PAPER *Special Section on Cryptography and Information Security*

Software Obfuscation on a Theoretical Basis and Its Implementation

Toshio OGISO[†*a)], **Yusuke SAKABE**[†], **Masakazu SOSHI**[†], ***Nonmembers,***
and **Atsuko MIYAJI**[†], ***Regular Member***

SUMMARY Software obfuscation is a promising approach to protect intellectual property rights and secret information of software in untrusted environments. Unfortunately previous software obfuscation techniques share a major drawback that they do not have a theoretical basis and thus it is unclear how effective they are. Therefore we propose new software obfuscation techniques in this paper. The techniques are based on the difficulty of interprocedural analysis of software programs. The essence of our obfuscation techniques is a new complexity problem to precisely determine the address a function pointer points to in the presence of arrays of function pointers. We show that the problem is NP-hard and the fact provides a theoretical basis for our obfuscation techniques. Furthermore, we have already implemented a prototype tool that obfuscates C programs according to our proposed techniques and in this paper we describe the implementation and discuss the experiments results.

key words: *tamper-resistant software, obfuscation, static analysis, computational complexity*

1. Introduction

The way of software distribution has been changing with the rapid spread of computer networks such as the Internet. Namely, although almost all of conventional software distribution was in binary code form, now it is becoming more common to circulate software in source code form. Notable examples of such a way of software distribution are via perl scripts, Java applets, and JavaScripts. Especially in mobile agent systems, which are paid much attention to as computing environments in the next generation, software programs called mobile agents move around over networks and they also stimulate a trend of software distribution in source code form because the distribution in such a form facilitates the execution of mobile agents on platforms of different architectures.

In such situations, malicious users can analyze software programs distributed over a network and extract secret information and/or proprietary algorithms from them. Unfortunately encryption is hardly competent to solve the problem since encrypted programs must be eventually decrypted into executable forms and then adversaries can intercept them in hostile environments.

Consequently realization of software with *tamper-resistance*, which means the difficulty to read and modify the software in an unauthorized manner, becomes increasingly important. Although tamper-resistant software can be realized with the help of hardware, much attention is now being focused on *software obfuscation*, which transforms a program into a tamper-resistant form. Thus software obfuscation has been vigorously studied so far [3], [4], [7], [8], [10], [12], [16]. Unfortunately previous software obfuscation techniques share a major drawback that they do not have a theoretical basis and thus it is unclear how effective they are.

In order to mitigate such a situation, Wang et al. proposed a software obfuscation technique based on the fact that aliases in a program drastically reduce the precision of static analysis of the program [16]. However, their approach is limited to the *intraprocedural analysis* [1]. Since a program consists of many procedures[**] in general, whether or not it is obfuscated, we must conduct *interprocedural analysis* [1] in order to understand it more accurately. Moreover, interprocedural analysis usually involves intraprocedural analysis and most of software obfuscation techniques that obstruct interprocedural analysis also obstruct intraprocedural analysis. Consequently, it is desirable that an obfuscation technique is capable of obstructing interprocedural analysis. Obfuscation that hinders interprocedural analysis has another advantage. That is, since interprocedural analysis is essentially difficult to accomplish [6], [9], even a little application of such an obfuscation technique to a program can be quite effective.

Therefore we propose new software obfuscation techniques based on the difficulty of interprocedural analysis. Furthermore, we also provide a theoretical basis to the techniques. One outstanding feature of our obfuscation techniques is the introduction of function pointers [13], [18][***].

Function pointers are an indispensable tool for

Manuscript received March 23, 2002.
Manuscript revised July 7, 2002.
Final manuscript received September 20, 2002.

[†] The authors are with Japan Advanced Institute of Science and Technology, Ishikawa-ken, 923-1292 Japan.

[*] Presently, with Ministry of Land, Infrastructure and Transport.

a) E-mail: t-ogiso@jaist.ac.jp

[**] Throughout this paper we use the terms ‘procedure’ and ‘function’ interchangeably.

[***] Function pointers are a mechanism which enables indirect procedure invocations. Many of current programming languages, such as C, support this feature.

software obfuscation for the following two reasons:

1. The presence of function pointers significantly defeats static analysis, especially, interprocedural analysis. This is because the presence of procedure calls via function pointers makes it difficult to determine the control flow at compile time [13], [18]. As a result, most of conventional static analysis techniques cannot cope with function pointers and often ignore them.
2. A theoretical basis can be provided for our obfuscation techniques because the essence of them is a new complexity problem, which is, roughly speaking, the one to precisely determine the address a function pointer points to in the presence of arrays of function pointers. The problem is shown to be NP-hard in this paper. Note that although similar kinds of problems (also known as *alias problems*) have been considered and proved to be NP-hard or even undecidable [9], [14], [15], [18] so far, the complexity problem presented in this paper is appropriately adapted for software obfuscation and completely new.

In addition to use of function pointers, we propose two new obfuscation techniques to impede interprocedural analysis in this paper. They increase the number of *unrealizable paths* [9] of programs. Therefore, they drastically reduce the precision of static analysis and make the obfuscated programs significantly harder to understand.

We have already implemented a prototype tool that obfuscates C programs according to our proposed techniques and in this paper we describe the implementation and discuss the experiments results. The experimental results show that the precision of interprocedural analysis is greatly reduced and the call for graphs of obfuscated programs are made much more complicated than original ones. They imply the effectiveness of our obfuscation approaches.

The rest of the paper is structured as follows. In Sect. 2, we discuss tamper-resistant software and point out some drawbacks in previous work. In order to solve such problems, we propose new obfuscation techniques and give a theoretical basis to them in Sect. 3. In Sect. 4 we present the implementation of our obfuscation tool and show the experiments results. Finally we conclude this paper in Sect. 5.

2. Tamper-Resistant Software

In this section we shall introduce the concept of tamper-resistance and discuss various aspects of tamper-resistant software.

