3 Program Transformations and Obfuscations

Software feature extraction must cope with transformations that are intended to obscure, evolve, or rewrite the program. For example, malware polymorphism and metamorphism are transformations applied to the malicious code to evade signature detection. Robust signatures must identify the invariant birthmarks under these transformations. This chapter focuses on analysing these types of program transformations and obfuscations including compiler optimsations, recompilation, plagiarism, software theft, derivative works, malware packing, malware polymorphism and malware metamorphism.

Keywords: Program obfuscation, compiler optimisation, code packing, polymorphism, metamorphism.

3.1 Compiler Optimisation and Recompilation

Compiler optimisations and recompilation are semantic preserving transformations. These transformations rewrite the program but do not alter the behavioural properties of the software. Compiler optimisations make feature extraction more difficult. Even very minor changes to a program's source code can result in significant changes to the program's instruction stream once recompiled.

Many compiler optimisations are possible. We examine some in this section. Typical classes of code optimisation that may affect the birthmarks and feature extraction are:

- Instruction Reordering
- Loop Invariant Code Motion
- Code Fusion
- Function Inlining
- Loop Unrolling
- Branch/Loop Inversion
- Strength Reduction
- Algebraic Identities
- Register Assignment

3.1.1 Instruction Reordering

Instructions can be reordered or scheduled in such a way that they are semantically equivalent but perform faster due to caching. To determine if instructions inside a basic block can be reordered, a directed acyclic graph can be drawn of the data dependencies. Only instructions that have data dependencies between each other require strict ordering between those instructions.

3.1.2 Loop Invariant Code Motion

Code that is inside a loop may be moved to outside the loop if no semantic change occurs. This improves the efficiency of the code.

3.1.3 Code Fusion

Code inside loops in sequence can be fused into a single loop.

3.1.4 Function Inlining

Functions can be inlined to improve performance. Inlining a function means that a clone or copy of that function replaces the function call. This means that a function call is avoided and therefore improves performance.

3.1.5 Loop Unrolling

It can improve efficiency to unroll the loop by duplicating the loop body and termination condition.

3.1.6 Branch/Loop Inversion

Branching on equality or non equality can be inverted and may improve efficiency in some cases.

3.1.7 Strength Reduction

Strength reduction replaces expensive operations with equivalent but less expensive operations.

3.1.8 Algebraic Identities

Algebraic identities take note that some expressions are algebraically equivalent to other less expensive operations. For example, x+0 is equivalent to the less expensive expression x.

3.1.9 Register Reassignment

Register allocation is the process of assigning specific registers to instructions. The assignment of these registers can change while maintaining semantically equivalent code.

3.2 Program Obfuscation

Program obfuscation obscures the workings of a program [1].

Definition 3.1. *Let* $P \xrightarrow{T} P'$ *be a transformation of a source program P into a target program P'.* $P \xrightarrow{T} P'$ *is an obfuscating transformation, if P and P' have the same observable behaviour. More precisely, in order for* $P \xrightarrow{T} P'$ *to be a legal obfuscating transformation the following conditions must hold:*

- *If P fails to terminate or terminates with an error condition, then P' may or may not terminate.*
- *Otherwise, P' must terminate and produce the same output as P.*

3.3 Plagiarism, Software Theft, and Derivative Works

An incomplete list of source code plagiarism techniques is described in [2]. The authors state that such a list is never ending, so a comprehensive list is impossible. Nevertheless, they identified the following forms of plagiarism:

- Lexical Changes
  - Comments can be reworded, added and omitted
  - Formatting can be changed.
  - Identifier names can be modified.
  - Line numbers can be changed (e.g., in Fortran programs).
- Structural Changes
  - Loops can be replaced (e.g, replacing a while loop with a for loop)
  - Nested if statements can be replaced by case statements and vice versa.
  - Statement order can be changed.
  - Procedures can be replaced by functions (e.g., in Pascal)
  - Procedures may be inlined
  - Ordering of operands may be changed (e.g., x < y becomes x >= y)

3.3.1 Semantic Changes

An extension to syntactic changes is that of semantic changes where the new variant is a derived work of the original malware. Semantic changes occur due to the software authors modifying the original source code or functionality. This can occur to a natural evolution of the software during its development life cycle. Additionally, it can occur when a software author reuses existing code in a new program instance.

