Covert Computation
Hiding Code in Code
for Obfuscation Purposes
Malware Obfuscation

34.79 % not packed
65.21 % packed

Source: Rodrigo Rubira Branco (2012)
Packers hide the actual code from static analysis!
symbols
library
dependencies
data
code

symbols
library
dependencies
compressed data
unpacker
Mimimorphism: A New Approach to Binary Code Obfuscation

Zhenyu Wu, Steven Gianvecchio, Mengjun Xie; and Haining Wang
The College of William and Mary
Williamsburg, VA 23187, USA
{adamwu, srgian, mjjxie, hnw}@cs.wm.edu

ABSTRACT

Binary obfuscation plays an essential role in evading malware static analysis and detection. The widely used code obfuscation techniques, such as polymorphism and metamorphism, focus on evading syntax-based detection. However, statistical test and semantic analysis techniques have been developed to thwart their evasion attempts. More recent binary obfuscation techniques are divided in their purposes of attacking either statistical or semantic approach, but not both. In this paper, we introduce mimimorphism, a novel binary obfuscation technique with the potential of evading both statistical and semantic detections. Mimimorphic malware uses instruction-syntax-aware high-order mimic functions to transform its binary into mimicry executables that exhibit high similarity to benign programs in terms of statistical properties and semantic characteristics. We implement a prototype of the mimimorphic engine on the Intel x86 platform, and evaluate its capability of evading statistical anomaly detection and semantic analysis detection techniques. Our experimental results demonstrate that the mimicry executables are indistinguishable from benign programs in terms of byte frequency distribution and entropy, as well as control flow fingerprint.

1. INTRODUCTION

Real-time malicious binary detection is the first line of defense against malicious software. With the prevalence of anti-malware software nowadays, in order to be executed on a host computer, a piece of binary code is subjected to a number of detection scans during transportation and before execution. Consequently, evading real-time binary detection is critical for malware to succeed in propagation.

To date, real-time malware detection largely relies on static binary analysis, due to its significant speed and resource consumption advantages over dynamic executable analysis [1, 2, 3, 4, 5]. Malware mainly evades static analysis detections through binary obfuscations, namely oligomorphism, polymorphism, and metamorphism [6]. Oligomorphism is used to evade byte sequence signature detections on the malware functional code. It utilizes simple operations such as XOR to scramble malware functional code before propagation, and decodes it while executing. Evolved from oligomorphism, polymorphism encodes malicious code by “packing” (i.e., compressing or encrypting), and then camouflages the “unpacker” (the decompressing or decrypting code) by using binary mutation techniques, such as instruction substitution and register remapping. Instead of packing program binaries, metamorphism generates different instruction combinations to represent the same functional part of a malicious program in its variants. The major techniques employed by metamorphism are binary-level mutations and meta-level transformations. A meta-level transformation first translates binary code into a temporary representation called R-code, and finally compiles a new intermediate code.
Hiding code in data
Hiding code in code
Malicious code is hidden inside other code - useless - but - executable.
Using side effects of the microprocessor to implement hidden functionality!
ADD EAX, EBX

Sum of EAX and EBX is stored in EAX

FLAGS register is modified (Overflow, Sign, Zero, Parity, Carry, Adjust)
MOV ECX, 200
XCHG EAX, ECX
LOOP -1

Equivalent instruction:
SUB EAX, 200

LOOP decrements ECX by one!
*) What about semantic-aware malware detection?**

**) M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware malware detection. Oakland ‘05
Semantic-aware malware detection

- Static analysis with advanced patterns
- Templates for malicious behavior
- Decision based on result instead of actual implementation
- Model of the microprocessor
Limitations

- Evaluation of maliciousness based on model of machine
- Models are an abstract representation of the real world
- Not strong enough to map the entire functionality of hardware
- Moreover: Mapping functionality is difficult!
Implementation

Source Code:
would get removed by optimizer at compile time

Binary:
binary rewriting is error-prone
Implementation

Compile-time obfuscation!
Compile-Time Obfuscation

• Implemented by intercepting the compilation process of LLVM

• Compiler wrapper
Performance Evaluation

![Bar chart showing performance evaluation results for different file types and encryption algorithms. The chart compares the time taken in seconds for Image, Audio, Text, and Movie files encrypted with `aescrypt` and `aescrypt (obfuscated)`.]
Performance Evaluation
Conclusion

• Today’s microprocessor architectures are highly complex

• Side effects are difficult to map with machine models - *knowledge gap*

• Hiding code in code

• Moderate effects on binary size, increased complexity (relevance?)