A Software Protection Method Based on Instruction Camouflage

2nd COE Postdoctoral and Doctoral Researchers Technical Presentation
May 26, 2005

Yuichiro Kanazaki
Software Engineering Laboratory, Graduate School of Information Science, Nara Institute of Science and Technology

We need a method for protecting software to create a safe ubiquitous computing environment.

Program Obfuscation
difficult to automate

Assembly Program

Narrow the Range of Analysis

Making expressions and procedures in a program more complex than the original[1].

Making a program harder to understand with encryption[2].

Approach - Self-modification mechanism

We add a self-modification mechanism to a program, to increase the cost of understanding a program.

self-modification: An instruction in the program replaces another instruction in the same program at run-time

Background
Software cracking has posed a serious problem for copyright protection of the software.

Example
An attacker analyzes a digital contents distribution system and obtains a secret key[1].

An attacker analyzes a program embedded in a set-top box and steals a device key[2].

Attacker: an individual who illegally analyzes software, and uses the outcome for other purposes.

We need a method for protecting software to create a safe ubiquitous computing environment.

Methods to increase cost for understanding a program

Program Obfuscation
Making expressions and procedures in a program more complex than the original[1].

Program Encryption
Making a program harder to understand with encryption[2].

Previous methods are still impractical:
- difficult to automate
- easy to nullify

We propose a practical method for protecting software and develop a system.

Camouflaging an instruction

Overwrite an original instruction with a dummy, which makes attackers misread the program.

1. We overwrite a target instruction with a dummy instruction.
2. We add self-modification routines that replace the dummy instruction with the original one within a certain period of execution.
Extending a range of analysis
Camouflaged instructions force attackers into extending the range of analysis.

A part that an attacker tries to understand

To read the long program costs an attacker a lot of time.

A program to be protected

Multiple camouflaging
We camouflaged many of the original instructions by dummy instructions and add routines.

Many instructions are camouflaged by dummy instructions.

No matter where an attacker tries to analyze, he will encounter some camouflaged instruction.

Outline of our system

1. Determine a target instruction and the positions of self-modification routines
2. Determine a dummy instruction
3. Generate self-modification routines
4. Embed the dummy and the routines in the program

RINRUN
We have implemented a system that automates the construction of camouflaged programs[1].

RINRUN outputs camouflaged Windows Executable from C source (GCC)

Discussion – Overview
We discuss the effectiveness of the proposed method.

Discussion Items (based on [1]):
- obscurity cost for understanding a part of program
- resilience difficulty of constructing an automatic tool to undo
- stealth difficulty of finding protection mechanism embedded
- overhead the execution time/space penalty incurred

Example Program
A simple data decryption program using C2
C2(Cryptomeria Cipher):
- A Feistel network-based block cipher designed for use in the area of digital entertainment content protection
- Algorithm is open to the public[2]
- Secret Key is security-sensitive

C source code (open)

Disassembled code (original)

Disassembled code (camouflaged)

Discussion – Obscurity(1/2)

kmpr = (WORD32) key[1], (WORD32) key[2]
**C source code**
```
secretconst = 0x20, 0x35, 0x4f
```

**Disassembled code (original)**
```
mov $20, -24(%ebp)
mov $35, -23(%ebp)
mov $4f, -22(%ebp)
```

**Disassembled code (camouflaged)**
```
mov $20, -24(%ebp)
mov $35, -23(%ebp)
mov $4f, -22(%ebp)
```

Easy to obtain the secret key
Difficult to obtain the secret key

---

**Discussion - Stealth**

Is it difficult for attackers to find the protection mechanism embedded?

The self-modification routines mainly consist of common instructions (e.g., `mov`, `add`, ...)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>50%</td>
</tr>
<tr>
<td>shift</td>
<td>9%</td>
</tr>
<tr>
<td>add</td>
<td>8%</td>
</tr>
<tr>
<td>lea</td>
<td>6%</td>
</tr>
<tr>
<td>or</td>
<td>6%</td>
</tr>
<tr>
<td>prel/pref</td>
<td>4%</td>
</tr>
<tr>
<td>other</td>
<td>12%</td>
</tr>
</tbody>
</table>

It is not easy to notice that the mechanism is embedded.

---

**Discussion – Performance Overhead**

**Target program**: `crypt` (well-known GNU utility for encrypting files)

When 500 instructions are camouflaged, the average execution time is about 2.9 seconds, which is about 47 times as long as the original (0.06 seconds).

---

**Conclusion and future plan**

**Conclusion**
- We presented a systematic method for protecting software against code analysis, by camouflaging instructions.
- We discuss the effectiveness of the proposed method.
  - Costs for understanding the program seems to be drastically increased.
  - The more we camouflaged the instructions, the more expensive program overhead becomes.

**Future Plan**
- Improving our system in consideration of architectural aspects to reduce performance overhead.