3.3.2 Code Insertion

Code insertion occurs when new functionality is added to the malware.

3.3.3 Code Deletion

Code deletion occurs when functionality is removed from the malware.

3.3.4 Code Substitution

Code substitution occurs when functionality in the malware is replaced by an alternative algorithm or code.

3.3.4 Code Transposition

Code transposition occurs when specific code and functionality of the malware is removed from its initial location and inserted into a semantically different location in the malware.

3.4 Malware Packing, Polymorphism, and Metamorphism

The two categories of malware obfuscation are syntactic and semantic changes. Semantic changes include those described for plagiarism and software theft. A syntactic polymorphic malware technique is a method that changes the syntactic structure of the malware [3]. Though the syntactic structure changes in polymorphic malware, the malware semantically remains identical. The technique is predominantly used to evade byte level signature based detection and classification that is routinely employed by traditional Antivirus. Polymorphism borrows many of the techniques from the field of program obfuscation.

Polymorphism is sometimes described by the similar term of metamorphism. In that usage it is used to describe the automated syntactic mutation of the malware's code and instructions. Under such terminology, polymorphism is used to describe

syntactic mutation of limited parts of the malware's instruction content. The remaining parts of the malware are encoded at the byte level without regard to the instruction syntax or semantics. In this book we treat polymorphism and metamorphism as identical to each other.

Syntactic malware obfuscations and transformations include:

- Dead Code Insertion
- Instruction Substitution
- Variable Renaming
- Code Reordering
- Branch Inversion and Flipping
- Opaque Predicate Insertion
- Code Packing

3.4.1 Dead Code Insertion

Dead code is also known as junk code and a semantic nop [3]. Dead code is semantically equivalent to a nil operation. Insertion of this type of code has no semantic impact on the malware. The insertion increases the size of the malware and modifies the byte and instruction level content of the malware.

```
push %ebx
pop %ebx
```

Fig. 3.1 A semantic nop

3.4.2 Instruction Substitution

Instruction substitution replaces specific instructions or sequences of instructions with semantically equivalent, but differing instructions and instruction sequences. The size of the malware may grow or shrink in this procedure.

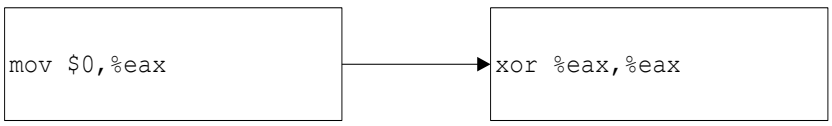

Fig. 3.2 Instruction substituion.

3.4.3 Variable Renaming

Variable renaming [4] and the associated technique of register reassignment alters the use of variables and registers in a sequence of code such that the instructions are semantically equivalent but use different variables and registers when compared to the original code.

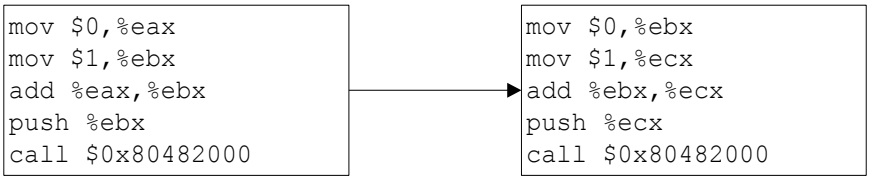

Fig. 3.3 Register reassignment.

3.4.4 Code Reordering

Code reordering [4] changes the syntactic order of the code in the malware [3]. The actual or semantic execution path of the program does not change. However, the syntactic order as present in the malware image is altered. Code reordering includes the techniques of branch obfuscation, branch inversion, branch flipping, and the use of opaque predicates.