First of all, in this paper *tamper-resistance* means the difficulty to observe and modify an object in an illegal way. *Tamper-resistant software* is the one that has such a property and may be realized with the support of hardware device.

2.1 Software Analysis

If an adversary is trying to gather or tamper secret information in a program, first he must analyze it by some means. The major and important approach of software analysis is *static analysis* [1]. The objective of static analysis is to extract useful information from a program without running it. Generally speaking, static analysis first builds the (control) flow graph of the program and then examines the control flow or data flow of the program through the graph.

Static analysis of programs can be further classified into two types: intraprocedural analysis and interprocedural analysis. *Intraprocedural analysis* investigates semantic information within each procedure but is not concerned with the inter-relationships between procedures via procedure calls. On the other hand, *interprocedural analysis* determines semantic information of programs in consideration of legal call/return paths of procedures, on which every procedure call returns to the point where the procedure was actually called. Such paths are called *realizable paths*. The paths that are not realizable are called *unrealizable paths*.

Since a program is composed of many procedures in general, more precise understanding of the program needs to conduct interprocedural analysis. Unfortunately, the existence of unrealizable paths makes it difficult to perform interprocedural analysis [6], [9]. We will further discuss it in Sect. 3.2.2.

2.2 Realization of Tamper-Resistance

Major approaches to achieve software tamper-resistance can be divided into the following three types [4]:

1. remote execution,
2. tamper-resistance with a specialized hardware, or
3. obfuscation.

The first is tamper-resistance in remote execution. That is, a part of a program to be protected is kept in a secure trusted server and users execute the program remotely. Obvious disadvantage of this approach is performance degradation due to network communication and heavy load imposed on the server when many users access to it during a short period.

The second approach provides tamper-resistance of software by utilizing a specialized software (readers can find such a system in [17]). For instance, at least in principle, it may be possible to encrypt a program, send it to a user's host, and then decrypt and execute it in a specialized hardware on the host. Unfortunately, this approach is unreasonable from the viewpoint of hardware cost at present [10] and impedes wide applicability.

The final one is *software obfuscation*, which transforms programs into the form that is hard to understand and tamper illegally. Software obfuscation is considered to be the most viable approach to protect intellectual property rights and secret information of software in untrusted environments [4] and in this paper we study software tamper resistance by obfuscation techniques thoroughly.

2.3 Related Work

In this section, we discuss some of existing software tamper-resistance approaches.

Aucsmith addressed a threat model and design principles to develop tamper resistant software [3]. Also he discussed a method to embed a small code fragment called Integrity Verification Kernel (IVK) into a program to realize software tamper resistance.

In 1997, Mambo proposed new software obfuscation techniques in which frequency distributions of instructions in obfuscated programs are made as uniformly as possible by limiting available instructions for obfuscation [10].

Keeping application to mobile agent systems in mind, Hohl proposed the concept of 'time-limited blackbox security,' which provides tamper-resistance until a prescribed time limit in order to protect mobile agents against attacks mounted by malicious hosts [7].

Unfortunately previous obfuscation techniques share a major drawback that they are not given a theoretical basis and they often based their tamper resistance of a software upon the difficulty that human users experience when the users tamper the software. Therefore, it is still unclear how effective they are.

In order to mitigate such a situation, Wang et al. proposed a software obfuscation technique based on the fact that aliases in a program severely reduce the precision of static analysis of the program [16]. However, their approach is limited to the *intraprocedural analysis*. Since a program consists of many procedures in general, whether or not it is obfuscated, we must conduct *interprocedural analysis* in order to understand it more accurately. Moreover, interprocedural analysis usually involves intraprocedural analysis and most of software obfuscation techniques that obstruct interprocedural analysis also obstruct intraprocedural analysis. Consequently, it is desirable that an obfuscation technique is capable of obstructing interprocedural analysis. Obfuscation that hinders interprocedural analysis has another advantage. That is, since interprocedural analysis is essentially difficult to accomplish [6], [9], even a little application of such an obfuscation technique to a program can be quite effective.

3. Our Approach

From the discussions in Sect. 2.3, we shall propose new software obfuscation techniques based on the difficulty of interprocedural analysis in this section. Furthermore, we provide a theoretical basis to the techniques.

In order to achieve such goals, at first we introduce function pointers [13], [18] to our obfuscation techniques and in Sect. 3.1 we discuss how difficult the problem of determining the precise address a function pointer points to in the presence of arrays of function pointers. Note that although similar kinds of problems (also known as alias problems) have been considered and proved to be NP-hard or even undecidable [9], [14], [15], [18] so far, the complexity problem presented in Sect. 3.1 is appropriately adapted for software obfuscation and completely new.

Moreover, in addition to use of function pointers, we present two new obfuscation techniques to impede interprocedural analysis in Sect. 3.2.

3.1 On the Difficulty of Analyzing Function Pointers

In this section, in order to provide a theoretical basis for our obfuscation techniques, we show that the problem of precisely determining the address a function pointer points to in the presence of arrays of function pointers is NP-hard [5].

Theorem 1: In the presence of assignments for function pointers from arrays of function pointers and procedure calls via function pointers, where function pointers point to functions returning integers, the problem of precisely determining if there exists an execution path in a program, on which a given function pointer points to a given procedure at a point of the program is NP-hard[†].

Proof: We prove Theorem 1 by showing that 3-SAT problem [5], which is known to be NP-complete, is polynomial time reducible to the problem of Theorem 1.

