions, 2) replacing a control transfer instruction with a signal instruction, which redirects a program's control flow to the target of the original control transfer at runtime, but hides the actual control transfer from disassemblers, and 3) encrypting the body of an IF-statement, which is decrypted and executed only when IF-condition is satisfied at runtime, but makes disassmblers interpret as data.

However, those previous techniques have weaknesses in terms of coverage and stealth of obfuscation. At first, the code block that one intends to obfuscate could not be entirely covered by the previous obfuscation techniques. In reality, the previous techniques are employed onto specific parts such as branch instructions [14, 19] and the body of if-else structure [22]; in this case, the code block that has neither branch instruction nor if-else structure could not be obfuscated. Such a partially-covered obfuscation might expose critical information related to software license, which can severely damage the software protection integrity. There-

fore we need a solution to enable obfuscation to cover the entire code block, regardless of wherever it resides.

Another weakness of the previous techniques is that stealth, which measures how difficult it is to know whether it is obfuscated or not, could not be guaranteed. Because many instructions or data bytes are added into an original binary in the obfuscation process, the obfuscated binary code inevitably shows abnormal statistical properties in distribution of instructions, the number of loops, or the usage of operands. This abnormality can be easily detected by using statistical analysis [12]. Furthermore such a non-stealthy obfuscation offers deobfuscators a clue to identify which obfuscation technique is applied to the original code, because each technique has special characteristics. For example, inserting 'junk' bytes introduces many invalid instructions observed rarely in a normal binary [14,19], and encrypting the original code introduces many data bytes observed rarely in an executable binary [22]. Once one figures out the obfuscation technique applied, a deobfuscation process would be much easier. Therefore we need a solution to hide a special characteristic of obfuscation.

In this paper, we propose a framework for fully-covered and stealthy binary obfuscation, called *binOb+*, which addresses the weaknesses mentioned above. In the framework, codes that confuse reverse engineering tools are optimally inserted so as to cover the entire part of binaries. In addition, a frequency distribution of instructions in obfuscated binary is adjusted to a non-obfuscated binary by employing the data-mining technique.

The main contribution of this paper is as follows:

- We propose a generalized obfuscation algorithm applicable to any part of binary code with good performance. The obfuscation algorithm is independent of location and frequency of particular instructions within the binary, and thereafter can obfuscate any specific or whole part of the binary code. In addition, to maximize the obfuscation potency, defined as how obscure the obfuscated binary is, the algorithm finds the most confusing code and a location where the best-confusing code should be inserted. Experimental results show that our obfuscation algorithm has 87% of obfuscation potency on average.

- We employ the data-mining approach to make an obfuscated binary look like a non-obfuscated binary. Using opcode statistics, we calculate the difference between an obfuscated binary and a set of non-obfuscated binaries, and then the obfuscated binary is manipulated to eliminate the difference. According to our experimental results, the obfuscated binary is statistically similar to the non-obfuscated binary. To the best of our knowledge, this is the first study showing importance of stealthy binary obfuscation.

- We develop a novel exception-hooking mechanism for Windows binaries to handle all exceptions raised in both the original code segment and the new code segment that have been added during obfuscation process. The implementation is based on Windows SEH (Structured Exception Handling). Our obfuscator is the first study to provide binary obfuscation for Windows binaries.

The reminder of the paper is organized as follows. Section 2 provides background information on static analysis of binary and disassembly algorithms. Section 3 presents our proposed framework for binary obfuscation in detail. Section 4 explains how to implement our obfuscator on Windows, and section 5 shows our experimental results of the obfuscation performance. Section 6 discusses some related works of binary obfuscation and Section 7 concludes the paper.

2. BACKGROUND

2.1 Static Analysis of Binary

Reverse engineering of binary can be categorized into two main classes: dynamic analysis [17, 25] and static analysis [3, 11]. Dynamic analysis is performed by monitoring executions and getting the context information in runtime. Thus it can only cover executed part of the binary and the covered part may be changed at each run according to the input parameters. It is also a complex and time-consuming task, even with automated tools. On the contrary, static analysis is performed without executing binary. It can cover the whole segment of binary and there exists many support tools that enable someone to analyze binary code in automatic way and a reasonable time. Hence, the static analysis technique is much more popular for reverse engineers.

In this regard, we focus on impeding such a static analysis technique, which is a more pressing matter than the dynamic one. Once our obfuscation succeeds in impeding the static analysis technique effectively, other complementary techniques targeting dynamic analysis, such as anti-debugging [8], can be used together.

2.2 Disassembly Algorithms

The fundamental process of static analysis is *disassembly*, that is, to get assembly code from binary. There are two widely-used disassembly algorithms [7]: linear sweep and recursive traversal. The linear sweep algorithm begins at the binary program's entry point and reads sequentially until it gets to the end of the program. The linear sweep algorithm has a critical weakness, it is susceptible to misinterpret data as code when the data is embedded in the code section. The most popular tools implementing this algorithm, called *disassemblers*, are GNU utility objdump [9] and Microsoft's DumpBin [2]. The recursive traversal algorithm explores a binary program, along with the control flow of the program. Control flow of the program is the order in which basic blocks that have one entry point and one exit point are executed. The main advantage of this algorithm is that it might follow through the actual executing flow of program. However, it suffers from indirect jumps or calls because it cannot track the executing flow beyond these instructions. Disassemblers implementing this algorithm are IDA [1] and a disassembler embedded in OllyDbg [31].

Recently, new disassembly algorithms were proposed to handle an obfuscated binary. The most outstanding work is Kruegel et al.'s exhaustive disassembler [12], which is the most sophisticated disassembler we are aware of. It first lists all possible branch instructions for each function, then examines the basic block boundaries by finding genuine branch instructions based on heuristic and statistical method. The remarkable point of this work is that it considers all possible instructions for each function and rules out the impossible

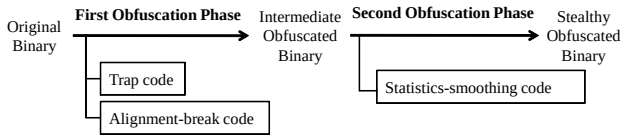


Figure 1: Two-phase obfuscation framework

ones. However, it also relies on branch instructions like recursive traversal algorithm.

3. POTENT & STEALTHY OBFUSCATION

This section will describe how to obfuscate a binary in our framework, and it consists of two phases: fully covered obfuscation and stealthy obfuscation. The first phase aims at making a binary misunderstood by the static disassemblers and the second phase aims at hiding whether a binary is obfuscated or not. After the two-phase obfuscation, we can finally obtain a potent and stealthy obfuscated binary, as shown in Figure 1. In turn, technical details of the two phases will be presented.

