Protecting Software by Instruction Camouflage

4th COE Postdoctoral and Doctoral Researchers
Technical Presentation
July 22nd , 2004

Yuichiro Kanzaki

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

Table of Contents

- Background
- Key idea of our software protection method
- Procedure for protecting a program
- Case study
- Conclusion and future plan

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].

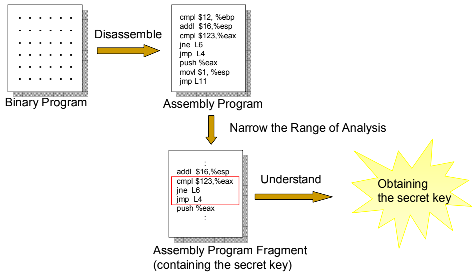
Attacker : an individual who illegally analyzes software, and uses the outcome for other purposes.

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

[1] S. Chow, P. Eisen, H. Johnson and P.C. van Oorschot: A white-box DES implementation for DRM applications, Proc. 2nd ACM Workshop on Digital Rights Management, pp.1-15, Nov. 2002.
[2] The United Kingdom Parliament, ``The mobile telephones (re-programming) bill,'' House of Commons Library Research Paper no.02/47, July 2002.

How is software attacked ?

A scenario of obtaining the secret key in a program



```
cmpl $12, %ebp
addl  $16,%esp
cmpl $123,%eax
jne  L6
jmp  L4
push %eax
movl $1, %esp
jmp L11
```

```
        :
addl  $16,%esp
cmpl $123,%eax
jne  L6
jmp  L4
push %eax
        :
```

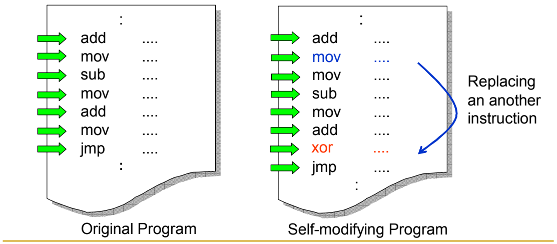
Disassemble
Binary Program
Assembly Program
Narrow the Range of Analysis
Understand
Obtaining the secret key
Assembly Program Fragment
(containing the secret key)

An effective solution to protect software against illegal code analysis is to increase costs for understanding the program.

Self-modification mechanism

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

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

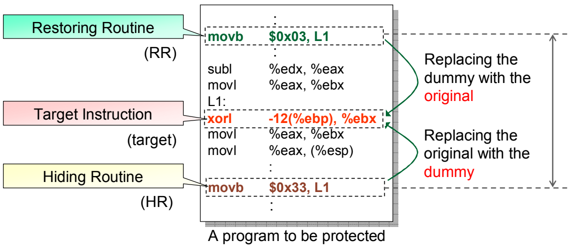


```
     :
add    ....
mov    ....
sub    ....
mov    ....
add    ....
mov    ....
jmp    ....
     :
```

```
     :
add    ....
mov    ....
mov    ....
sub    ....
mov    ....
add    ....
xor    ....
jmp    ....
     :
```

Camouflaging an instruction

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



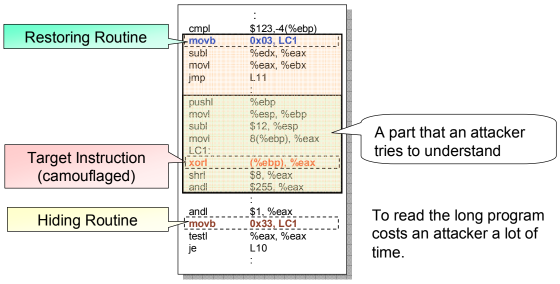
```
          :
movb      $0x03, L1
          :
subl      %edx, %eax
movl      %eax, %ebx
L1:
xorl      -12(%ebp), %ebx
movl      %eax, %ebx
movl      %eax, (%esp)
          :
movb      $0x33, L1
          :
```

Restoring Routine
(RR)
Target Instruction
(target)
Hiding Routine
(HR)
Replacing the dummy with the original
Replacing the original with the dummy
A program to be protected

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.


Restoring Routine
Target Instruction (camouflaged)
Hiding Routine
A part that an attacker tries to understand

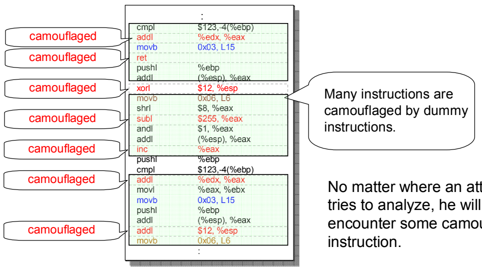
```
                :
cmpl      $123,-4(%ebp)
movb      0x03, LC1
subl      %edx, %eax
movl      %eax, %ebx
jmp       L11
          :
pushl     %ebp
movl      %esp, %ebp
subl      $12, %esp
movl      8(%ebp), %eax
LC1:
xorl      (%ebp), %eax
shrl      $8, %eax
andl      $255, %eax
          :
andl      $1, %eax
movb      0x33, LC1
testl     %eax, %eax
je        L10
          :
```

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

A program to be protected

Multiple camouflaging

We camouflage many of the original instructions by dummy instructions and add routines.



