Implementation of an Obfuscation Tool for C/C++ Source Code Protection on the XScale Architecture[*]

Seongje Cho[1], Hyeyoung Chang[1], and Yookun Cho[2]

[1] Dept. of Computer Science & Engineering, Dankook University, Gyeonggi-do, Korea
[2] School of Computer Science and Engineering, Seoul National University, Seoul, Korea
{sjcho,hychang}@dankook.ac.kr, cho@os.snu.ac.kr

Abstract. Obfuscation is one of the most effective methods to protect software against malicious reverse engineering intentionally making the code more complex and confusing. In this paper, we implement and evaluate an obfuscation tool, or *obfuscator* for protecting the intellectual property of C/C++ source code. That is, this paper presents an implementation of a code obfuscator, a tool which transforms a C/C++ source program into an equivalent one that is much harder to understand. We have used the ANTRL parser generator for parsing C/C++ programs, and applied some obfuscation algorithms. Performance analysis is conducted by executing two obfuscated programs on the XScale architecture to establish the relationship between the complexity and the performance of each program. When the obfuscated source code has been compared with the original source code, it has enough effectiveness in terms of potency and resilience though it incurs some run-time overhead.

Keywords: Obfuscation, Source Code Protection, Reverse Engineering.

1 Introduction

The major types of attack against software protection mechanisms can be classified as software piracy, malicious reverse engineering, and tampering. *Software piracy* is the illegal distribution and/or reproduction of software applications for business or personal use. Global PC software piracy alone accounted for nearly $40 billion annual loss [1] to the software industry in 2006. Many software developers therefore try to protect their programs against illegal copying. They also worry about their applications being *reverse engineered* [2,3,4,5]. Certain classes of automated reverse engineering tools can successfully attack compiled software to expose underlying code. In some cases, a valuable piece of code may be extracted from an application and incorporated into a competitor's code. Another related threat is *software tampering* [2,5,6]. Any illicit modification of program file or attack against program integrity should make the software unusable.

As the use of a client code like 'mobile agent' programs downloaded or installed on a host becomes more general, the client software is more frequently threatened by

[*] This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD, Basic Research Promotion Fund)(KRF-2008-314-D00340).

the host. This results from the power of the adversary model in digital rights management (DRM) systems, which is significantly more vulnerable than in the traditional security scenarios. The adversary can even gain complete control of the client node–supervisory privileges along with the full physical as well as architectural object observational capabilities. Unfortunately, the traditional security techniques to protect software from malicious client may not be applicable to protect a client code against a host attack [2, 3]. As a result, software protection has recently attracted tremendous commercial interest, from major software vendors to mobile DRM venders.

While it is generally believed that complete protection of software is an unattainable goal, recent results have shown that some degree of protection can be achieved. Software watermarking, obfuscation, and tamper-proofing have emerged as feasible methods for the intellectual property (IP) protection of software [2-11]. Watermarking, a defense against software piracy, is a process that makes it possible to determine the origin of software. Obfuscation, a defense against reverse engineering, is a process that renders software unintelligible but still functional. Tamper-proofing, a defense against tampering, is a process so that unauthorized modifications to software (for example, to remove a watermark) will result in nonfunctional code.

In this paper, we focus only on obfuscation techniques useful for protecting software from reverse engineering. The paper describes the implementation and evaluation of an obfuscation tool which converts a C/C++ source codes into an equivalent one that is much harder to understand. We implement some obfuscation algorithms on the XScale architecture and evaluate the performance and effectiveness of the obfuscation tool in terms of potency, resilience, and cost.

The rests of the section in this paper is organized as follows. Section 2 explains obfuscation, its related work, and the evaluation metrics of obfuscation. It is then followed by the description of the proposed method in section 3. Section 4 describes the implementation of obfuscation algorithms. We present the performance results of our implementation in section 5. Finally, section 6 concludes the paper.

2 Obfuscation

Software obfuscation can be defined as a semantics-preserving code transformation of a program in an attempt to make the code as complex and confusing as possible. Obfuscation protects the intellectual property (IP) of software from reverse-engineering attacks. The IP can be the software design, algorithms, or data contained in the software. Obfuscating transformations are primarily classified depending on the kind of information they target. Some simple transformations target the lexical structure (the layout) of the program while others target the data structures used by the program or its flow of control [2,4,7,8,11].

Layout obfuscations are aimed at making the code unreadable by introducing 'formatting change', 'remove comments', 'remove debug information', and 'scramble identifiers' methods. Most commercial obfuscators fall in this category. Crema, one of the oldest Java obfuscators, uses layout obfuscation. Data obfuscations are aimed at obscuring data and data structures used in the program. These data transformations can be classified into the following methods: 'split variables', 'array transformation including splitting and folding', and 'modifying inheritance including class split and class insertion'.