3.1 Fully Covered Obfuscation

We aim at making an obfuscated binary whose disassembly result is completely different from the semantics of the original binary. This goal is related to one of the obfuscation qualities called *potency* [5], which is defined as how obscure the obfuscated binary is. For the sake of this goal, we take advantage of two principles: disassembly-alignment break and control-flow redirection.

First of all, if a partial instruction could be inserted between two certain successive instructions, a disassembler would fail[1] in reading the instructions located after the partial instruction until it accidentally reached a correct alignment of instructions. In other words, a disassembler misinterprets the partial instruction as the beginning of the following instruction, and thereafter the disassembly alignment of the instructions would be shifted ahead. The reason for this alignment break is that disassemblers generally assume the next instruction will be right after the previous instruction. Hereafter, we refer to the partial instruction as *alignment-break code*, and the sequence of instructions that are mis-disassembled due to the alignment-break code are referred to as *alignment-broken area*.

In addition, in order for an *alignment-broken area* to run the same as the original binary, the execution is forced to skip over the *alignment-break code* by inserting a *trap code* right before the *alignment-break code*. As shown in Figure 2, the *trap code* deliberately raises an exception, and our exception handler catches the exception and redirects the control flow to the instruction right after *alignment-break code*. This control-flow redirection is also possible when inserting unconditional-jump instructions instead of trap codes, and the use for unconditional-jump instructions is more simple and general. However, recursive disassemblers are based on control-flow with tracking branch instructions. Therefore, our obfuscation utilizes *trap codes* for deluding the disassemblers.

In order to break the disassembly alignment of the entire code section in the binary program, the next *trap* and *alignment-break codes* are inserted right after the *alignment-broken area* which is covered with the previous *alignment-break code*. An algorithm for inserting *trap* and *alignment-break codes* is described in Algorithm 1. It starts by obtaining a sequence of instructions in the original binary (line 1) and initializing a variable named SizeofAlignBrokenArea with zero (line 2). Next, it repeats for every instruction of the original binary (line 3-10).

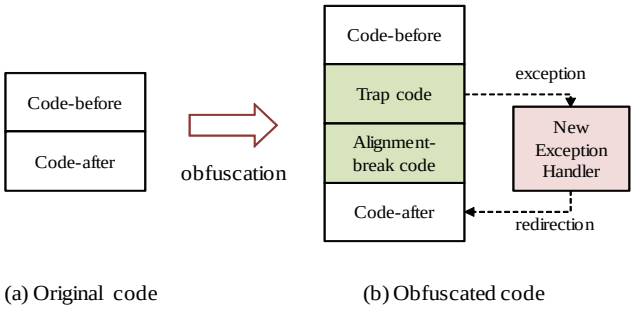


Figure 2: Insertion of *trap* and *alignment-break codes* for obfuscation

A function named GetAlignBreakCode returns two things: 1) the best alignment-break code to cover the maximum size of the *alignment-broken area* by predicting a disassembly result while testing all the combinations of two bytes, and 2) the covered size (line 5). A *trap code* is determined by calling a function GetTrapCode() (line 6). Note that this function can return a variety of trap codes so as not to stand out in static analysis[2]. The determined *trap* and *alignment-break codes* are inserted between *code-before* and *code-after* (line 7-8). For example, as shown in Figure 3, an *alignment-break code* '24' in hex byte breaks a disassembly result for the following five bytes, which is the size of the *alignment-broken area*. The part of inserting new *trap* and *alignment-break codes* (line 5-8) will be executed only if the previous *alignment-broken area* ends. As a consequence, this algorithm works to cover the entire code section and minimize the number of *trap* and *alignment-break codes*.

```
   Input: OrgBin  /* original binary */
   Output: ObfBin  /* obfuscated binary */
1  Instructions = GetSequentialInstructions(OrgBin);
2  SizeofAlignBrokenArea = 0;
3  foreach  instr ∈ Instructions do
4      if SizeofAlignBrokenArea <= 0 then
5          (AlignBreakCode,SizeofAlignBrokenArea) =
           GetAlignBreakCode(instr,Instructions);
6          TrapCode = GetTrapCode();
7          InsertInstruction(ObfBin, TrapCode);
8          InsertInstruction(ObfBin, AlignBreakCode);
9      InsertInstruction(ObfBin, instr);
10     SizeofAlignBrokenArea − = GetSize(instr);
11 return ObfBin
```

Algorithm 1: Inserting *trap code* and *alignment-break code* into binary

[1]The failure occurs only in variable-instruction-length architectures such as the IA-32.

[2]Generation of polymorphic trap codes is well explained in [19].

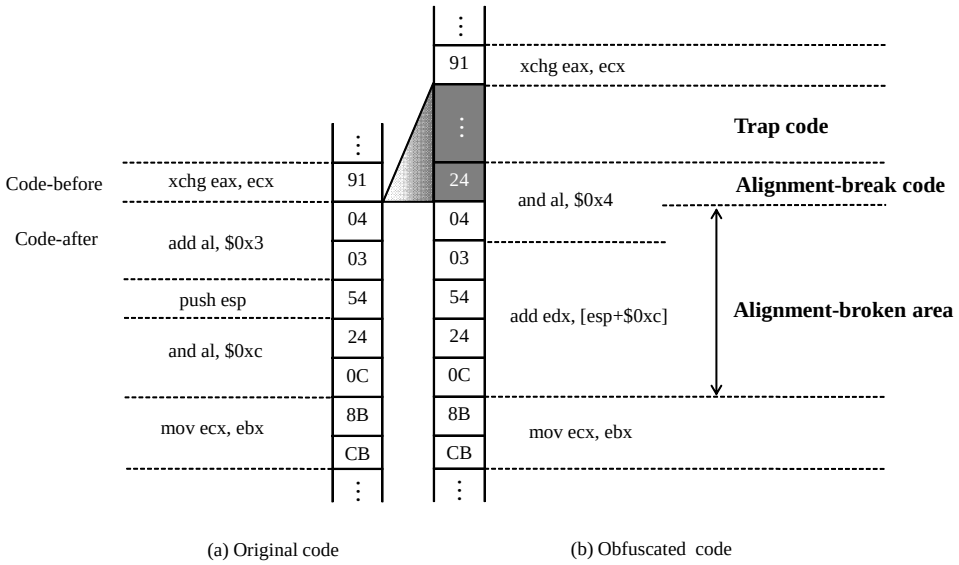


Figure 3: A snapshot of a binary code before-and-after obfuscation

3.2 Stealthy Obfuscation

In this section we present a new approach to stealthy obfuscation, which makes an obfuscated binary look like a non-obfuscated binary. *Stealth* in obfuscation measures how difficult it is to identify whether binary is obfuscated or not. This measure is quite important for the obfuscation in that a stealthy obfuscated binary can fortify the protection level of an obfuscated binary, because attackers should put more efforts to figure out the obfuscation techniques used.