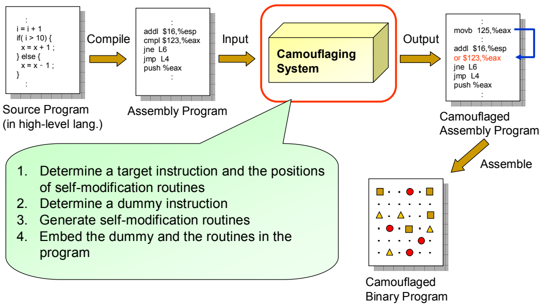
```
                :
cmpl      $123,-4(%ebp)
addl      %edx, %eax
movb      0x03, L15
ret
pushl     %ebp
addl      (%esp), %eax
xorl      $12, %esp
movb      0x06, L6
shrl      $8, %eax
subl      $255, %eax
andl      $1, %eax
addl      (%esp), %eax
inc       %eax
pushl     %ebp
cmpl      $123,-4(%ebp)
addl      %edx, %eax
movl      %eax, %ebx
movb      0x03, L15
pushl     %ebp
addl      (%esp), %eax
addl      $12, %esp
movb      0x06, L6
          :
```

Many instructions are camouflaged by dummy instructions.

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

A program to be protected

Outline of our system


Compile
Input
Camouflaging System
Output
Assemble
Source Program (in high-level lang.)
Assembly Program
Camouflaged Assembly Program
Camouflaged Binary Program

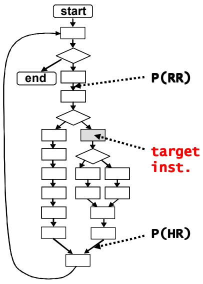
```
  :
i = i + 1
if( i > 10) {
  x = x + 1 ;
} else {
  x = x - 1 ;
}
  :
```

```
     :
addl  $16,%esp
cmpl $123,%eax
jne  L6
jmp  L4
push %eax
     :
```

```
     :
movb  125,%eax
     :
addl  $16,%esp
or $123,%eax
jne  L6
jmp  L4
push %eax
     :
```

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

(Step 1) Determine a target instruction and the positions of self-modification routines



Example of a control flow graph

RR : a restoring routine
HR : a hiding routine
P(RR) : position of inserting RR
P(HR) : position of inserting HR

Sufficient Conditions for correct execution:

1. ***P(RR)*** *must exist on every control flow path from the program entry to the* ***target****.*
2. ***P(HR)*** *must not exist on every control flow path from* ***P(RR)*** *to* ***target****.*
3. ***P(RR)*** *must exist on every control flow path from* ***P(HR)*** *to* ***target****.*
4. ***P(HR)*** *must exist on every control flow path from* ***target*** *to the program exit.*

These are sufficient conditions for a program not to cause malfunction .

(Step 2) Determine a dummy instruction

Example:

target instruction (original)

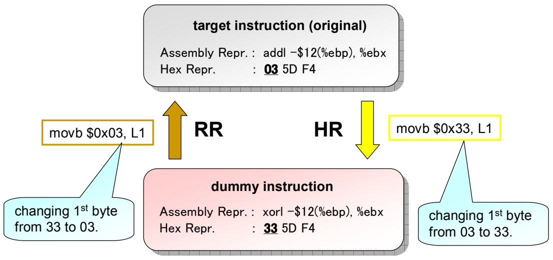
Assembly Repr. : addl −$12(%ebp), %ebx
Hex. Repr. : **03** 5D F4

A dummy instruction is obtained by changing the content of the target instruction.

dummy instruction

Assembly Repr. : xorl −$12(%ebp), %ebx
Hex. Repr. : **33** 5D F4

(Step 3) Generate self-modification routines



target instruction (original)

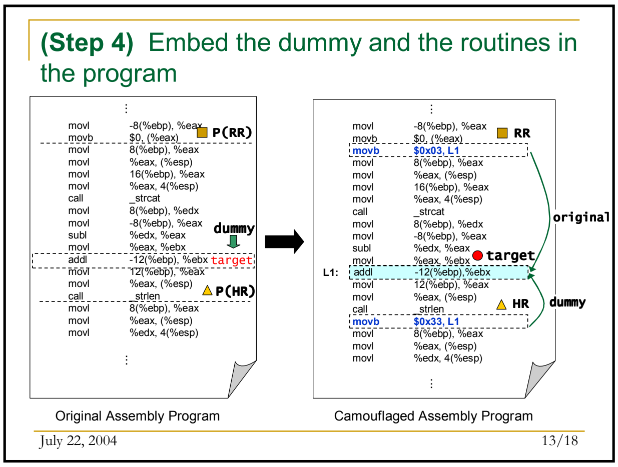
Assembly Repr. : addl −$12(%ebp), %ebx
Hex Repr. : **03** 5D F4

dummy instruction

Assembly Repr. : xorl −$12(%ebp), %ebx
Hex Repr. : **33** 5D F4

changing 1st byte from 33 to 03.

changing 1st byte from 03 to 33.