Now, suppose that we are given the 3-SAT problem with the propositional variables $\{v_1, v_2, \ldots, v_m\}$ whose values are either true or false, and the formula $\wedge_{i=1}^{n}(\vee_{j=1}^{3} l_{ij})$ where l_{ij} is a literal and is either v_k or $\overline{v_k}$ for some k $(1 \leq k \leq m)$. Furthermore, each $\vee_{j=1}^{3} l_{ij}$ $(i = 1, 2, \ldots, n)$ is called a clause. Then we construct the C program in Fig. 1, the size of which is obviously polynomial of the length of the formula. Note that the condition parts of the if-statements in Fig. 1 do not matter since we have assumed that all paths are executable. Thus they are omitted but the symbol '-' is put in each if-statement instead.

In the code fragment L1, v_i $(i = 1, 2, \ldots, m)$ is declared as a function pointer and corresponds to the

[†] Here static analysis of a program is conducted under the assumption that all execution paths within procedures, without regard to interprocedural paths, are executable. This assumption is commonly found in the literature and is often called 'meet over all paths' [11]. For further backgrounds behind the way of this proof, see [14], for example.

```
int $true$() { return 1; }
int $false$() { return 0; }
main()
{
L1: int (*$fp$)(); (*$v_1$)(); (*$\overline{v_1}$)(); $\cdots$; (*$v_m$)(); (*$\overline{v_m}$)();
    int (*$A$[2])();

L2: $A[0] = false$; $A[1] = true$;

L3: if (-) { $v_1 = true$; $\overline{v_1} = false$; }
                                else { $v_1 = false$; $\overline{v_1} = true$; }
    . . .
    if (-) { $v_m = true$; $\overline{v_m} = false$; }
                                else { $v_m = false$; $\overline{v_m} = true$; }

L4: if (-) $fp = l_{1,1}$; else if (-) $fp = l_{1,2}$; else $fp = l_{1,3}$;
    if (-) $fp = A[(fp)()$&& $l_{2,1}()]$;
                      else if (-) $fp = A[(fp)()$&& $l_{2,2}()]$;
                                    else $fp = A[(fp)()$&&$l_{2,3}()]$;
    . . .
    if (-) $fp = A[(fp)()$&&$l_{n,1}()]$;
                      else if (-) $fp = A[(fp)()$&&$l_{n,2}()]$;
                                    else $fp = A[(fp)()$&&$l_{n,3}()]$;
L5:
}
```

Fig. 1 Reduction of 3-SAT to the problem in Theorem 1.

propositional variable v_i of the 3-SAT problem. Similarly, $\overline{v_i}$ is also declared as a function pointer, but it corresponds to the negation of the propositional variable v_i. Moreover, l_{ij} ($i = 1, 2, \ldots, n$, $j = 1, 2, 3$) in Fig. 1 corresponds to the j-th literal l_{ij} of i-th clause in the formula, i.e., v_k or $\overline{v_k}$ for some k ($1 \leq k \leq m$), and should be regarded as such. L2 assigns the addresses of functions $false$ and $true$ to $A[0]$ and $A[1]$, respectively.

Any execution path through if-statements in L3 corresponds to a truth value assignment of the 3-SAT problem and the converse is also true. Thus if the 3-SAT problem has a solution, then every clause has at least one literal that is true and the corresponding literal variable in Fig. 1 points to the function address $true$. Consequently we have the corresponding execution path, on which function pointer fp points to function $true$ at L5.

Furthermore, if the 3-SAT problem has no solution, there exists at least one clause whose all three literals are false. In such a case, fp does not point to function $true$, but to $false$, at L5 on any execution path.

On the other hand, now it should be clear that if function pointer fp points to function $true$ at L5 on an execution path, then the 3-SAT problem has a solution with the corresponding truth value assignment.

For the reasons stated above, the 3-SAT problem has a solution if and only if we can determine if there exists an execution path, on which function pointer fp points to function address $true$ at L5. This completes the proof. □

3.2 Proposed Obfuscation Techniques

Theorem 1 in Sect. 3.1 means that the complexity is NP-hard to conduct precise interprocedural analysis on programs that have assignments for function pointers from arrays of function pointers and procedure calls via function pointers. Thus, this fact gives a theoretical basis to software obfuscation with such techniques.

Based on this discussion, in this section we propose software obfuscation techniques that transform programs into the forms described above. First, we present how function pointers are used in our proposed approach. Then we additionally propose two new obfuscation techniques to significantly reduce the precision of interprocedural analysis by increasing the number of *unrealizable paths* of programs.

3.2.1 Use of Function Pointers for Software Obfuscation

In particular, one of useful obfuscation techniques that can be used along with function pointers are arrays. Arrays have essentially the same semantics as pointers and computation of indices of array variables is difficult for the similar reason as in the case of pointers [1]. Therefore in order to obstruct static analysis, we find it useful to store function addresses or pointers in arrays and to make procedure calls via the arrays and function pointers. This is why they are used in Theorem 1.

Our obfuscation procedures with respect to function pointers are given below. They consist of three phases, i.e., (1) *Decomposition of procedures*, (2) *Use of function pointers*, and (3) *Introduction of arrays of function pointers*. Below, the procedures are concisely described because of space limitation, although, it should be noted that they roughly correspond to the algorithm that was implemented in our prototype obfuscation tool discussed in Sect. 4.1. Also notice that although the example programs below that result from obfuscation are intentionally not so obfuscated for the purpose of explanation, it is not difficult to transform a program into any more obfuscated form, as our obfuscation tool does. Moreover, the NP-hardness result of Theorem 1 means that the complexity of interprocedural analysis of the obfuscated programs is expected to be exponential of the program sizes. Therefore we can hardly expect the precise analysis of the programs.

Now we are ready to present our obfuscation procedures with respect to function pointers.