However, the resultant binary of the first obfuscation phase (Section 3.1), called *intermediate obfuscated binary (iOB)*, can be obviously identified with its abnormal instruction distribution. A set of instructions disassembled from *iOB* is statistically different from a set of instructions disassembled from a non-obfuscated binary, due to disassembly-alignment break, by which unexpected or random instructions can appear. Actually, the abnormality of *iOB* may be detected by the statistics-based anomaly detection techniques that have been used in detecting malicious codes [10,24,15,16]. Therefore, we have to eliminate the abnormality of *iOB*. Note that *iOB* does not show any particular pattern that can be detected with signature-based detection techniques [13, 26], after using polymorphic *trap codes*.

In the second obfuscation phase, *iOB* is adjusted to look like non-obfuscated binary in terms of instruction distribution. At first, we estimate which and what number of instructions should be inserted to smooth the statistical difference, and then insert them into *iOB* properly. Technical details will be described in the following subsections.

3.2.1 Getting a new histogram of opcodes for stealthy obfuscated binary

In order to quantify the abnormality of an obfuscated binary, we apply a Mahalanobis distance which has been used for statistics-based malware detection [24, 21]. The Mahalanobis distance is a dissimilarity measure between two random vectors whose components are scalar-valued random variables on the same probability space [27]. In our work, the abnormality of an obfuscated binary is defined as the Mahalanobis distance between an obfuscated binary and a set of non-obfuscated binaries:

$$md(\overrightarrow{N}, \overrightarrow{X}) = \sqrt{(\overrightarrow{N} - \overrightarrow{X})^T S^{-1} (\overrightarrow{N} - \overrightarrow{X})},$$

where $\overrightarrow{N}$ is a vector, each component of which is a probability mass function (pmf) of each opcode's mean occurrence in the non-obfuscated binaries; $\overrightarrow{X}$ is a vector, each component of which is a pmf of each opcode's occurrence in the obfuscated binary; S is a diagonal matrix whose components are variances of mean values of each opcode for the non-obfuscated binaries. We can say that if a Mahalanobis distance is greater, an obfuscated binary is more abnormal.

In order to smooth the statistical abnormality of *iOB*, we have to minimize the mahalanobis distance between *iOB* and the non-obfuscated binaries. In Algorithm 2, an adjusted histogram[3] of the opcodes' occurrence in the demanding stealthy obfuscated binary is obtained so as to minimize the Mahalanobis distance from non-obfuscated binaries. The adjusted histogram is named AdjustHisto. The inputs of the algorithm are 1) IntermObfHisto, an occurrence histogram of opcodes in *iOB*, and 2) mdTh, a threshold value within which the Mahalanobis distance should be. It begins from obtaining a pmf vector of the opcodes' mean occurrence in non-obfuscated binaries and a diagonal matrix for the variance of the mean values (line 1-2). It also calculates the total number of instructions in *iOB* (line 3). Next, it tries to find an approximate histogram for the non-obfuscated binaries' histogram within the threshold value while increasing the total number of instructions little by little[4] (line 5-10). Finally it returns the approximate histogram of opcodes (line 11).

[3] A pmf can be obtained from a histogram by calculating a probability of occurrence.

[4] The incremental factor is returned from a function GetIncFactor(), which depends on obfuscation environment.

Input: IntermObfHisto /* opcode histogram of intermediate obfuscated binary */
Input: mdTh /* threshold value of Mahalanobis distance */
Output: AdjustHisto /* adjusted opcode histogram */

```
1  N⃗ = GetNonobPmf();
2  S = GetNonobVariances();
3  NumInstrs = GetNumberOfInstructions(IntermObfHisto);
4  NewNumInstrs = NumInstrs;
5  repeat
6  |  NewNumInstrs += NumInstrs * GetIncFactor();
7  |  AdjustHisto = GetApproxHisto(IntermObfHisto, NewNumInstrs);
8  |  X⃗' = GetPmf(AdjustHisto);
9  |  md = GetMahalanobisDistance( X⃗', N⃗, S);
10 until md < mdTh ;
11 return AdjustHisto
```

Algorithm 2: Finding an adjusted occurrence histogram of opcodes in the demanding stealthy obfuscated binary.

We can obtain a statistics-smoothing histogram by subtracting the input histogram from the output histogram. The statistics-smoothing histogram can be expressed with a set of (op_i, $\#op_i$), where an opcode indexed by i should be inserted $\#op_i$ times into iOB to compensate the abnormality of obfuscated binary. Note that the statistics of *trap code* and *alignment-break code* are smoothed as well since iOB contains such codes.

3.2.2 Adjusting to stealthy obfuscated binary

According to the statistics-smoothing histogram expressed by (op_i, $\#op_i$), every opcode (op_i) is inserted as many times as its paired number ($\#op_i$) between any *trap code* and *alignment-break code*, as shown in Figure 4. Insertion of an opcode means insertion of an instruction that is formed by concatenating the opcode and randomly chosen operands. Although operands could be chosen among the set of common ones in the normal binaries, but we simply selected operands in our current implementation. We leave further improvements as our future work. The instructions inserted between the *trap code* and *alignment-break code* are called *statistics-smoothing code*, which smooth the statistical abnormality of the obfuscated binary. Note that the *statistics-smoothing code* is not executed because of the control-flow redirection caused by *trap code*.

4. IMPLEMENTATION IN WINDOWS

We implemented a binary obfuscator for Windows, which is named *binOb+Win* in *Python*, with about 11,000 lines. We also implemented a new exception handler in *C*, with about 1,000 lines, and the handler is integrated into the obfuscated binary with an original binary. We will first describe our implementation architecture for *binOb+Win*, and then explain how it handles exceptions using Windows Structured Exception Handling (SEH).