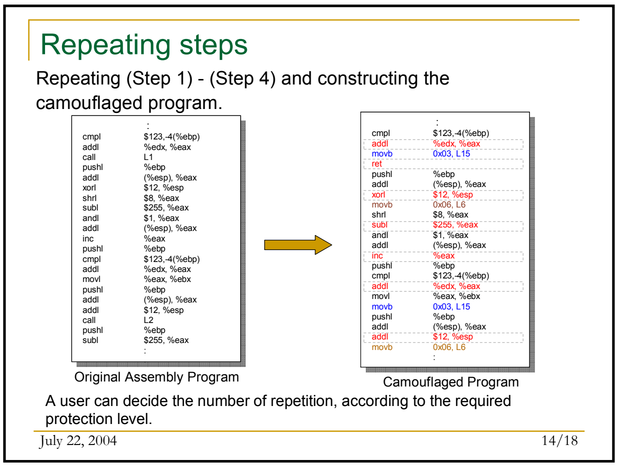
(Step 4) Embed the dummy and the routines in the program

```
                    :
     movl      -8(%ebp), %eax
     movb      $0, (%eax)                P(RR)
---------------------------------------------
     movl      8(%ebp), %eax
     movl      %eax, (%esp)
     movl      16(%ebp), %eax
     movl      %eax, 4(%esp)
     call      _strcat
     movl      8(%ebp), %edx
     movl      -8(%ebp), %eax          dummy
     subl      %edx, %eax
     movl      %eax, %ebx
     addl      -12(%ebp), %ebx  target
     movl      12(%ebp), %eax
     movl      %eax, (%esp)             P(HR)
     call      _strlen
---------------------------------------------
     movl      8(%ebp), %eax
     movl      %eax, (%esp)
     movl      %edx, 4(%esp)
                    :
```

Original Assembly Program

```
                    :
     movl      -8(%ebp), %eax
     movb      $0, (%eax)              RR
     movb      $0x03, L1
     movl      8(%ebp), %eax
     movl      %eax, (%esp)
     movl      16(%ebp), %eax
     movl      %eax, 4(%esp)                 original
     call      _strcat
     movl      8(%ebp), %edx
     movl      -8(%ebp), %eax
     subl      %edx, %eax
     movl      %eax, %ebx        target
L1:  addl      -12(%ebp),%ebx
     movl      12(%ebp), %eax
     movl      %eax, (%esp)            HR    dummy
     call      _strlen
     movb      $0x33, L1
     movl      8(%ebp), %eax
     movl      %eax, (%esp)
     movl      %edx, 4(%esp)
                    :
```

Camouflaged Assembly Program



Repeating steps

Repeating (Step 1) - (Step 4) and constructing the camouflaged program.

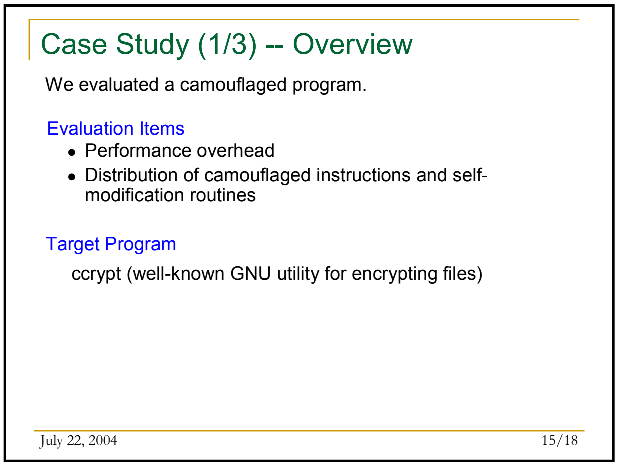
```
                 :
cmpl       $123,-4(%ebp)
addl       %edx, %eax
call       L1
pushl      %ebp
addl       (%esp), %eax
xorl       $12, %esp
shrl       $8, %eax
subl       $255, %eax
andl       $1, %eax
addl       (%esp), %eax
inc        %eax
pushl      %ebp
cmpl       $123,-4(%ebp)
addl       %edx, %eax
movl       %eax, %ebx
pushl      %ebp
addl       (%esp), %eax
addl       $12, %esp
call       L2
pushl      %ebp
subl       $255, %eax
                 :
```