Control obfuscations are aimed at obfuscating the flow of execution by applying 'opaque construct', 'redundant code introducing opaque predicates and multiple obfuscated loops', 'inline removing procedural abstraction', and 'outline creating bogus procedural abstraction' algorithms [2,7,8]. Several control obfuscations rely on the existence of *opaque variables* and *opaque predicates*. A variable V is opaque if it has some property q which is known a priori to the obfuscator, however is difficult for de-obfuscator to deduce. Similarly, a predicate P (a Boolean expression) is opaque if its outcome is known at obfuscation time, but is difficult for the de-obfuscator to deduce. We write P^T (P^F) if P always evaluates to TRUE (FALSE), and $P^?$ if P may sometimes evaluates to TRUE and sometimes to FALSE.

In general, three criteria are considered in evaluating the quality of an obfuscation method; including potency, resilience, and cost [2-9]. The potency refers to what degree the transformed code is more obscure than the original. Software complexity metrics define various complexity measures for software, such as number of predicates it contains, depth of its inheritance tree, nesting levels, etc. While the goal of good software design is to minimize complexity based on these parameters, the goal of obfuscation is to maximize it.

The resilience of the software is a measure of how well the transformed code can resist attacks from either the programmer or an automatic de-obfuscator. It is a combination of the programmer effort to create a de-obfuscator and the time and space required by the de-obfuscator. The highest degree of resilience is a *one-way* transformation that cannot be undone by a de-obfuscator. An example is when the obfuscation removes information such as source code formatting. The difference between potency and resilience is that a transformation is potent if it can confuse a human reader, whereas it is resilient if a de-obfuscator tool cannot undo the transformation.

The cost of a transformation defines to how much computational overhead is added to the obfuscated program. Examples of the cost are the extra execution time and space penalty incurred by the obfuscation.

There are many software protection tools such as Cloakware, DashO, Dotfuscator, Kava (Konfused Java), JHide, and Semantic Designs' source code obfuscator [2,4,5,9,10,11]. Cloakware is capable of providing significant control and dataflow obfuscations of C source code. DashO and Dotfuscator can construct layout transformations including dead code removal and identifier renaming for Java and Microsoft Intermediate Language (MSIL), respectively. Semantic Designs' source code obfuscators provide a software developer with identifier renaming and optional whitespace removal for several high-level languages. A tool called Sandmark measures the effectiveness of software-based methods for protecting software against piracy, reverse engineering, and tampering [4]. MacBride et. al. [9] presented a qualitative measurement of the capability of two commercial obfuscators, DashO-Pro and KlassMaster. The measurement showed the two obfuscators both could cause variations in the performance of the algorithms used for testing.

3 The Structure of C/C++ Source Code Obfuscator

The approach we are going to consider is source code obfuscation to protect intellectual property embedded in C/C++ source programs. The source code obfuscator accepts a source file, and generates another functionally equivalent source file which

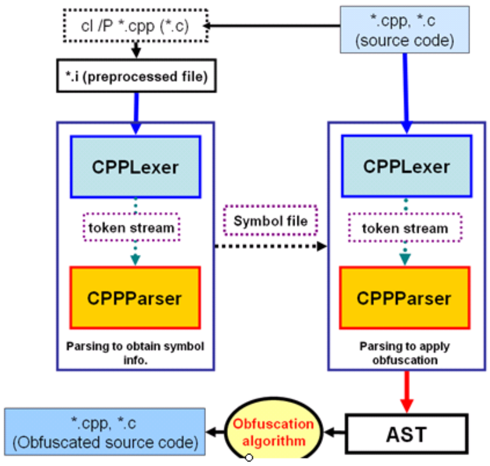


Fig. 1. Structure of high-level obfuscator

is much harder to understand or reverse-engineer. This is useful for technical protection of intellectual property in the following cases[1]. First, the source code must be delivered for public execution purposes. Second, commercial software components must be delivered in source form for direct integration by a customer into her end product (portable applications in C or PHP etc., code libraries or hardware components coded in Verilog or VHDL). Third, we have to send test cases derived from proprietary code to vendors. Fourth, an object code still contains many clues such as class public methods used only inside an application, as with java class files.