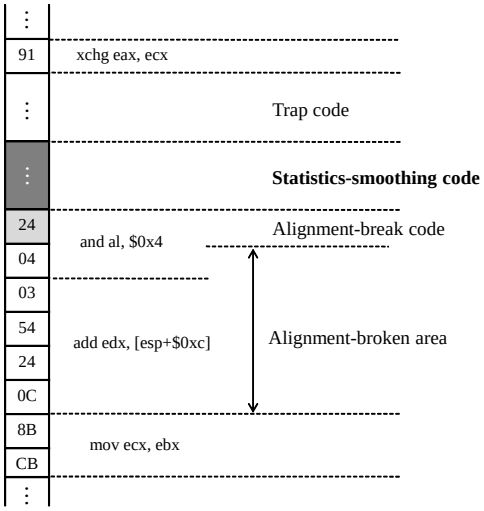


Figure 4: Adding *statistics-smoothing code*

4.1 Implementation Architecture

Figure 5 shows our implementation architecture, which takes an input of an original binary that needs to be obfuscated, and outputs a resultant stealthy obfuscated binary. First, the original binary and exception handling binary are disassembled by *IDA* disassembler [1], which generates binary information that contain both the assembly code and information of functions in two binaries. Note that we prepare in advance the exception handling binary in DLL (dynamic-link library) after compiling a new exception handler that we implemented in *C*. Next, with the binary information (labeled 1), *Obfuscation Engine* generates the intermediate obfuscated binary by inserting three kinds of codes into the original binary: 1) *trap code*, 2) *alignment-break code*, and 3) the exception-handler code, according to Algorithm 1. Next, *Statistics Smoothing Engine* takes the intermediate obfuscated binary (labeled 2) with an opcode histogram of normal binaries, and produces an adjusted histogram of opcodes in the ideal-case stealthy obfuscated binary (labeled 3), according to Algorithm 2. The histogram of normal binaries is generated in advance by analyzing 350 binaries which are default system binaries in Windows. Finally, *Obfuscation Engine* uses the adjusted opcode histogram to insert *statistics-smoothing code* and returns the stealthy obfuscated binary (labeled 4).

4.2 Exception Handling Using Windows SEH

In order to handle exceptions raised from *trap code* in obfuscated binary, we use an exception handling mechanism provided by Windows, called Structured Exception Handling (SEH) [18], instead of a signal handling mechanism. The signal handling mechanism cannot be applied for binary obfuscation in Windows, because Windows SEH has a priority to signal handling mechanisms. As a result, there can be an exception that is not handled by a signal handler, but by a pre-installed exception handler. For example, suppose that a certain program uses *Try-Except* clauses which are popular in use with the Windows C program to handle exceptions for a user-process. In such a program, *Except* block catches an exception first and then terminates the pro-

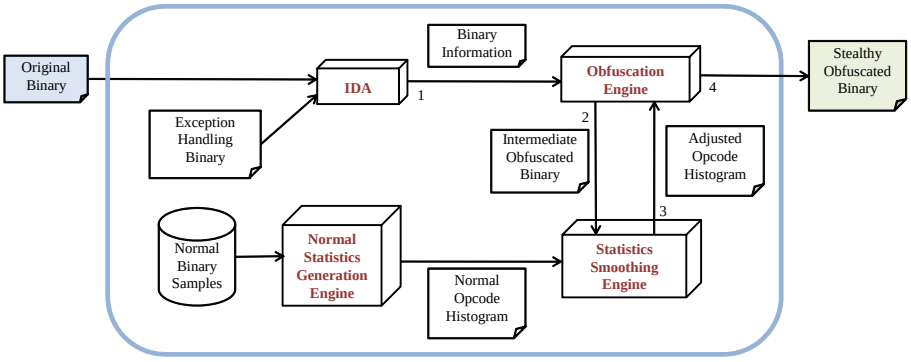
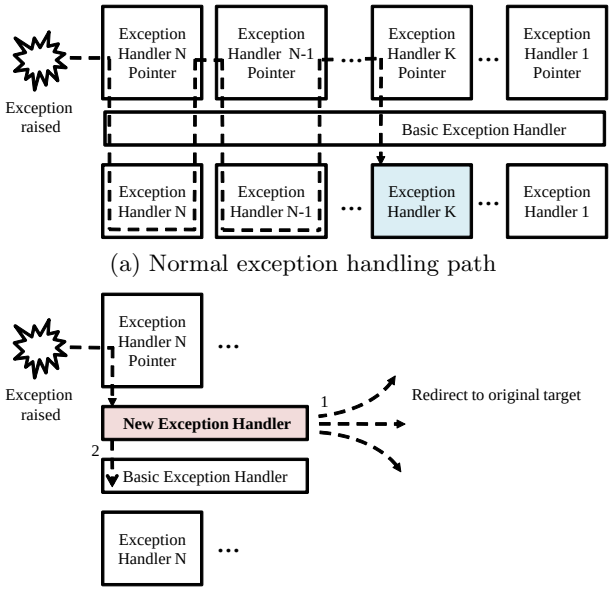


Figure 5: Implementation architecture



(a) Normal exception handling path

(b) Our exception handling path

Figure 6: Exception handling path in Windows

cess, so a signal handler will have no chance to handle the exception.

In Windows SEH mechanism, all exception handler pointers in executables are pointing to the basic exception handler. When an exception is raised, the basic exception handler redirects an exception control to an appropriate exception handler. For example, as shown in Figure 6(a), an exception that is supposed to be handled by '*Exception Handler k*' goes to '*Exception Handler k*' indirectly through the basic exception handler.

In this work, we take advantage of the indirect handling principle for handling exceptions raised from *trap code* in obfuscated binary. Our new exception handler is placed in the upper layer of the basic exception handler. As shown in Figure 6(b), our new exception handler intercepts all exceptions raised during the execution of binaries. If the exception is raised from *trap code* of the obfuscated binary, the new exception handler redirects a control-flow to *code-after* (labeled 1). Otherwise, the handler passes the exception to the basic exception handler so that it may be handled normally (labeled 2).

Technically, the handler layering is possible by hooking the basic exception handler, such as *'_except_handler3'* or *'_except_handler4'* [5], which is included in C Run-time library and integrated by Microsoft Visual C++ at link time. Such a hooking way works well for all binaries compiled with Microsoft Visual C++ because the binaries always have the basic exception handlers.

However, the use of this hooking technique produces a side effect on the obfuscation. We should not obfuscate functions in a binary, that are supposed to be executed before the basic exception handler is installed into the memory stack. The installation is usually performed at the very beginning of the binary's execution. If we obfuscate the functions, the basic handler would not be installed yet, so there would be no way to handle exceptions raised from the functions. However, the total size of excluded functions for this reason is only 530 bytes in average, which takes up less than 1% in the original binary. In addition, we believe that such functions belong to C Run-time library and usually do not contain useful information for a reverse engineer. Therefore, this side effect is not a weakness of our obfuscation.