Original Assembly Program

```
                 :
cmpl       $123,-4(%ebp)
addl       %edx, %eax
movb       0x03, L15
ret
pushl      %ebp
addl       (%esp), %eax
xorl       $12, %esp
movb       0x06, L6
shrl       $8, %eax
subl       $255, %eax
andl       $1, %eax
addl       (%esp), %eax
inc        %eax
pushl      %ebp
cmpl       $123,-4(%ebp)
addl       %edx, %eax
movl       %eax, %ebx
movb       0x03, L15
pushl      %ebp
addl       (%esp), %eax
addl       $12, %esp
movb       0x06, L6
                 :
```

Camouflaged Program

A user can decide the number of repetition, according to the required protection level.



Case Study (1/3) -- Overview

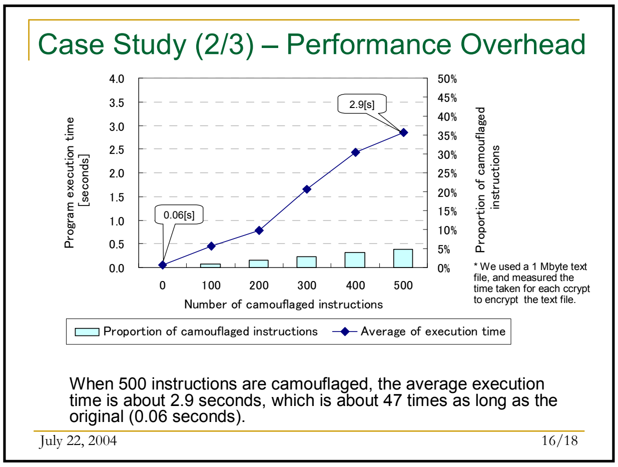
We evaluated a camouflaged program.

Evaluation Items

- Performance overhead
- Distribution of camouflaged instructions and self-modification routines

Target Program

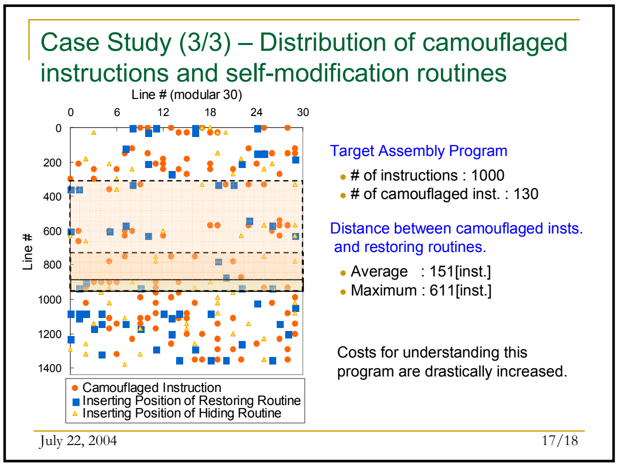
ccrypt (well-known GNU utility for encrypting files)



Case Study (2/3) – Performance Overhead

* We used a 1 Mbyte text file, and measured the time taken for each ccrypt to encrypt the text file.

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).



Case Study (3/3) – Distribution of camouflaged instructions and self-modification routines

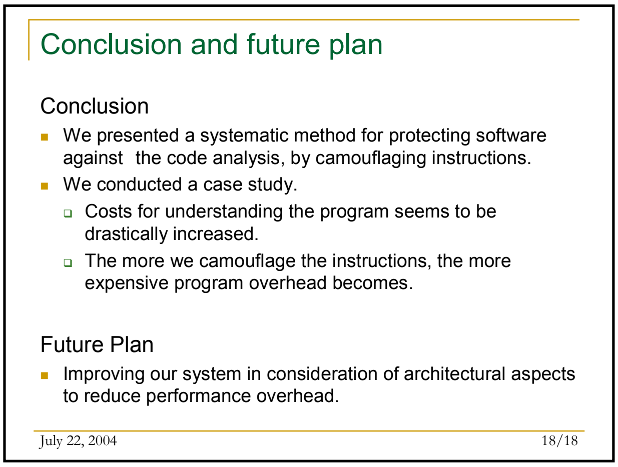
Target Assembly Program

- # of instructions : 1000
- # of camouflaged inst. : 130

Distance between camouflaged insts. and restoring routines.

- Average : 151[inst.]
- Maximum : 611[inst.]

Costs for understanding this program are drastically increased.



Conclusion and future plan

Conclusion

- We presented a systematic method for protecting software against the code analysis, by camouflaging instructions.
- We conducted a case study.
  - Costs for understanding the program seems to be drastically increased.
  - The more we camouflage the instructions, the more expensive program overhead becomes.

Future Plan

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