(1) Decomposition of procedures

At first we randomly pick a procedure, decomposes it into smaller procedures, and reconstruct the original procedure with the decomposed ones while maintaining the original semantics. The fundamental principle of decomposition of procedures is to randomly

```
func() {
  int a, b;
  ...
  if (a > b)
    a = b;
  ...
  b = a+1;
  ...
}
```

```
int a, b;
...
func1() { a = b; }
func2() { b = a+1; }
func() {
  ...
  if (a > b)
    func1();
  ...
  func2();
  ...
}
```

Fig. 2 Decomposition of procedures.

choose some consecutive statements and to organize them into a new procedure. In the simplest case, first we randomly select a procedure that does not have global branches (e.g., procedure calls, goto-statements, return-statements, and so forth). Such a procedure may have if-statements or for-statements since they are local branches within the procedure. Then we further choose a sequence of statements in the procedure and form them into a new procedure. The newly created procedure must reflect its computation on the original procedure. In the simplest case above, it can be done by letting these procedures sharing global variables, which have been promoted from local variables within the original procedure. This process is illustrated in Fig. 2.

The step above is repeated at random on multiple procedures in the program. At this stage we might insert dummy functions into the program. Thus the numbers of nodes and edges of the control flow graph and the call graph become larger and as a result the technique makes interprocedural analysis of the program more difficult.

(2) Use of function pointers

A set of procedures and decomposed ones randomly chosen are now forced to be called via function pointers. For instance, the program at the right hand side of Fig. 2 might be transformed into the one in Fig. 3. As drawn in Fig. 3, two new if-statements have been newly introduced. Note that since the values of the condition expressions `a*(a+1)%2` and `(b-2)*(b-1)*b%6` always equal to zero regardless of the values of `a` and `b` respectively, the semantics of the original program is maintained. However, generally speaking, in static analysis it is very difficult to evaluate such expressions and this results in a difficulty in determining the execution paths in the presence of if-statements[†]. Needless to say, such condition expressions can be made arbitrarily complicated as long as the original semantics is retained. Therefore the if-statements make it difficult to determine the function addresses that `fp` points to.

```
int a, b; int (*fp)();
...
func1() { a = b; }
func2() { b = a+1; }
func() { ...
  if (a > b) {
    if (a*(a+1)%2 == 0)
      fp = func1;
    else
      fp = func2;
    ...
    (fp)();
  }
  ...
  if ((b-2)*(b-1)*b%6 != 0)
    fp = func1;
  else
    fp = func2;
  ... (fp)(); ...
}
```

Fig. 3 Use of function pointers.

(3) Introduction of arrays of function pointers

Here we introduce arrays of function pointers. Then we store function addresses at random into the arrays and prepare expressions to compute each index using some (direct or indirect) function calls. Now some of procedure calls are replaced with the calls via the arrays. For example, one possible program into which the program in Fig. 3 are converted is presented in Fig. 4. There the array `A` of function pointers, and function `func0` that helps index calculation of `A` are provided. In this case, note that `func0` always returns an even integer irrespective of `a` and thus the semantics of the original program is maintained. Various methods of obfuscating index computation are also possible. For example, we may embed the address of i-th function in the $((i \times k) \bmod p)$-th element of an array, where p is a prime, $1 \leq i \leq p-1$, and k is a constant randomly chosen in the range $1 \leq k \leq (p-1)$. Hence each value of $ik \bmod p$ is different and scattered over the array. Some of others are also implemented in our prototype tool.

Now assignments to `fp` depend on the function call via (previous value of) `fp` and the corresponding element of `A`. Combination of them significantly defeat static analysis as discussed so far.

3.2.2 Obfuscation to Increase the Number of Unrealizable Paths

One of the reasons why interprocedural analysis is difficult is that it must follow the execution paths that are realizable, as stated in Sect. 2.1. Based on this fact, in this section we propose two novel software obfuscation techniques to hinder interprocedural analysis: *Mergence of procedure calls into one call* and *Additions*

[†] This leads to the 'meet over all paths' assumption as stated in Theorem 1. Note that this assumption works fine for intraprocedural analysis, but cannot necessarily cope with some interprocedural analysis in the face of unrealizable paths. This will be further discussed in Sect. 3.2.2.

of redundant return-statements. The fundamental idea of these two techniques is to increase the number of unrealizable paths of programs. The techniques fundamentally reduce the precision of static analysis and make the obfuscated programs harder to read.

Notice that they are general obfuscation techniques and are not necessarily used combinedly with function pointers, although they are in our obduction tool.

(1) Merge procedure calls into one call

This technique accommodates multiple procedure calls into a newly created procedure. More detailed process of this is as follows. First we randomly select multiple procedure calls $f_{i_1}(...)$, ..., $f_{i_n}(...)$ in a procedure, say, $func$, where f_{i_1}, ..., f_{i_n} have the same type of return values. Next we create a new procedure (let us call this procedure f_{n+1}) and include $f_{i_1}(...)$, ..., $f_{i_n}(...)$ in f_{n+1}. At that time, we introduce a position variable (e.g., sw) to remember positions where f_{i_1}, ..., f_{i_n} are called in $func$. Finally we replace $f_{i_1}(...)$, ..., $f_{i_n}(...)$ in $func$ with the calls to $func_{n+1}$. If some of f_{i_1}, ..., f_{i_n} need parameters, they can be passed via parameters of $func_{n+1}$.

For example, consider a trivial case of the transformation above. This is shown in Fig. 5. As illustrated in the figure, two procedure calls `func1()` and `func2()` are selected at random, procedure `func3()` is newly created, and finally the two calls are embedded into `func3()` in some obfuscated fashion. After such obfuscation is performed, the call graph changes as depicted in Fig. 6.