3.4.5 Branch Obfuscation

Branch obfuscation attempts to hide the target of a branch instruction. Examples include the use of Structured Exception Handling (SEH) on the Microsoft Windows platform. The use of SEH to obscure control flow is common in modern malware. Similar techniques involve indirect branching. Indirect branching uses data content as the target of a branch. This translates control flow identification into a harder data flow analysis problem. The use of a branch function [5] extends this approach and dispatches multiple branches through a single routine. The main purpose of branch obfuscation is to make the static analysis of the malware by an analyst or automated system harder to perform.

```
mov $0x8048200,%eax
jmp *%eax
```

Fig. 3.4 An indirect branch.

3.4.6 Branch Inversion and Flipping

Branch inversion inverts the branch condition in conditional branches. Whereas the branch may originally transfer control when the condition is true, branch inversion alters the condition to branch when false. To maintain the original semantics of the program the branch instruction is also inverted. For example, a branch on condition true statement can be changed to a branch on condition false statement. Additionally, the condition being tested would also be inverted. Branch inversion is effectively a form of instruction substitution on control flow statements.

Branch flipping [5] is a similar technique to branch inversion and rewrites the branch instruction by substituting it with semantically equivalent code with different control flow properties. For example, if the original code is to branch on condition true then the new code branches on condition false to the original fall-through instruction. The new fall-through instruction then unconditionally branches to the original conditional branch target.

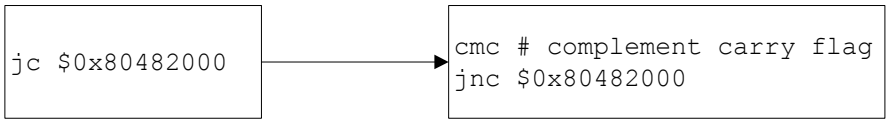

Fig. 3.5 Branch inversion.

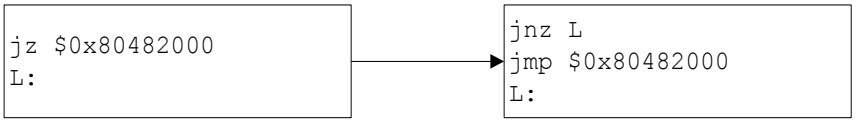

Fig. 3.6 Branch flipping.

3.4.7 Opaque Predicate Insertion

An opaque predicate [5] is a predicate that always evaluates to the same result. An opaque predicate is constructed so that it is difficult for an analyst or automated analysis to know the predicate result. Opaque predicates can be used to insert superfluous branching in the malware's control flow. They can also be used to assign variables values which are hard to determine statically. The use of opaque predicates is primarily for code obfuscation, and to prevent understanding by an analyst or automated static analysis.

3.4.8 Malware Obfuscation Using Code Packing

Code packing [6-7] is the dominant technique used to obfuscate malware and hinder an analyst's understanding of the malware's intent. In one month during 2007, 79% of identified malware from a commercial Antivirus vendor was found to be packed [8]. Additionally, almost 50% of new malware in 2006 were repacked versions of existing malware [9].

Code packing, in addition to obfuscating the understanding of the malware by an analyst, is also used by malware to evade an Antivirus system's detection. Polypack [10] evaluated the effectiveness of code packing against Antivirus detection by providing a service to pack malware using a variety of code packing tools. Antivirus systems often have the capabilities of unpacking known code packing tools, and unpacking unknown tools has also had commercial interest [11]. However, Polypack demonstrated that packing can be an effective tool to defeat an Antivirus system with many commercial malware detection systems failing to identify the packed versions of existing malware.

Code packing is used in the majority of malware, but code packing also serves to provide compression and software protection for the intellectual property contained in a program. It is not necessarily advantageous to flag all occurrences of code packing as being indicative of malicious activity. Code packing tools are freely available [12] and commercially sold to the public as legitimate software [13]. For this reason, unpacking of packed programs provides benefit. It is advisable to determine if the packed contents are malicious, rather than identifying only the fact that unknown contents are packed.