Figure 1 shows the overall structure of our source code obfuscator. We use a parser generator called ANTRL, *ANother Tool for Language Recognition* [12], to obfuscate the C/C++ source programs. The parser generated by the ANTRL takes C/C++ programs as input and analyzes a sequence of tokens to determine grammatical structure with respect to a given formal grammar. It captures the implied hierarchy of the input text and transforms it into abstract syntax tree (AST), or just syntax tree. The parser can use a separate lexical analyzer (*lexer*) to create tokens from the sequence of input characters. The AST is a finite, labeled, and directed tree, where each interior node represents a programming language construct and the children of that node represent meaningful components of the construct. It is used in the parser as an intermediate between a parse tree and a data structure. Based on the information contained in the AST, we implement the obfuscation algorithms by inserting, modifying, and restructuring a proper node after locating the node to apply the algorithms.

The obfuscation tool consists of two parts; one part shown in left side of Figure 1 obtains symbol information and the other part shown in right side constructs obfuscation algorithms utilizing the derived symbol information. The symbol information includes the attributes of identifiers such as the name, type, and size of all the variables. We can finally transform an original source program into an obfuscated source program by both using the symbol information and reconstructing the AST.

[1] http://www.semdesigns.com/Products/Obfuscators

4 The Implementation of Obfuscation Algorithms

In the remainder of this paper we will describe and evaluate various obfuscating transformations. We start by formalizing the notion of an *obfuscating transformation*. Given a set of obfuscating transformations $T = \{T_1, \ldots, T_n\}$ and a program C consisting of source code objects (classes, methods, statements, etc.) $\{S_1, \ldots, S_k\}$, find a new program $C' = \{ \ldots, S'_j = T_i(S_j), \ldots\}$ such that C' has the same observable behavior as C, i.e., the transformations are *semantics-preserving*. Our obfuscator have currently implemented some obfuscation algorithms: *modifying an original program's layout*, *splitting variables*, *restructuring arrays*, *extending loop conditions*, and *adding redundant operand*. As the target programs to apply the obfuscation algorithms, we have selected three programs, bubblesort, advanced encryption standard (AES), and Diffie-Hellman key exchange programs. In this section, we mainly consider the original source code and the obfuscated code of the AES program.

4.1 Layout Transformations

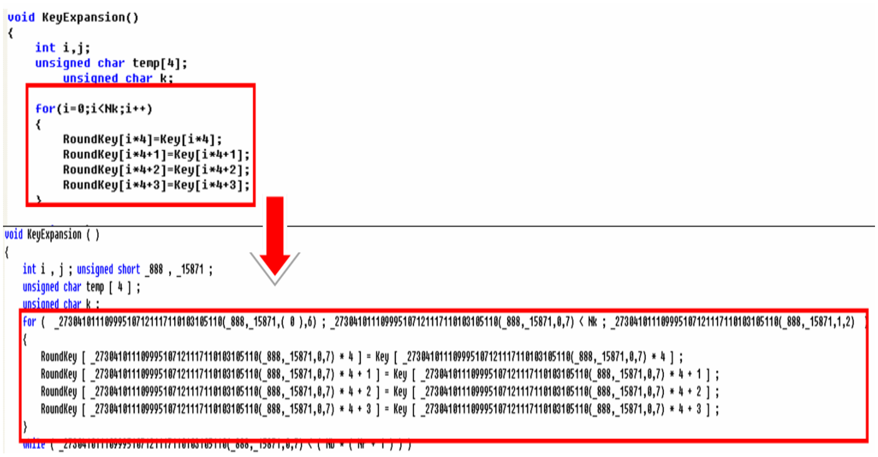
We first introduce layout obfuscation altering the formatting of the source file. This involves removing source code comments, and changing the names of elements such as the class, member variables, and the local variable. Source code comment removal and formatting removal are free transformations, since there is no increase in space and time from the original application. The potency is low because there is very little semantic content in formatting. It is a one-way transformation because the formatting, once removed, cannot be recovered. Scrambling of variable names is also a one-way and free transformation, but it has much higher potency than formatting removal.

4.2 Split Variable

Integer variables and other variables of restricted range can be split into two or more variables. Figure 2 shows an example where the splitting principle is applied to integer variables. Here, the elements of i are distributed over two short variables, _888 and _15871. The algorithm can sometimes substitute a target variable with a function which returns the same value as the variable. The potency, resilience, and cost of this method all increase with the number of variables into which the original variable is split.

4.3 Restructure Arrays: Array Folding

A number of transformations can be devised for obscuring operations performed on arrays: we are trying for a programmer to be able to split an array into several subarrays, merge two or more arrays into one array, fold an array (increasing the number of dimensions), or flatten an array (decreasing the number of dimensions). Figure 3 demonstrates how a one-dimensional array sbox can be folded into a two-dimensional array sbox. Array folding increases the data structure complexity of the potency metrics.

