Toward Secure Data Obfuscation

7th COE Postdoctoral and Doctoral Researchers Technical Presentation
October 27th , 2005

Software Engineering Lab.
Yuichiro Kanzaki

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 the secret key[1].
- An attacker analyzes a program embedded in a set-top box and steals the device key[2].

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.

Goal

We develop an effective obfuscation method to protect secret parts of software.

Obfuscation :
Transforming a program into an equivalent one that is harder to reverse engineer[1]

```
int a = 2,b = 3;

for(i=0;i<5;i++){
  a *= pow(b,i);
}

printf("%d",a);
```

original program

obfuscate

```
int a[] = {2,11};
int i=1;
LOOP:
if(i!=5){a[0] *=
pow((a[1]-2)/3,i);
i++;goto LOOP;
}printf("%d",a[0]);
```

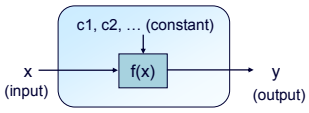
obfuscated program

Developers can reduce a risk of attack to their program

[1] C. Collberg, C. Thomborson, ``Watermarking, Tamper-Proofing, and Obfuscation — Tools for Software Protection," IEEE Trans. Software Eng., vol.28, no.8, pp.735-746, June 2002.

Target and Attacker Model

- Target
  A function that has a secret expression *f(x)* which calculates *y* using input *x* and internal constant values *c1, c2,…*




```
unsigned char f (int x)
{
   int c1=3, c2=5, y;

   y = c1 * x + c2;
   return y;
}
```

- Attacker model
  - Attackers can use tools for analyzing executable programs
  - Attackers try to obtain *f(x)* that must be kept secret

Our goal is to make the program difficult to guess *f(x)*

Problem of Conventional Obfuscation

It is easy to guess *f(x)* from the obfuscated function since there is a similar expression

```
unsigned char f (int x)
{
   int c1 =3, c2=5, y;

   y = c1 * x + c2;
   return y;
}
```

f(x)

Original Function

Obfuscation

```
unsigned char f (int x)
{
   int c1=7, c2=11, y;

   y = (c1+1) * x + c2;
   return (y-1)/2;
}
```

Obfuscated Function

We hide the expression that could be clue to obtain the secret expression *f(x)*

Key idea of our method (1/3)

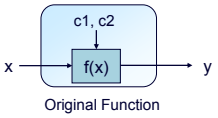
Hiding *f(x)* using an I/O table relating input to output

```
unsigned char f (int x)
{
   int c1=3, c2=5, y;

   y = c1 * x + c2;
   return y;
}
```

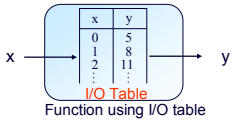
f(x)

obfuscate

```
unsigned char F[] = {
   5, 8, 11,.....
}
unsigned char f (int x)
{
   return F[x];
}
```



Original Function



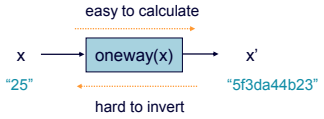
Function using I/O table

It is easy to guess *f(x)* since the relation between x and y is clear

Key idea of our method (2/3)

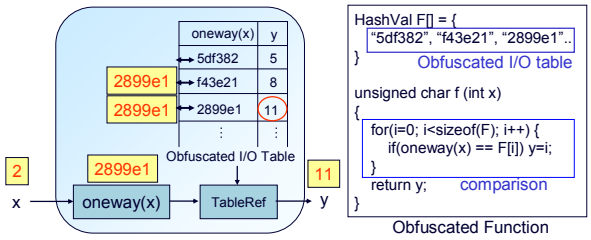
We obfuscate the relation between input *x* and output *y* using a one-way function.

One-way function:
A function which is easy to calculate but highly hard to invert (e.g. MD5, SHA-1)



Key idea of our method (3/3)

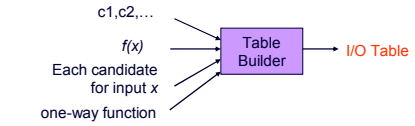
We construct an obfuscated function that has a table relating the result of the one-way function of *x* to the output *y*.



To guess input *x* from output *y* is as difficult as inverting the one-way function.

Procedure of obfuscation (1/2)

1. Create an I/O table from the target function




```
unsigned char f (int x)
{
  int c1=3, c2=5, y;

  y = c1 * x + c2;
  return y;
}
```

Target Function

x	y	oneway(x)
0	5	5df382
1	8	f43e21
2	11	2899e1
3	13	e3dad0
⋮	⋮	⋮

I/O Table

The size of the table depends on the number of candidates for input

Procedure of obfuscation (2/2)

2. Generate the obfuscated function from the I/O table

x	y	oneway(x)
0	5	5df382
1	8	f43e21
2	11	2899e1
3	13	e3dad0
⋮	⋮	⋮

I/O Table

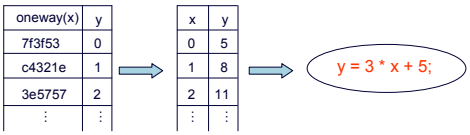
```
HashVal F[] = {
  "5df382", "f43e21", "2899e1"..
}
oneway(x)

unsigned char f (int x)
{
  for(i=0; i<sizeof(F); i++) {
    if(oneway(x) == F[i]) y=i;
  }
  return y;
}
comparison
```

Obfuscated Function

The number of comparison time depends on the size of table

Discussion of security

How difficult to obtain *f(x)* from the obfuscated function?



Obfuscated I/O table | Original I/O table | *f(x)* in Original Function

- As long as the attacker analyzes the function statically, to guess *f(x)* from the obfuscated I/O table is impossible
- To generate the original I/O table based on the behavior during execution, the difficulty depends on the number of candidates for input

Experiment (1/3) : Overview

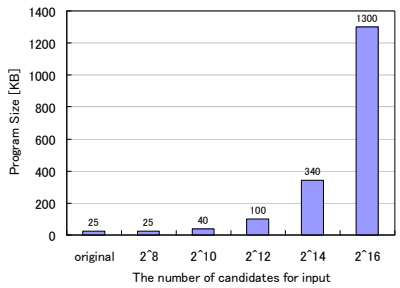
We evaluated the overhead of the obfuscated program

- Evaluation Items
  - Size Overhead
  - Performance Overhead
- Target Function
  - A simple function which has 1 input and 1 expression
  - We call the function 10 times
  - We use MD5 algorithm
  - The number of candidates for input was varied

```
int f (int x)
{
   int c1=3, c2=5, y;
   y = c1 * x + c2;
   return y;
}

int main() {
  for(i=0; i<10; i++) {
    f(rand()*candidates);
  }
}
```

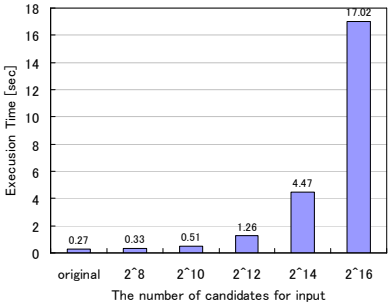
Target Program (original)

Experiment (2/3) : Size Overhead



The program size increases in proportion to the number of candidates for input due to I/O tables.

Experiment (3/3) : Performance Overhead



The execution time increases in proportion to the number of candidates for input due to comparison routines.

Conclusion and Future Work

Conclusion

- We have proposed an obfuscation method using I/O table and one-way function
- The method drastically increases the cost of analyzing
- The current system imposes significant overhead

Future Work

- Improving our system to reduce performance overhead
- Investigation of the suitable application domain