Now look at the call graph more carefully. As Fig. 6

```
int a, b; int (*fp)(); int (*A[10])(); ...
func0() { return ((a-1)*a); }
func1() { a = b; }
func2() { b = a+1; }
func() { ...
  A[0] = A[1] = func0; A[2] = func1;
  A[3] = func2;
  A[4] = A[6] = func0; /* dummy */
  A[5] = A[9] = func1; /* dummy */
  A[7] = A[8] = func2; /* dummy */
  ...
  fp = A[(func0()%2)*a*b]; ...
  if (a > b) {
    if (a*(a+1)%2 == 0)
      fp = A[((fp)()%2)+2];
    else
      fp = A[((fp)()%2)+4];
    ...
    (fp)(); ...
  }
  ... fp = A[b&1]; ...
  if ((b-2)*(b-1)*b%6 != 0)
    fp = A[((fp)()%2)+5];
  else
    fp = A[((fp)()%2)+3];
  ...
  (fp)(); ...
}
```

Fig. 4 Introduction of arrays of function pointers.

```
func1() { ... }
func2() { ... }

func() {
  ...
  func1();
  ...
  func2();
  ...
}
```

```
int sw;
func1() { ... }
func2() { ... }
func3() { ...
  switch (sw) {
  case 0: func1(); break;
  case 1: func2(); break;
  ...
  }
  ...
}

func() { ...
  sw = (sw-1)*sw%2; ...
  func3(); ...
  sw = sw*sw*(sw+1)*(sw+1)%4+1; ...
  func3(); ...
}
```

Fig. 5 Merge procedure calls into one call.

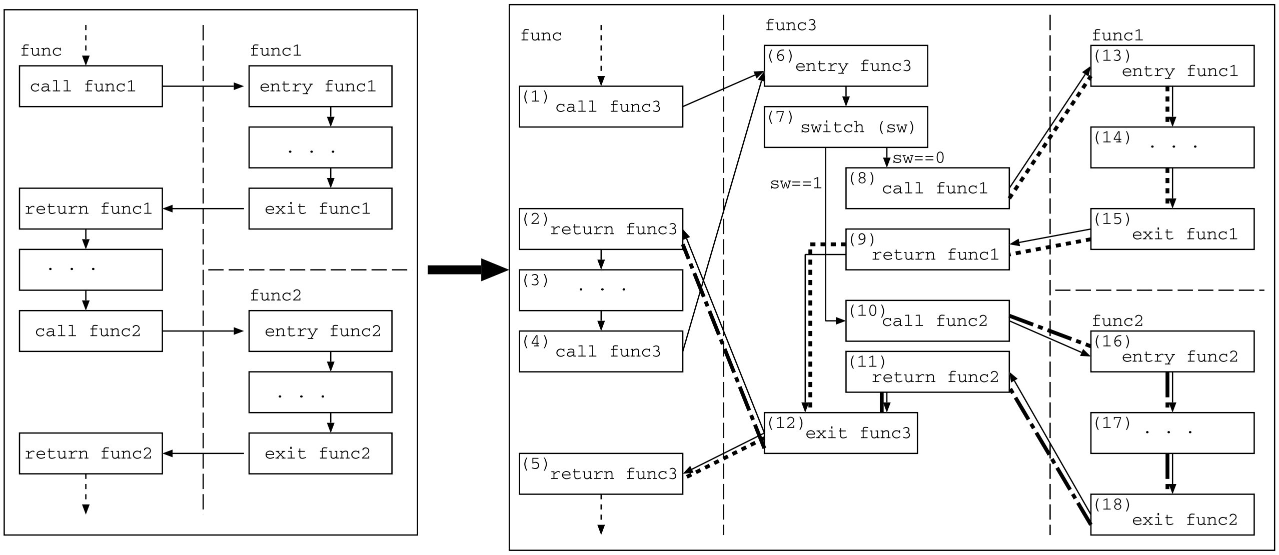
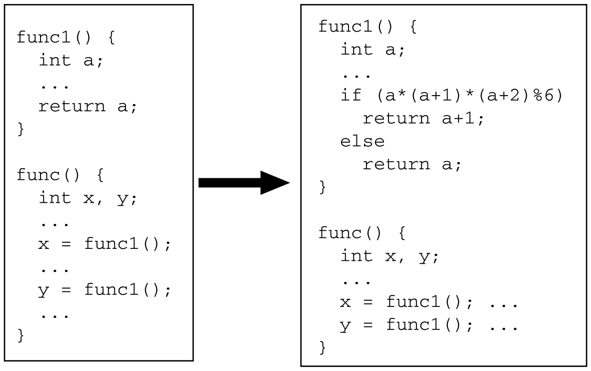


Fig. 6 Call graph change after procedure calls merged.

shows, it is straightforward to follow the execution path on the call graph before transformed. On the other hand, the call graph after transformed (at the right hand side of Fig. 6) has now two unrealizable paths, namely, one is '··· → [(15) `exit func1`] → [(9) `return func1`] → [(12) `exit func3`] → [(5) `return func3`] → ···,' and the other is '··· → [(18) `exit func2`] → [(11) `return func2`] → [(12) `exit func3`] → [(2) `return func3`] → ···.' If interprocedural analysis ignores the unrealizable paths, it only fails or otherwise yields imprecise analysis results [9]. However, even if interprocedural analysis tries to follow the realizable paths, it becomes more difficult as the size of the program becomes larger or our obfuscation techniques are applied to the program more and more times.

(2) Additions of redundant return-statements

Another obfuscation technique here can also complicates the call graph and hinder interprocedural analysis. This is done by adding redundant return-statements. For example, see Fig. 7. The call graph change due to the obfuscation is drawn in Fig. 8. It is not hard to see that the call graph becomes more complicated and the number of unrealizable paths increases from two to four. Thus the same discussion in Sect. (1) also applies to the obfuscation technique in this section and demonstrates its validity.



```
func1() {
  int a;
  ...
  return a;
}

func() {
  int x, y;
  ...
  x = func1();
  ...
  y = func1();
  ...
}
```

```
func1() {
  int a;
  ...
  if (a*(a+1)*(a+2)%6)
    return a+1;
  else
    return a;
}

func() {
  int x, y;
  ...
  x = func1(); ...
  y = func1(); ...
}
```

Fig. 7 Additions of redundant return-statements.