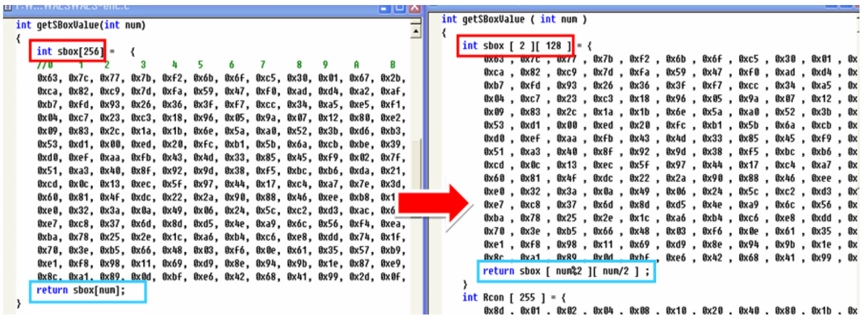


```
void KeyExpansion()
{
    int i,j;
    unsigned char temp[4];
        unsigned char k;

    for(i=0;i<Nk;i++)
    {
        RoundKey[i*4]=Key[i*4];
        RoundKey[i*4+1]=Key[i*4+1];
        RoundKey[i*4+2]=Key[i*4+2];
        RoundKey[i*4+3]=Key[i*4+3];
    }
```

```
void KeyExpansion ( )
{
   int i , j ; unsigned short _888 , _15871 ;
   unsigned char temp [ 4 ] ;
   unsigned char k ;
   for (  _2730410111099951071211171101031051110(_888,_15871,( 0 ),6) ; _2730410111099951071211171101031051110(_888,_15871,0,7) < Nk ; _2730410111099951071211171101031051110(_888,_15871,1,2)  )
   {
      RoundKey [ _2730410111099951071211171101031051110(_888,_15871,0,7) * 4 ] = Key [ _2730410111099951071211171101031051110(_888,_15871,0,7) * 4 ] ;
      RoundKey [ _2730410111099951071211171101031051110(_888,_15871,0,7) * 4 + 1 ] = Key [ _2730410111099951071211171101031051110(_888,_15871,0,7) * 4 + 1 ] ;
      RoundKey [ _2730410111099951071211171101031051110(_888,_15871,0,7) * 4 + 2 ] = Key [ _2730410111099951071211171101031051110(_888,_15871,0,7) * 4 + 2 ] ;
      RoundKey [ _2730410111099951071211171101031051110(_888,_15871,0,7) * 4 + 3 ] = Key [ _2730410111099951071211171101031051110(_888,_15871,0,7) * 4 + 3 ] ;
   }
   while ( _2730410111099951071211171101031051110(_888,_15871,0,7) < ( Nb * ( Nr + 1 ) ) )
```

Fig. 2. A data transformation that splits variables



```
int getSBoxValue(int num)
{
    int sbox[256] =   {
    //0     1     2     3     4     5     6     7     8     9     A     B
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x1
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x6
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f,
    return sbox[num];
}
```

```
int getSBoxValue ( int num )
{
   int sbox [ 2 ][ 128 ] = {
        0x63 , 0x7c , 0x77 , 0x7b , 0xf2 , 0x6b , 0x6f , 0xc5 , 0x30 , 0x01 , 0x
        0xca , 0x82 , 0xc9 , 0x7d , 0xfa , 0x59 , 0x47 , 0xf0 , 0xad , 0xd4 , 0x
        0xb7 , 0xfd , 0x93 , 0x26 , 0x36 , 0x3f , 0xf7 , 0xcc , 0x34 , 0xa5 , 0x
        0x04 , 0xc7 , 0x23 , 0xc3 , 0x18 , 0x96 , 0x05 , 0x9a , 0x07 , 0x12 , 0x
        0x09 , 0x83 , 0x2c , 0x1a , 0x1b , 0x6e , 0x5a , 0xa0 , 0x52 , 0x3b , 0x
        0x53 , 0xd1 , 0x00 , 0xed , 0x20 , 0xfc , 0xb1 , 0x5b , 0x6a , 0xcb , 0x
        0xd0 , 0xef , 0xaa , 0xfb , 0x43 , 0x4d , 0x33 , 0x85 , 0x45 , 0xf9 , 0x
        0x51 , 0xa3 , 0x40 , 0x8f , 0x92 , 0x9d , 0x38 , 0xf5 , 0xbc , 0xb6 , 0x
        0xcd , 0x0c , 0x13 , 0xec , 0x5f , 0x97 , 0x44 , 0x17 , 0xc4 , 0xa7 , 0x
        0x60 , 0x81 , 0x4f , 0xdc , 0x22 , 0x2a , 0x90 , 0x88 , 0x46 , 0xee , 0x
        0xe0 , 0x32 , 0x3a , 0x0a , 0x49 , 0x06 , 0x24 , 0x5c , 0xc2 , 0xd3 , 0x
        0xe7 , 0xc8 , 0x37 , 0x6d , 0x8d , 0xd5 , 0x4e , 0xa9 , 0x6c , 0x56 , 0x
        0xba , 0x78 , 0x25 , 0x2e , 0x1c , 0xa6 , 0xb4 , 0xc6 , 0xe8 , 0xdd , 0x
        0x70 , 0x3e , 0xb5 , 0x66 , 0x48 , 0x03 , 0xf6 , 0x0e , 0x61 , 0x35 , 0x
        0xe1 , 0xf8 , 0x98 , 0x11 , 0x69 , 0xd9 , 0x8e , 0x94 , 0x9b , 0x1e , 0x
        0x8c , 0xa1 , 0x89 , 0x0d , 0xbf , 0xe6 , 0x42 , 0x68 , 0x41 , 0x99 , 0x
      return sbox [ num%2 ][ num/2 ] ;
   }
   int Rcon [ 255 ] = {
        0x8d , 0x01 , 0x02 , 0x04 , 0x08 , 0x10 , 0x20 , 0x40 , 0x80 , 0x1b , 0x
```