5. EXPERIMENTAL RESULTS

Our evaluation is done in two platforms, both Linux and Windows. In order to compare the effectiveness of our approach with the Linux-based previous obfuscation method proposed by Popov et al. [19], we also implemented our binary obfuscator in Linux, named *binOb+Linux*, which uses a signal handling mechanism like Popov et al.'s work to handle exceptions from *trap codes*. Popov et al.'s work obfuscates only the limited amount of instructions starting with branch instructions by using signals.

For evaluation on Linux, we obfuscated five binary programs including 'vortex', 'bzip2', 'mcf', 'gzip ', and 'vpr' compiled with gcc version 3.2.2 at optimization level -O3. Experiments ran on 2.4 GHz Pentium IV system with 1GB of main memory and ran RedHat Linux 9. For an evaluation on Windows, we obfuscated seven well-known Windows binaries, including 'sol.exe', 'winmine.exe', 'notepad.exe', 'ipconfig.exe', 'netstat.exe', 'nslookup.exe', and 'tftp.exe', all of which are default system binaries in Windows XP. Experiments ran on 2.4 GHz Pentium IV system with 1GB of main

[5]Function names vary according to the version of Microsoft Visual C++: *'_except_handler3'* ($\leq$ MS VC++ 6.0) or *'_except_handler4'* (> MS VC++ 6.0).

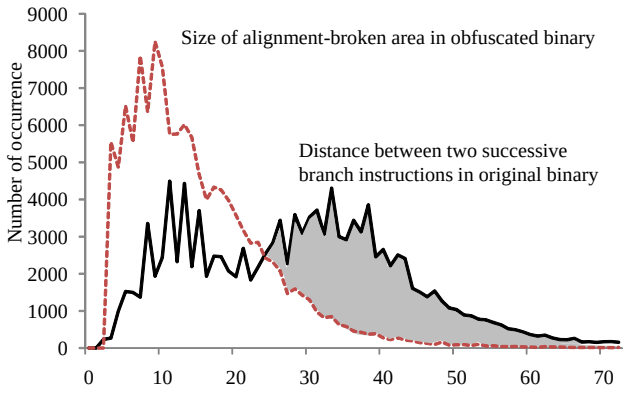


Figure 7: Size of alignment-broken area versus distance between branch instructions

memory which ran Windows XP. Each evaluation measure is described in the following subsections.

5.1 Disassembly Mismatch

We evaluate the extent of disassembly mismatch, which measures how different the disassembly result of an obfuscated binary is from the revealed semantics of its original binary. Intuitively, in order to evaluate a disassembly mismatch of instructions between an original binary and its obfuscated binary, one could calculate its difference set of disassembled instructions. More formally, this measure is expressed as *Confusion Factor*, which is used in Popov et al.'s work [19] as well.

$$\text{Confusion Factor} = |A - P|/|A|,$$

where A is the set of all actual instruction addresses, i.e., those that would be encountered when the program runs, and P is the set of all instruction addresses that are identified by the disassemblers in the obfuscated binary. Then A-P is the difference set of addresses that cannot be identified by disassemblers after obfuscation.

This confusion factor measures an instruction-by-instruction disassembly mismatch, and it can measure the disassembly mismatch for the basic blocks and functions as well. Confusion factors for basic blocks and functions are evaluated analogously; a single basic block or function is counted as being "incorrectly disassembled" if any of the instructions in it are incorrectly disassembled. Note that the confusion factor for instructions could measure how many disassembly mismatches happen, while confusion factors for basic blocks and functions could measure how well disassembly mismatches are distributed over the binary.

Table 1 summarizes the confusion factors for instructions, blocks, and functions of three obfuscators with four disassemblers. In Linux, two obfuscators, including Popov et al.'s work and *binOb+linux*, are tested with three linux-supported disassemblers, including *objdump* [9], *exhaustive* [12], and *IDA*[6] [1]. In Windows, one obfuscator, *binOb+Win*, is tested with two windows-supported disassemblers, including *Dumpbin*[7] [2] and *IDA* [1]. *Objdump* and *Dumpbin* are typical disassemblers both of which use linear sweep algorithms and work on Linux and Windows, respectively; *IDA* is the most popular disassembler for reverse engineers. It uses recursive traversal algorithms and works on both Linux and Windows; *Exhaustive* is the best-known disassembler specialized for obfuscated binary working on Linux. The values in the upper two rows (labeled as *Popov* and *binOb+Linux*) are geometric means over five linux programs. The values in the lowest row (labeled as *binOb+Win*) are geometric means over seven windows programs.

In Table 1, it turns out that our obfuscation method (*binOb+Linux*) significantly improves all of the confusion factors over the previous work (*Popov*) by 62% at best (in IDA) and 43% on average. We argue this result signifies that our obfuscation algorithm successfully covers the code sections of binary more widely than the previous work. This argument is supported by the data as shown in Figure 7, in which a dotted-line represents how many alignment-broken areas with a size of x-axis number exist in obfuscated binary and a solid-line represents how many pairs of two successive branch instructions with a distance of x-axis number exist in original binary. The gray-colored part indicates the code segments that cannot be covered in the previous work (*Popov*) because the distance between branch instructions is larger than the size of alignment-broken area.

More surprisingly, our method shows at least 83% and 95% of confusion factors for basic blocks and functions respectively, and the result implies that it succeeds in distributing disassembly mismatches over the binary. Our obfuscation method for Windows, *binOb+Win*, also maintains the superiority over the previous work (*Popov*) in terms of confusion factors, even though *binOb+Win* uses Windows SEH for handling exceptions. In the table, confusion factors in *objdump* and *Dumpbin* are lower than in the other two disassemblers, so the former disassemblers look more robust against binary obfuscation. However, this phenomenon comes from the fact that disassembled instructions are too small due to disassembly errors.

Based on our assumption, one might think that confusion factor of our approach should reach almost 100% because it could cover entire code sections of the original binary. However, the result is around 60%-90% due to the following reasons: 1) *alignment-break area* does not include any *trap code* for reducing the number of *trap codes*, which is a trade-off for the confusion factor; 2) we could not completely predict every disassembler's behavior since every disassembler has its own unique behavior; 3) we did not obfuscate some part of binary for proper exception handling. However, in spite of these reasons, our results show the confusion factor for instructions is significantly improved by around 40%-60% over Popov et al.'s work. Full data for disassembly mismatch in Linux and Windows is in Appendix A.1.

5.2 Stealth

The stealth of obfuscation measures how difficult it is to identify whether a binary is obfuscated or not. We regard the stealth of obfuscation as Mahalanobis distance (MD) between an obfuscated binary and normal binary samples as presented in Section 3.2. The opcode pmf and its variance are obtained with 350 normal binary samples, which have a mean of 3.32 and a variance of 3.17 in MD.