3.3 Example of Obfuscation

At the end of Sect. 3, for completeness of the description of this section, we show in Fig. 9 an example of an obfuscated program to which all obfuscation techniques are applied.

4. Prototype Implementation and Experiments

This section describes our implementation of the proposed obfuscation techniques and presents experiments results.

4.1 Prototype Obfuscation Tool

We have implemented an obfuscation tool based on our proposed technique with SUIF [2]. SUIF is a compiler infrastructure system being developed at Stanford University, and enables us to manipulate programs in SUIF's intermediate representation called IR. Furthermore, SUIF provides various transformation utilities between IR and various programming languages, and also has a lot of support packages. We used one of the packages to conduct interprocedural analysis of programs. A main part of the structure of our obfuscation tool is depicted in Fig. 10.

Our obfuscation tool is written in C++ and the main module of it is `ObfuscatePass` class, which is a subclass of `Pass` of SUIF2 system. `Pass` class corresponds to one compiler pass and analyses programs transformed into IR. An outline of `ObfuscatePass` class is given in Fig. 11.

Now our obfuscation tool works roughly as follows. The obfuscation tool first reads a tar-

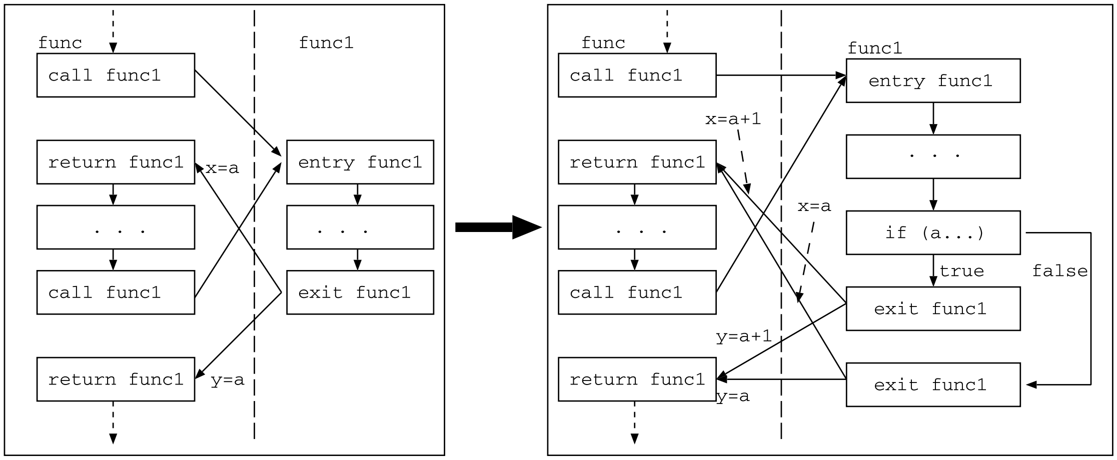


Fig. 8 Call graph change after return-statements added.

```
int a, b, c; int (*fp1)(), (*fp2)(); int (*A[10])(); ...
func0() { ... if (a*(a+1)*(a+2)%6) return (a*(a+2)); else return ((a-1)*a); }
func1() { a = b; }
func2() { b = a+1; }
func3() { ...
  switch (c) {
  case 0: fp1 = func1; break;
  case 1: fp1 = func2; break;
  ...
  }
  ... (fp1)(); ...
}
func() { ...
  A[0] = A[1] = func0; A[2] = A[3] = func3;
  A[4] = A[6] = func0; /* dummy */ A[5] = A[9] = func1; /* dummy */
  A[7] = A[8] = func2; /* dummy */
  ...
  fp2 = A[(func0()%2)*a*b]; ...
  if (a > b) {
    if (a*(a+1)%2 == 0) fp2 = A[((fp2)()%2)+2]; else fp2 = A[((fp2)()%2)+4];
    ... c = (c-1)*c%2; ...
    (fp2)(); ...
  }
  ... fp2 = A[b&1]; ...
  if ((b-2)*(b-1)*b%6 != 0) fp2 = A[((fp2)()%2)+5]; else fp2 = A[((fp2)()%2)+3];
  ... c = c*c*(c+1)*(c+1)%4+1; ...
  (fp2)(); ...
}
```

Fig. 9 Example of obfuscation.

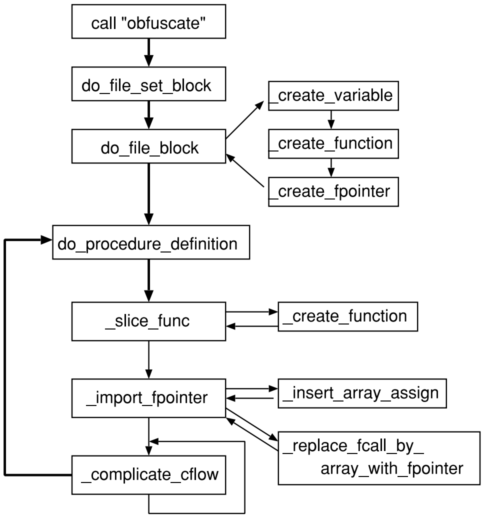


Fig. 10 Prototype obfuscation tool.

get program and transform it into IR representation with the help of SUIF2. Then two methods of ObfuscatePass class, namely, do_file_set_block and do_file_block, are executed in turn for initialization. do_file_set_block initialize the global data structures and do_file_block invokes methods _create_variable, _create_function, and _create_fpointer in order to generate auxiliary variables, dummy functions, function pointers and arrays of function pointers.