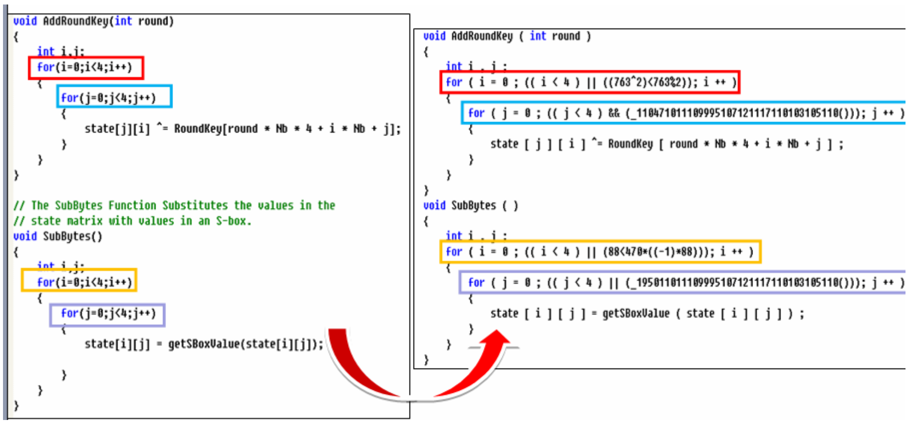
Fig. 3. Array restructuring: Array folding

4.4 Extend Loop Conditions

Figure 4 shows how we can obfuscate a loop by making the termination condition more complex. The basic idea is to extend the loop condition with a P^T or P^F predicate which will not affect the number of times the loop will execute. In Figure 4, our obfuscator has added to the termination condition of the loop the '&&' operator followed by the predicate P^T which will always evaluate to TRUE, and the '||' operator followed by the predicate P^F which will always evaluate to FALSE.

4.5 Add Redundant Operand

By constructing some opaque variables, we can use algebraic laws to add redundant operands to arithmetic expressions. This will increase the program length metric of the potency metrics. Obviously, this method works best with integer expressions where numerical accuracy is not an issue. In the obfuscated statement in Figure 5, we construct an *opaque sub-expression* (int) (856* 0.0001)*4 whose value is 4.



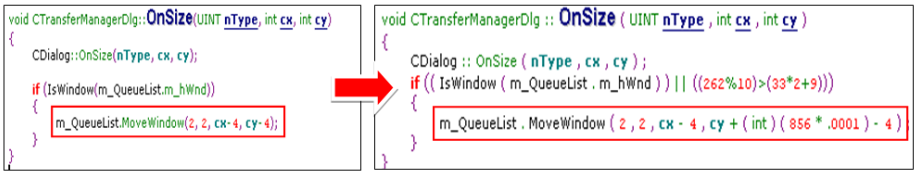
```
void AddRoundKey(int round)
{
    int i,j;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            state[j][i] ^= RoundKey[round * Nb * 4 + i * Nb + j];
        }
    }
}

// The SubBytes Function Substitutes the values in the
// state matrix with values in an S-box.
void SubBytes()
{
    int i,j;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            state[i][j] = getSBoxValue(state[i][j]);

        }
    }
}
```

```
void AddRoundKey ( int round )
{
    int i , j ;
    for ( i = 0 ; (( i < 4 ) || ((763^2)<763%2)); i ++ )
    {
        for ( j = 0 ; (( j < 4 ) && (_110471011109995107121117110103105110())); j ++ )
        {
            state [ j ] [ i ] ^= RoundKey [ round * Nb * 4 + i * Nb + j ] ;
        }
    }
}
void SubBytes ( )
{
    int i , j ;
    for ( i = 0 ; (( i < 4 ) || (88<470*((-1)*88))); i ++ )
    {
        for ( j = 0 ; (( j < 4 ) || (_195011011109995107121117110103105110())); j ++ )
        {
            state [ i ] [ j ] = getSBoxValue ( state [ i ] [ j ] ) ;
        }
    }
}
```