As shown in Table 2, our final obfuscation result, *stealthy obfuscated binary*, has a much lower MD value than *inter-*

[6]We used IDA Pro version 5.2 for evaluation.

[7]Dumpbin included in Microsoft Visual C++ package is a typical linear sweep disassembler in Windows.

Table 1: Disassembly mismatch in Linux and Windows (confusion factor, %)

		Objdump			Dumpbin			Exaustive			IDA		
		Instrs	Blocks	Funcs	Instrs	Blocks	Funcs	Instrs	Blocks	Funcs	Instrs	Blocks	Funcs
Linux	Popov (CF0)	42.72	71.13	89.24	-	-	-	57.86	63.65	88.85	57.34	63.76	95.02
	binOb+Linux (CF1)	61.52	88.18	95.85	-	-	-	71.14	83.67	97.10	92.84	95.70	97.09
	CF1/CF0	1.44	1.24	1.07	-	-	-	1.23	1.31	1.09	1.62	1.50	1.02
Windows	binOb+Win	-	-	-	65.17	89.35	96.14	-	-	-	86.30	94.08	97.60

Table 2: Mahalanobis distance of obfuscated binary from normal binaries

	Program	Mahalanobis Distance: Intermediate Obfuscated Binary (md0)	Mahalanobis Distance: Stealthy Obfuscated Binary (md1)	md1/md0
binOb+Win	sol	26.26	3.93	0.15
	winmine	26.23	4.66	0.18
	notepad	24.64	4.91	0.20
	ipconfig	27.27	5.51	0.20
	netstat	43.04	13.29	0.31
	nslookup	30.20	13.96	0.46
	tftp	24.91	3.23	0.13
	GEOM. MEAN	28.93	7.07	0.24

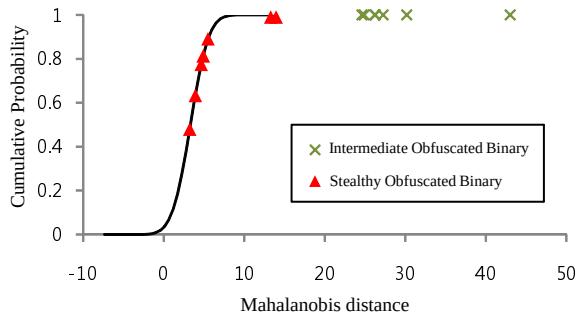


Figure 8: Cumulative density function of mahalanobis distance among normal binaries

mediate obfuscated binary which is an output of the first obfuscation phase. This result implies that our stealthy obfuscation method can successfully reduce the abnormality of an obfuscated binary.

In addition, we argue that *stealthy obfuscated binary* is statistically similar to normal binaries. To clarify the similarity, we transform the MD among normal binary samples into the normal distribution, and its cumulative probability function (cdf) is shown in Figure 8. Then we mark the MD of each obfuscated binary from normal binaries on the cdf. All marks of stealthy obfuscated binaries reside within the cdf, but those of intermediate obfuscated binaries reside outside the cdf. The variance of normal samples is similar to that of stealthy obfuscated binaries within 99% confidence interval. This result implies that attackers who want to analyze our stealthy obfuscated binary by using the statistical data-mining methods would have difficulty in distinguishing whether it is obfuscated or not.

Table 3: Effects of obfuscation on program size

(a) Program Size in Linux

	Popov			binOb+Linux		
	Code section	Data section	Combined (code+data)	Code section	Data section	combined (code+data)
obfus. / orig.	2.13	33.99	2.56	2.51	24.31	2.81

(b) Program Size in Windows

Program	orig.	binOb+Win: Intermediate Obfuscated Binary		binOb+Win: Stealthy Obfuscated Binary	
		obfus.	obfus./orig.	obfus.	obfus./orig.
sol	55,808	116,736	2.09	133,632	2.39
winmine	118,784	146,944	1.24	150,016	1.26
notepad	66,560	124,928	1.88	135,168	2.03
ipconfig	54,784	118,272	2.16	128,512	2.35
netstat	35,840	83,968	2.34	86,528	2.41
nslookup	75,776	159,744	2.11	166,912	2.20
tftp	15,872	36,352	2.29	39,424	2.48
GEOM. MEAN			2.02		2.16

obfus. : the size of obfuscated binary (in bytes)
orig. : the size of original binary (in bytes)

5.3 Program Size

Table 3(a) shows the effects of obfuscation on the program size in Linux, in terms of the increase factor. The size of a code and data section increases, because *trap code* and *alignment-break code* are inserted in the code section, and mapping tables for redirecting control-flow are inserted into the data section. In *binOb+Linux*, the size of the code section increases more than in *Popov* because we insert more *trap codes* and *alignment-break codes* to cover more widely. On the other hand, the size of the data section increases less, because the previous work should record the type of original instructions indicating whether the replaced instruction is 'return' or 'unconditional jump'. Contrary to the previous work, the information on the type of original instructions is unnecessary because no instruction is replaced. Although the increase factor in data section is over 20, the increase factor in combined sections stays below three. This is because the initial size of the data section is much smaller than the code section.

Table 3(b) shows the effects of obfuscation on the program size in Windows. We show only the combined size due to the limit of spaces, but the increase factor in each section is quite similar to Linux. The increase factor of combined size is around 2, which is similar to the increase factor in Linux. The program size of stealthy obfuscated binary is a little larger than one of intermediate obfuscated binary due

to statistics-smoothing codes.

We believe that the attacker cannot take an advantage of this feature although the program size is increased due to the obfuscation. This is because only the obfuscated binary would be released and the attacker would have no information on the matching original binary. Full data of the size of five programs obfuscated by *Popov* and *binOb+Linux* is in Appendix A.2.

5.4 Execution Speed

Table 4: Effects of obfuscation on execution speed in Linux (in sec)

Program	Original	Obfuscated by Popov		Obfuscated by binOb+Linux	
	exec. time (T0)	exec. time (T1)	slowdown (T1/T0)	exec. time (estimated) (T2)	slowdown (T2/T0)
vortex	235.65	240.65	1.02	241.41	1.02
bzip2	283.01	377.62	1.33	452.21	1.60
mcf	425.97	427.13	1.00	428.10	1.00
gzip	210.04	209.50	1.00	209.07	1.00
vpr	319.48	328.56	1.03	333.67	1.04
GEOM. MEAN			1.08		1.13

exec. time : execution time in seconds

Table 4 shows the effects of obfuscation on execution speed in Linux. The data for 'original' and 'obfuscated by Popov' is excerpted from the paper [19], since we do not have the same input for execution to compare. Using the excerpted data, the worst-case execution speed of a binary obfuscated with *binOb+Linux* is estimated based on the number of *trap codes* inserted in the obfuscated binary. The execution of a binary obfuscated by *binOb+Linux* slows down a little compared to the original binary, and the slowdown factor is similar to the previous work (the speed for gzip is suspected as an error due to the cache effects or experimental errors according to the paper [19]).