After the initialization phase, the tool obfuscates the target program via the call to do_procedure_definition, possibly multiple times.

```
class ObfuscatePass : public Pass {
public:
  ObfuscatePass(SuifEnv *enc, const LString &name)
                                   : Pass(env, name) {}
  virtual ~ObfuscatePass(void) {}
  void do_procedure_definition
              (ProcedureDefinition *proc_def) {
    _slice_func(proc_def);
    _import_fpointer(proc_def);
    _complicate_cflow(proc_def);
  }
protected:
  void _complicate_cflow
                  (ProcedureDefinition *proc_def) {
    ...
    list<Statement*> *slist=
                collect_objects<Statment>(proc_def)
    ...
    Expression *alws_true_cond =
                        _build_opaque_expression();
    ...
    else_part->append_statement(dummy_call);
  }
  ...
}
extern "C" void init_obfuscater(SuifEnv *suif_env) {
  ModuleSubSystem *ms =
                  suif_env->get_module_subsystem();
  ms->register_module(
    new ObfustatePass(suif_env, "obfuscate"))
}
```

Fig. 11 ObfuscatePass class.

do_procedure_definition method transforms procedures selected at random into obfuscated ones, by calling _slice_func (decomposition of procedures), _import_fpointer (introduction of function pointers and arrays), and _complicate_cflow (complicate the structure of the call graph).

4.2 Experiments

In this section we present application of our obfuscation tool to six programs, that is, RC6, MD5, jpeg2ps, Camellia, FFT, and coretest.

Table 1 shows the differences between the control flow graphs of the original programs and those of the obfuscated programs. We can readily see from the table that there exist increases of about 2.17 times in the number of nodes and about 2.22 times in the number of edges on the average. The increase of the numbers of nodes and edges of control flow graphs of programs significantly obstruct analysis of software programs, especially interprocedural analysis [1], [9]. Thus control flow and data flow analysis become harder. Furthermore, as discussed in Sect. (1) and Sect. (2), we can expect that these results immediately lead to the difficulty of interprocedural analysis.

Now turn to Table 2. The table indicates the changes of the numbers and the types of procedure calls after obfuscation. In Table 2, ‘All Call Sites’ and ‘Direct Call Sites’ represent the numbers of all procedure calls and direct calls via procedure names, respectively. Furthermore, in the table ‘Indirect Calls’ and ‘Indirect Call Targets’ mean the numbers of indirect procedure calls via function pointers and possible target addresses function pointers point to, respectively.

As shown in Table 2, all procedure calls of all programs before obfuscation are direct calls. On the other hand, after obfuscation we have many indirect calls. Intuitively speaking, this directly means the difficulty of interprocedural analysis.

More noteworthy in Table 2 is the number of the possible target addresses. The table shows that on the average we have 23.8 candidate addresses a function pointer points to per indirect call in the obfuscated programs. In particular FFT has 40 candidates on the average. These result give a good evidence that precision of interprocedural analysis is drastically reduced by our obfuscation techniques.

We have evaluated performance degradation due to the obfuscation, as indicated in Table 3 and Table 4. The experiments were conducted on a Sun Ultra 5 (UltraSPARC-II 400MHz) with Solaris 8 (SunOS 5.8). Programs were compiled by gcc 2.8.1 with no optimization option and with optimization option ‘-O2.’ Each execution time was the average of 10000 times execution. The average rate of execution times of obfuscated programs over original programs is 1.4 in non-optimized versions, on the other hand, the rate becomes 1.93 in optimized versions. Obfuscation interferes with optimization in nature, thus the difference of execution times of the original programs and the obfuscated ones would become larger if the programs were optimized versions.

Finally we show the change of the program sizes before and after obfuscation in Table 5 and Table 6. Similar discussion as in the case of execution time also holds here.

Table 1 Change of the control flow graph.

	Before Obfuscation		After Obfuscation	
program	#nodes	#edges	#nodes	#edges
RC6	143	146	464	488
MD5	684	684	1331	1353
jpeg2ps	965	1069	1728	1866
Camellia	617	597	1297	1356
FFT	1741	1817	2895	3040
coretest	205	212	470	485

Table 3 Change of execution time (non-optimized).

program	Before Obfuscation [sec]	After Obfuscation [sec]
RC6	0.21	0.27
MD5	0.78	1.51
jpeg2ps	0.23	0.23
Camellia	0.25	0.50
FFT	0.20	0.24
coretest	0.37	0.38

Table 4 Change of execution time (optimized).

program	Before Obfuscation [sec]	After Obfuscation [sec]
RC6	0.13	0.17
MD5	0.26	0.67
jpeg2ps	0.17	0.17
Camellia	0.07	0.30
FFT	0.08	0.11
coretest	0.37	0.37

Table 2 Change of procedure calls.

	Before Obfuscation				After Obfuscation			
program	Direct Call Sites	All Call Sites	Indirect Call Targets	Indirect Calls	Direct Call Sites	All Call Sites	Indirect Call Targets	Indirect Calls
RC6	0	0	0	0	2	41	351	39
MD5	11	11	0	0	15	95	2400	80
jpeg2ps	141	141	0	0	146	214	1904	68
Camellia	68	68	0	0	77	161	2016	84
FFT	75	75	0	0	87	227	5600	140
coretest	46	46	0	0	48	80	384	32

Table 5 Change of program size (non-optimized).

program	Before Obfuscation [sec]	After Obfuscation [sec]
RC6	9420	15696
MD5	18740	36824
jpeg2ps	24028	48988
Camellia	16744	31560
FFT	48068	92476
coretest	9540	16384

Table 6 Change of program size (optimized).

program	Before Obfuscation [sec]	After Obfuscation [sec]
RC6	8200	12168
MD5	13960	22396
jpeg2ps	19380	38260
Camellia	12280	38260
FFT	22228	48804
coretest	8892	13456

4.3 Some Remarks on Automatic Deobfuscation Tool

So far we have described our prototype implementation and the experimental results. In this section, we discuss automatic backword procedures to cancel obfuscation. Such an operation is often called *deobfuscation* [4].

Implementation of obfuscation is a relatively complicated task, but it is really implementable. On the other hand, deobfuscation is difficult to carry out and hence it is hard to implement. The reason is given below.