3.4.9 Traditional Code Packing

The most common method of code packing is described in [6]. Malware employing this method of code packing transforms executable code into data as a post-processing stage in the malware development cycle. This transformation may perform compression or encryption, hindering an analyst's understanding of the malware when using static analysis. At runtime, the data, or hidden code, is restored to its original executable form through dynamic code generation using an associated restoration routine [14]. Execution then resumes as normal to the original entry point. The original entry point marks the entry point of the original malware, before the code packing transformation is applied. Execution of the malware, once the restoration routine is complete and control is transferred to the original entry point, is transparent to the fact that code packing and restoration had been performed. A malware may have the code packing transformation applied more than once. After the restoration routine of one packing transformation has been applied, control may transfer another packed layer. The original entry point is derived from the last such layer. The process of this form of malware packing is shown in Fig. 3.7.

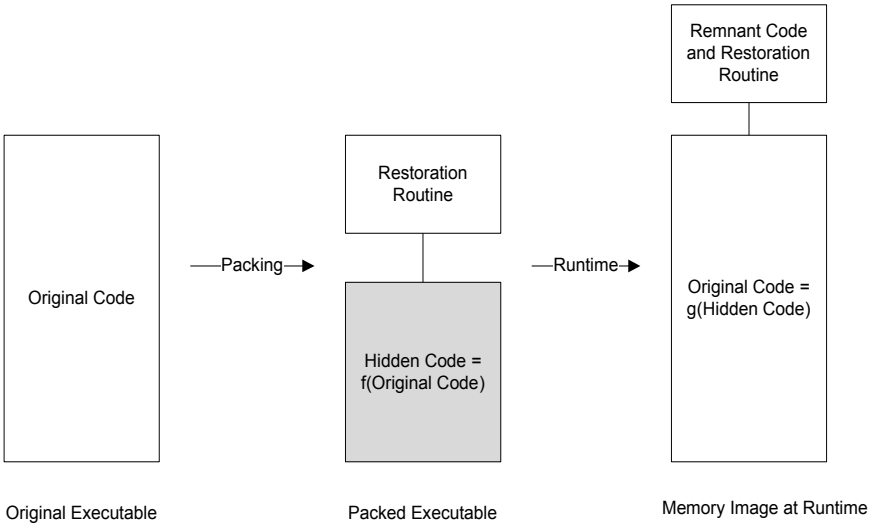

Fig. 3.7 The traditional code packing transformation.

3.4.10 Shifting Decode Frame

An extension to traditional code packing is to maintain as much of the packed image in an encrypted form at run-time. During execution of the malware, blocks of memory can be decrypted as needed and subsequently re-encrypted to prevent an analyst or automated system from having access to all the hidden code at any single moment in time. This technique is known as the shifting decode frame [15]. The granularity of encryption can occur at the page level, the basic block level, and the instruction level. This type of code packing is not often used in wild malware, and in practice, traditional code packing and instruction virtualization are the dominant techniques used in real malware. The process of this form of malware packing is shown in Fig. 3.8.

3.4.11 Instruction Virtualization and Malware Emulators

Code packing may employ the use of instruction virtualization also known as a malware emulator [7]. An emulator used by a malware should not be confused with an emulator used for automated unpacking of the malware. This type of code packing transformation employing an emulator is used in a minority of malware. In this form of code packing, packing translates the original native code into a byte-code which is subsequently emulated by the malware at run-time. Using this form of code packing, the hidden code in its original form is never revealed. The process of this form of malware packing is shown in Fig. 3.9.

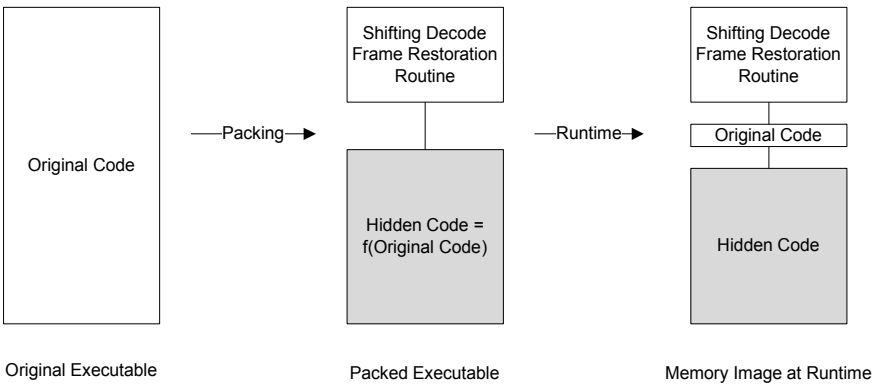

Fig. 3.8 Code packing using the shifting decode frame.