Table 5: Exception handling time in Windows

Exception Handling Time (usec)		
Signal in Windows (T0)	Windows SEH (T1)	Overhead (T0/T1)
8.59	6.41	1.34

We measure how long it takes to properly handle an exception raised during execution in Windows using 1) signal handling mechanism and 2) Windows SEH. This measurement is intended for showing the superiority of our exception handling mechanism over signal handling mechanism. For obtaining the accurate handling time, an exception is generated for several thousand iterations and handled by a special handler designated to handle the only exception every time. The handling time of one exception is calculated on average while considering the loop overheads [23]. As shown in Table 5, the exception handling time using Windows SEH is shorter than the signal handling time, because Windows SEH mechanism is ahead of signal handling mechanism.

Overall, the obfuscated binary executes more slowly than the original program. However, note that slowdown factor would not be an important matter for some applications such as 'network-centric applications' and 'CPU non-intensive applications'. In addition, one could apply obfuscation only to some important functions that should be protected in the program.

6. RELATED WORK

Many different approaches for obfuscation have been developed, and the earliest work is to overlap adjacent instructions to fool a disassembler [4]. Collberg et al.'s work [6,5] is a fundamental work of obfuscation, which is done by transforming control flow and data flow. Sharif et al. suggested how to impede malware analysis by using conditional code obfuscation [22]. The work is remarkable in hiding semantics of program, but the execution cost is too expensive and stealth of obfuscated binary is not guaranteed at all. Linn et al. [14] proposed two techniques: insertion of "junk bytes" and redirection of all function calls to "branch functions" to disrupt a disassembler. Based on Linn et al.'s work, Popov et al. presented an advanced work by replacing branch instructions with trap and bogus instructions [19].

There has also been significant improvement in the de-obfuscation techniques to invade obfuscation techniques. The most outstanding work is an exhaustive disassembler proposed by Kruegel et al. [12]. The disassembler generates all possible control-flow graph in obfuscated binaries and identifies genuine ones using heuristical and statistical technique, which is based on statistics of instruction pairs to identify abnormal instructions like *alignment-break code*. The disassembler analyzes obfuscated binaries using control-flow graph and statistical technique, which is based on statistics of instruction pairs to identify abnormal instructions like *align-break code*. Raber et al. proposed another de-obfuscation method, which finds specific patterns of obfuscation algorithms [20]. However, our work is resistant to the de-obfuscation techniques because we considered both statistical compensation and polymorphic variation.

In addition, to cope with obfuscation, malware detection techniques using data mining approach have been used, which are suitable for both known and unknown malicious codes. Kolter et al. [10] proposed a fundamental malware detection based on an applied machine learning and data mining technique. It extracts n-gram features from a training set of executables in byte-by-byte and tests various classification methods to detect malicious one. n-gram is a subsequence of n items from a given sequence. Masud et al. [15] extended the work by extracting n-gram features more complicatedly in instruction-by-instruction instead of in byte-by-byte. The most relevant one with our approach is Stolfo et al.'s work [24], called *Fileprint Analysis*, which uses 1-gram model and measures Mahalanobis distance to compare similarities between trained ones and test ones. Our obfuscation method embraces these techniques for tolerating them.

7. CONCLUSION AND FUTURE WORK

This paper proposed a novel binary obfuscation framework, called *binOb+*, which addresses the following three challenges: 1) to hide real instructions while covering the entire code sections; 2) to eliminate statistical abnormality of obfuscated binary; and 3) to make it work on not only Linux binaries but also Windows binaries, the most popular target for reverse engineering. The proposed framework consists of

two phases: it obfuscates most of code section of the original binary by breaking the disassembly alignment with *trap code* and *alignment-break codes*; it also smoothes the statistical difference of an obfuscated binary from non-obfuscated binaries by inserting *statistics-compensation code*. Our obfuscator is implemented in Windows XP using Windows SEH to handle exceptions properly.

The experimental results show that *binOb+* successfully disrupts disassembly processes, including the most powerful commercial disassembler *IDA* and a specialized disassembler *exhaustive*, better than the previous work [19]. Even more, the obfuscated binary is shown to be similar to the original binary in terms of the Mahalanobis distance. Windows SEH can be successfully applied to handle exceptions in obfuscated Windows binaries without losing the advantages of signals in Linux.

There are some remaining issues that need to be addressed, however. Firstly, our obfuscator can be fortified against advanced disassembly techniques by making fake control-flows between *alignment-broken areas*. If we insert a fake branch instruction, whose destination is somewhere in *alignment-broken areas*, into *statistics-smoothing codes*, then disassemblers can be disrupted more. Secondly, local n-gram property in binary can be applied for compensating the statistical difference better. At present, our obfuscator supports a global 1-gram property, but the local n-gram property can be guaranteed if we apply a Markov transition matrix method [21] and sliding-window technique to obfuscated binary. Finally, in order to consider operands as well as opcodes for stealthy, we will use *operand abstraction*, which models the possible types of operands such as registers, immediate values, and memory addresses. Based on their appearing frequences in normal binaries, the same technique as we did for opcodes would be applicable as well.

8. ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their helpful comments. This research was supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency (NIPA-2009-C1090-0902-0045) and supported by WCU (World Class University) program through the National Research Foundation of Korea funded by the Ministry of Education, Science and Technology (R31-2008-000-10100-0).

9. REFERENCES

[1] Data Rescue. IDA Pro. http://www.datarescue.com/idabase.

[2] Microsoft Corporation. DumpBin. http://support.microsoft.com/kb/177429.

[3] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th conference on USENIX Security Symposium*, pages 169–186. USENIX Association, 2003.

[4] F. B. Cohen. Operating system protection through program evolution, 1992. http://all.net/books/IP/evelve.html.

[5] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *IEEE International Conference on Computer Languages*, pages 28–38. IEEE Computer Society, 1998.

[6] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, 1998.

[7] E. Eilam. *Reversing: secrets of reverse engineering*. Wiley, 2005.

[8] M. Gagnon, S. Taylor, and A. Ghosh. Software Protection through Anti-Debugging. *IEEE Security and Privacy*, 5(3):82–84, 2007.

[9] GNU Project-Free Software Foundation. Objdump, GNU Manuals Online. http://sourceware.org/binutils/docs/binutils/objdump.html.