Fig. 4. Loop condition insertion



```
void CTransferManagerDlg::OnSize(UINT nType, int cx, int cy)
{
    CDialog::OnSize(nType, cx, cy);

    if (IsWindow(m_QueueList.m_hWnd))
    {
        m_QueueList.MoveWindow(2, 2, cx-4, cy-4);
    }
}
```

```
void CTransferManagerDlg :: OnSize ( UINT nType , int cx , int cy )
{
    CDialog :: OnSize ( nType , cx , cy ) ;
    if (( IsWindow ( m_QueueList . m_hWnd ) ) || ((262%10)>(33*2+9)))
    {
        m_QueueList . MoveWindow ( 2 , 2 , cx - 4 , cy + ( int ) ( 856 * .0001 ) - 4 )
    }
}
```

Fig. 5. Add redundant operand

5 Performance Evaluation

The transformation constructing the obfuscation algorithms may increase execution time, program complexity, and cost. We think there will always be a trade-off between the level of obfuscation and the performance overhead incurred. In this section, we have analyzed the quality of the obfuscation algorithms on an embedded board equipped with the Intel XScale PXA255 400MHz CPU, 128 megabyte SDRAM, and 32 megabyte Flash ROM. Embedded Linux kernel 2.4.19, g++ compiler, and the AES and Diffie-Hellman programs have been used for performing the experiments. The potency, resilience, and cost are considered in evaluating the quality of obfuscation methods: 'layout transformations', 'split variable', 'array folding', 'extend loop conditions', and 'add redundant operand'.

5.1 Measures of Potency

Even though there are many complexity metrics to evaluate the degree of the potency [8], we consider only some of the complexity measures listed in Table 1. The goal of an obfuscating method is to maximize these measures. The *potency* is measured by the summation of the series for the five complexity values in Table 1. An obfuscation method is a *potent obfuscating transformation* if the following equation, its *relative potency ratio* with respect to a program, is satisfied.

$$\{Potency(\text{obfuscated program}) \,/\, Potency(\text{original program})\} - 1 > 0 \tag{1}$$

Table 1. Overview of some software complexity measures

Metric	Metric name and Its meaning
μ_1	Program Length
	Complexity of a program increases with the number of operators and operands in a program
μ_2	Cyclomatic Complexity
	Complexity of a function or method increases with the number of predicates in a function or method
μ_3	Nesting Complexity
	Complexity of a function or method increases with the nesting level of conditionals
μ_4	Data Flow Complexity
	Complexity of a function or method increases with the number of inter-basic block variable references
μ_5	Fan-in/out Complexity
	Complexity of a function or method increases with the number of formal parameters to the function or method, and with the number of global data structures read or updated by the function or method.

Table 2 shows the complexity values and relative potency ratio obtained by measuring the five metric values of the AES and Diffie-Hellman programs. In Table 2, we can see that the obfuscator has increased the relative potency ratio by 0.675 for the AES program and 0.848 for the Diffie-Hellman program, respectively when both data and control transformations were applied.

Table 2. Complexity and potency ratio of each code before and after applying obfuscation

	AES				Diffie-Hellman			
	Original	Data	Control	Data+Control	Original	Data	Control	Data+Control
μ_1	10356	15605	13264	17311	3299	5001	4519	6094
μ_2	17	23	38	32	22	30	31	41
μ_3	21	25	43	37	21	25	30	37
μ_4	29	50	64	71	18	34	35	50
μ_5	12	21	27	32	26	33	31	36
Potency ratio		0.507	0.288	0.675		0.513	0.372	0.848

5.2 Measures of Resilience

It is not easy to quantitatively measure resilience of the obfuscated codes. As shown in Figure 6, we measure it on a scale from trivial to one-way according to the criteria proposed by Collberg et. al. in [8]. One-way transformations are the highest resilience