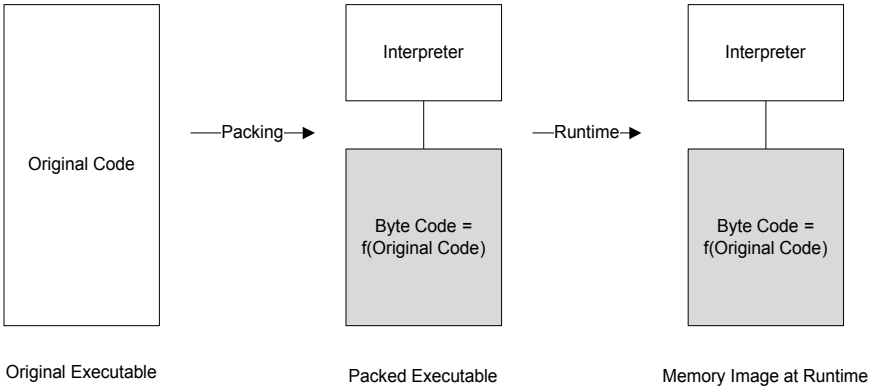

Fig. 3.9 Code packing using instruction virtualization.

3.5 Features under Program Transformations

Program features may change under program transformations and obfuscation. The challenge then is in choosing features which remain invariant under these conditions. The raw or byte level content deals poorly with program transformations. Small changes in high level source code may result in large changes in the raw content. Instruction level content is also prone to large changes under transformations such as when registers are reassigned or the instruction stream is modified. Control flow is more invariant than most syntactic features and can be a good choice. At a source code level, program and system dependency graphs have been popular. The APIs used by a program represent a good choice and have been widely used in behavioural analysis of malware. For static analysis of malware, the malware must be unpacked to reveal its hidden code. Unpacking of malware is not addressed in this book.

References

1. Collberg C, Thomborson C, Low D (1997) A taxonomy of obfuscating transformations. Department of Computer Science, The University of Auckland, New Zealand,
2. Joy M, Luck M (1999) Plagiarism in programming assignments. Education, IEEE Transactions on 42 (2):129-133
3. Christodorescu M, Kinder J, Jha S, Katzenbeisser S, Veith H (2005) Malware normalization. University of Wisconsin, Madison, Wisconsin, USA,

4. Mihai C, Somesh J (2004) Testing malware detectors. Paper presented at the Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, Boston, Massachusetts, USA,
5. Cullen L, Saumya D (2003) Obfuscation of executable code to improve resistance to static disassembly. Paper presented at the Proceedings of the 10th ACM conference on Computer and communications security, Washington D.C., USA,
6. Royal P, Halpin M, Dagon D, Edmonds R, Lee W Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In: Computer Security Applications Conference, 2006. pp 289-300
7. Sharif M, Lanzi A, Giffin J, Lee W (2009) Rotalume: A Tool for Automatic Reverse Engineering of Malware Emulators.
8. Panda Research (2007) Mal(ware)formation statistics - Panda Research Blog.
9. Stepan A Improving proactive detection of packed malware. In: Virus Bulletin Conference, 2006.
10. Oberheide J, Bailey M, Jahanian F Polypack. In: USENIX Workshop on Offensive Technologies (WOOT '09), Montreal, Canada, 2009.
11. Graf T (2005) Generic unpacking: How to handle modified or unknown PE compression engines. Paper presented at the Virus Bulletin Conference,
12. UPX: the Ultimate Packer for eXecutables. (2010). http://upx.sourceforge.net/. Accessed 6 April 2010 2010
13. Themida. (2010). http://www.themida.com/. Accessed 6 April 2010 2010
14. Kang MG, Poosankam P, Yin H Renovo: A hidden code extractor for packed executables. In: Workshop on Recurring Malcode, 2007. pp 46-53
15. Boehne L (2008) Pandora's Bochs: Automatic Unpacking of Malware. University of Mannheim,