[10] J. Z. Kolter and M. A. Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the 10th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 470–478. ACM, 2004.

[11] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th conference on USENIX Security Symposium*, pages 161–176. USENIX Association, 2005.

[12] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*. USENIX Association, 2004.

[13] S. Kumar and E. Spafford. A generic virus scanner for C++. In *Proceedings of the 8th Annual Computer Security Applications Conference*, pages 210–219. IEEE Computer Society, 1992.

[14] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[15] M. Masud, L. Khan, and B. Thuraisingham. A hybrid model to detect malicious executables. In *Proceedings of IEEE International Conference on Communications*, pages 1443–1448. IEEE Communication Society, 2007.

[16] R. Moskovitch, C. Feher, N. Tzachar, E. Berger, M. Gitelman, S. Dolev, and Y. Elovici. Unknown Malcode Detection Using OPCODE Representation. In *Proceedings of the 1st European Conference on Intelligence and Security Informatics*, pages 204–215. Springer, 2008.

[17] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100. ACM, 2007.

[18] M. Pietrek. A Crash Course on the Depths of Win32 [R] Structured Exception Handling. *Microsoft Systems Journal-UK-*, 6:19–33, 1997.

[19] I. V. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *Proceedings of the 16th USENIX Security Symposium*, pages 1–16. USENIX Association, 2007.

[20] J. Raber and E. Laspe. Deobfuscator: An automated approach to the identification and removal of code obfuscation. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 275–276. IEEE Computer Society, 2007.

[21] M. Shafiq, S. Khayam, M. Farooq, P. Islamabad, and P. Rawalpindi. Embedded Malware Detection Using Markov n-Grams. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–107. Springer, 2008.

[22] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*. The Internet Society, 2008.

[23] C. Staelin and L. McVoy. mhz: anatomy of a micro-benchmark. In *Proceedings of the 1008 USENIX Annual Technical Conference*, pages 155–166. USENIX Association, 1998.

[24] S. Stolfo, K. Wang, and W. Li. Towards stealthy malware detection. *Malware Detection*, 27:231–249, 2007.

[25] E. Stroulia and T. Systä. Dynamic analysis for reverse engineering and program understanding. *ACM SIGAPP Applied Computing Review*, 10(1):8–17, 2002.

[26] A. Sung, J. Xu, P. Chavez, and S. Mukkamala. Static analyzer of vicious executables (save). In *Proceedings of the 20th Annual Computer Security Applications Conference*, pages 326–334. IEEE Computer Society, 2004.

[27] S. Theodoridis and K. Koutroumbas. *Pattern recognition*. Academic press, 2006.

[28] US-Cert. Advisory CA-2001-10 Code Red Worm. http://www.cert.org/advisories/CA-2001-19.html.

[29] US-Cert. Advisory CA-2003-04 MS-SQL Server Worm. http://www.cert.org/advisories/CA-2003-04.html.

[30] G. Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.

[31] O. Yuschuk. OllyDbg. http://www.ollydbg.de.

APPENDIX

A. SUPPLEMENTARY EXPERIMENTAL RESULTS

This appendix contains supplementary experimental results about disassembly mismatches and program size.

A.1 Disassembly Mismatch

Table 6 and Table 7 show disassembly mismatches in Linux and Windows, respectively. The confusion factor of 'gzip' obfuscated by Popov cannot be evaluated in IDA due to the error of 'IDA' when it loads the binary.

Table 6: Disassembly Mismatch in Linux (Confusion Factor, %)

	Program	Objdump			Exaustive			IDA		
		Instrs	Blocks	Funcs	Instrs	Blocks	Funcs	Instrs	Blocks	Funcs
Popov	vortex	38.93	75.50	93.08	63.25	72.10	93.74	58.71	72.58	96.75
	bzip2	44.33	69.61	88.34	56.19	60.75	87.45	55.62	58.43	94.78
	mcf	43.65	70.31	88.81	55.74	61.03	88.12	56.96	60.99	94.31
	gzip	44.23	69.63	89.10	55.05	59.58	87.31	-	-	-
	vpr	42.45	70.62	86.84	59.10	64.80	87.64	58.08	63.04	94.25
	GEOM. MEAN	42.72	71.13	89.24	57.86	63.65	88.85	57.34	63.76	95.02
binOb+Linux	vortex	61.53	91.33	98.07	74.27	88.53	98.56	94.75	98.26	98.73
	bzip2	60.26	86.00	95.56	68.97	81.33	96.83	90.67	93.14	96.95
	mcf	61.71	89.59	96.27	70.31	83.27	97.38	93.83	97.36	97.78
	gzip	60.36	85.38	94.62	70.84	82.26	96.28	91.08	93.27	96.01
	vpr	63.74	88.62	94.74	71.31	82.94	96.45	93.88	96.48	95.97
	GEOM. MEAN	61.52	88.18	95.85	71.14	83.67	97.10	92.84	95.70	97.09

Table 7: Disassembly Mismatch in Windows (Confusion Factor, %)

	Program	Dumpbin			IDA		
		Instrs	Blocks	Funcs	Instrs	Blocks	Funcs
binOb+Win	sol	67.51	87.47	97.10	80.78	88.51	97.83
	winmine	58.68	75.97	97.44	76.51	83.66	98.72
	notepad	64.24	88.50	95.06	86.46	92.22	96.30
	ipconfig	65.83	95.32	97.98	88.89	98.42	98.99
	netstat	66.13	93.77	95.65	87.84	98.75	97.83
	nslookup	68.45	95.19	97.87	93.22	98.94	98.94
	tftp	65.34	89.22	91.89	90.38	98.09	94.59
	GEOM. MEAN	65.17	89.35	96.14	86.30	94.08	97.60

A.2 Program Size

Table 8 shows the effects of obfuscation on program size in Linux.

Table 8: Effects of Obfuscation on Program Size in Linux

Program	code section			data section			combined (code section+ data section)		
	orig.	Popov	binOb+Linux	orig.	Popov	binOb+Linux	orig.	Popov	binOb+Linux
vortex	725,316	1,676,143	1,926,603	19,768	417,436	279,236	745,084	2,093,579	2,205,839
bzip2	343,624	706,648	849,459	6,420	153,396	113,168	350,044	860,044	962,627
mcf	301,636	631,771	745,496	3,284	135,212	98,252	304,920	766,983	843,748
gzip	344,920	713,534	848,104	5,812	153,496	112,012	350,732	867,030	960,116
vpr	407,084	853,914	1,021,633	3,416	195,924	138,672	410,500	1,049,838	1,160,305

obfus. : the size of obfuscated binary (in bytes)
orig. : the size of original binary (in bytes)