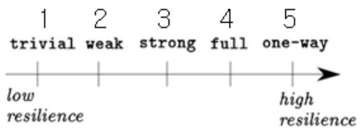


Fig. 6. Resilience of an obfuscating method

Table 3. Resilience of the implemented algorithms

Target	Transformation Algorithm	Resilience	Value
Layout	Remove Comments	One-way	5
Control flow	Extend Loop Condition	Weak ~ Strong	2~3
	Add Redundant Operands	Weak ~ Strong	2~3
Data	Split Variable	Weak	2
	Fold Array	Weak	2

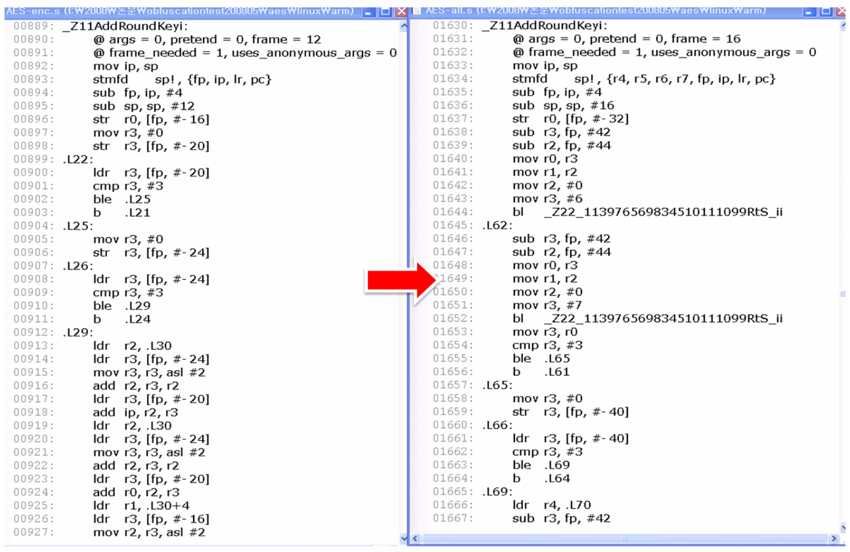
in the sense that they can never be undone. Other transformations add unnecessary information to the program that do not change its functional behavior, however which make it difficult to construct an automatic tool to undo the transformations or executing such a tool will be extremely time-consuming. Table 3 shows the resilience of the obfuscated algorithms implemented in Section 4.

5.3 Measures of Cost

We measured the file size and execution time of the target programs before and after applying obfuscation methods. The experimental results are shown in Table 4. Each execution time of the AES encryption and Diffie-Hellman key distribution programs present the average time consumed to encrypt a plaintext file of 262144 bytes and to generate a secret key of 128 bits, respectively. We can see from the table that the obfuscator increases the file size and execution time of the obfuscated programs.

Table 4. File size (in *bytes*) and execution time (in *seconds*) before and after applying obfuscation

	AES				Diffie-Hellman			
	Original	Data	Control	Data+ Control	Original	Data	Control	Data+ Control
Source file size	9658	15605	13352	17332	3299	5001	4519	6094
Object file size	9180	13228	13200	15416	2904	3748	3896	4660
Execution time	6.610s	7.666s	6.677s	7.711s	0.176s	0.210s	0.225s	0.250s


AES-enc.s (f:₩2008₩논문₩obfuscationtest200805₩aes₩linux₩arm)

```
_Z11AddRoundKeyi:
        @ args = 0, pretend = 0, frame = 12
        @ frame_needed = 1, uses_anonymous_args = 0
        mov ip, sp
        stmfd     sp!, {fp, ip, lr, pc}
        sub  fp, ip, #4
        sub  sp, sp, #12
        str   r0, [fp, #-16]
        mov r3, #0
        str   r3, [fp, #-20]
.L22:
        ldr   r3, [fp, #-20]
        cmp r3, #3
        ble   .L25
        b     .L21
.L25:
        mov r3, #0
        str   r3, [fp, #-24]
.L26:
        ldr   r3, [fp, #-24]
        cmp r3, #3
        ble   .L29
        b     .L24
.L29:
        ldr   r2, .L30
        ldr   r3, [fp, #-24]
        mov r3, r3, asl #2
        add  r2, r3, r2
        ldr   r3, [fp, #-20]
        add  ip, r2, r3
        ldr   r2, .L30
        ldr   r3, [fp, #-24]
        mov r3, r3, asl #2
        add  r2, r3, r2
        ldr   r3, [fp, #-20]
        add  r0, r2, r3
        ldr   r1, .L30+4
        ldr   r3, [fp, #-16]
        mov r2, r3, asl #2
```

AES-all.s (f:₩2008₩논문₩obfuscationtest200805₩aes₩linux₩arm)

```
_Z11AddRoundKeyi:
        @ args = 0, pretend = 0, frame = 16
        @ frame_needed = 1, uses_anonymous_args = 0
        mov ip, sp
        stmfd     sp!, {r4, r5, r6, r7, fp, ip, lr, pc}
        sub  fp, ip, #4
        sub  sp, sp, #16
        str   r0, [fp, #-32]
        sub  r3, fp, #42
        sub  r2, fp, #44
        mov r0, r3
        mov r1, r2
        mov r2, #0
        mov r3, #6
        bl     _Z22_11397656983451011109 9RtS_ii
.L62:
        sub  r3, fp, #42
        sub  r2, fp, #44
        mov r0, r3
        mov r1, r2
        mov r2, #0
        mov r3, #7
        bl     _Z22_113976569834510111099RtS_ii
        mov r3, r0
        cmp r3, #3
        ble   .L65
        b     .L61
.L65:
        mov r3, #0
        str   r3, [fp, #-40]
.L66:
        ldr   r3, [fp, #-40]
        cmp r3, #3
        ble   .L69
        b     .L64
.L69:
        ldr   r4, .L70
        sub  r3, fp, #42
```

Fig. 7. Comparison of assembly codes before and after applying obfuscation

5.4 Comparison of Assembly Codes

Finally, we have compared the ARM assembly language of an original program with that of its obfuscated one to check if the transformation algorithms are effective in the machine-level code. Figure 7 shows the assembly codes corresponding to some part of the function *AddRoundKey()* in the AES program. The right side part of the figure shows the assembly code after applying two obfuscation algorithms, '*split variable*' and '*extend loop condition*'. The assembly code of the obfuscated function is quite different from that of the original one. As a result, our C/C++ obfuscator for the XScale architecture is effective even though it incurs some space and time overhead.

6 Conclusion and Future Work

This paper presents the implementation of an obfuscation tool, or *obfuscator* on the XScale architecture that protects C/C++ source code against malicious reverse engineering by making the code as complex and confusing as possible, but still functional. To render software unintelligible, the obfuscator uses layout transformations, data transformations including '*split variable*' and '*fold array*', and control transformations such as '*extend loop conditions*' and '*add redundant operand*'. We have also evaluated the quality of obfuscation methods using three criteria: potency, resilience, and cost. Experimental results have shown that our obfuscator can enhance the potency and resilience of the obfuscated code, but incur some space penalty and the extra execution time.

The future work for this research is to continue to introduce other obfuscation algorithms in this obfuscator to make more obscure the control-flow of the source program and the data structure used in it. We will also develop another obfuscation method for a low-level program like assembly or machine languages, and then incorporate it with the current obfuscation method.

References

1. Business Software Alliance, Fourth Annual BSA and IDC Global Software Piracy Study (2006)
2. Collberg, C.S., Thomborson, C.: Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection. IEEE Transactions on Software Engineering 28(8), 735–746 (2002)
3. Gomathisankaran, M., Tyagi, A.: Architecture Support for 3D Obfuscation. IEEE Transaction on Computer 55(5), 497–507 (2006)
4. Collberg, C., Myles, G., Huntwork, A.: Sandmark – Tool for Software Protection Research. IEEE Security & Privacy (Software Protection), 40–49 (July/August 2003)
5. Naumovicb, G., Memon, N.: Preventing Piracy, Reverse Engineering, and Tampering. IEEE Computer, 64–71 (2003)
6. Fu, B., Richard III, G.G., Chen, Y., Husseiny, A.: Some New Approaches For Preventing Software Tampering. In: Proc. of the 44th ACM Southeast Regional Conference (ACM SE 2006), pp. 655–660 (2006)
7. van Oorschot, P.C.: Revisiting Software Protection. In: Boyd, C., Mao, W. (eds.) ISC 2003. LNCS, vol. 2851, pp. 1–13. Springer, Heidelberg (2003)
8. Collberg, C., Thomborson, C., Low, D.: A Taxonomy of Obfuscating Transformations. Technical report 148, Dept. of Computer Science, University of Auckland, New Zealand (1997)
9. MacBride, J., Mascioli, C., Marks, S., Tang, G., Head, L.M.: A Comparative Study of Java Obfuscators. In: IASTED International Conference on Software Engineering and Applications, Phoenix, Arizona, November 14 –16, 2005, pp. 82–86 (2005)
10. Ertaul, L., Venkatesh, S.: JHide – a tool kit for code obfuscation. In: Proc. of the 8th IASTED International Conference Software Engineering and Applications (2004)
11. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static Disassembly of Obfuscated Binaries. In: Proc. of the 13th USENIX Security Symposium, pp. 255–270 (2004)
12. ANTLR, `http://www.antlr.org`