In general, once syntax analysis has finished, program transformation is possible without static analysis[†]. Obfuscation is a kind of (complicated) program transformation and does not need static analysis (i.e., data flow or control flow analysis). Therefore it is implementable in principle. Most of complicated tasks of obfuscation lies in maintaining the semantics of the original program.

On the other hand, deobfuscation is also a kind of program transformation, although, it is by no means an easy operation. This is because in deobfuscation procedures, it is not sufficient to transform a program into another form while maintaining the original semantics. In addition, deobfuscation must transform the obfuscated program into more *intelligible* form. Thus, for example, in deobfuscation it may be necessary to remove dummy functions introduced in obfuscation, investigate the obfuscated execution paths and recover the original paths from those, etc. (Hence deobfuscation is similar to optimization in a sense [4].) Therefore static analysis is mandatory in deobfuscation process. However, our obfuscation techniques make it difficult in various ways to conduct such static analysis, as thoroughly discussed in our paper.

In summary, software obfuscation is one-way transformation in nature.

5. Conclusion

Software obfuscation is promising to protect intellectual property rights and secret information of software in untrusted environments. Therefore we have proposed new software obfuscation techniques in this paper. The techniques are based on the difficulty of interprocedural analysis of software programs. The essence of our obfuscation techniques is a new computational complexity problem, which is, roughly speaking, the one to precisely determine the address a function pointer points to in the presence of arrays of function pointers. We have shown that the problem is NP-hard and the fact provides a theoretical basis for our obfuscation techniques. Furthermore, we have already implemented a prototype tool which obfuscates C programs according to our proposed techniques and in this paper we describe the implementation and discuss the experiments results by means of our obfuscation tool. The experimental results show that the precision of interprocedural analysis is greatly reduced and the call for graphs of obfuscated programs are made much more complicated than original ones. They implies the effectiveness of our obfuscation approaches.

Acknowledgment

The authors would like to thank the anonymous referees for valuable and helpful comments on this paper.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, 1986.

[2] G. Aigner, A. Diwan, D.L. Heine, M.S. Lam, D.L. Moore, B.R. Murphy, and C. Sapuntzakis, An overview of the SUIF2 compiler infrastructure, Computer Systems Laboratory, Stanford University, http://suif.stanford.edu/.

[3] D. Aucsmith, “Tamper resistant software: An implementation,” in R.J. Anderson ed., Information Hiding: First International Workshop, Lecture Notes in Computer Science, vol.1174, pp.317–333, Springer-Verlag, 1996.

[4] C. Collberg, C. Thomborson, and D. Low, A taxonomy of obfuscating transformations, Technical Report 148, Department of Computer Science, the University of Auckland, Auckland, New Zealand, 1997.

[5] M.R. Garey and D.S. Johnson, Computers and Intractability—A Guide to the Theory of NP-completeness, W.H. Freeman and Co., 1979.

[†] Of course since we have not conducted static analysis yet, we cannot understand the behaviour of the program (for instance, we cannot know execution paths of the program, or use-def relationships of data in the program, and so forth, at this stage). In spite of that, we can transform a program into an equivalent one. It is easy to understand such a situation if you imagine program transformation of a source code in a high-level programming language into a binary code.

[6] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," ACM Trans. Prog. Lang. Syst., vol.21, no.4, pp.848–894, 1999.

[7] F. Hohl, "Time limited blackbox security: Protecting mobile agents from malicious hosts," in G. Vigna ed., Mobile Agents Security, Lecture Notes in Computer Science, vol.1419, pp.92–113, Springer-Verlag, 1998.

[8] T. Iwai, K. Kuriyama, W. Wu, and F. Mizoguchi, "A proposal for tamper-resistant mobile agents," Computer Security Symposium (CSS99), pp.43–48, Oct. 1999.

[9] W.A. Landi, Interprocedural aliasing in the presence of pointers, PhD thesis, Rutgers University, New Brunswick, NJ, Jan. 1992.

[10] M. Mambo, T. Murayama, and E. Okamoto, "A tentative approach to constructing tamper-resistant software," New Security Paradigm Workshop, pp.23–33, Sept. 1997.

[11] T.J. Marlowe and B.G. Ryder, "Properties of data flow frameworks: A unified model," Acta Inf., vol.28, no.2, pp.121–163, 1990.

[12] M. Misawa, K. Akai, and T. Matsumoto, "Evaluation of obfuscator by searching runtime data," Symposium on Cryptography and Information Security (SCIS 2002), pp.365–370, Jan. 2002.

[13] R. Muth and S. Debray, On the complexity of function pointer may-alias analysis, Technical Report TR96-18, Department of Computer Science, University of Arizona, Oct. 1996.

[14] E.W. Myers, "A precise inter-procedural data flow algorithm," Conference record of the 8th ACM Symposium on Principles of Programming Languages (POPL), pp.219–230, 1981.

[15] G. Ramalingam, "The undecidability of aliasing," ACM Trans. Prog. Lang. Syst., vol.16, no.6, pp.1467–1471, Nov. 1994.

[16] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software tamper resistance: Obstructing static analysis of programs," Technical Report CS-2000-12, Department of Computer Science, University of Virginia, Dec. 2000.

[17] U.G. Wilhelm, S. Staamann, and L. Buttyán, On the problem of trust in mobile agent systems, Symposium on Network and Distributed System Security, Internet Society, March 1998.

[18] S. Zhang and B. Ryder, Complexity of single level function pointer aliasing analysis, Technical Report LCSR-TR-233, Laboratory of Computer Science Research, Rutgers University, Oct. 1994.

Toshio Ogiso received the B.E. from National Institution for Academic Degrees (NIAD) and M.info.Sc. from Japan Advanced Institute of Science and Technology (JAIST) in 2002. He has joined Ministry of Land, Infrastructure and Transport since 2002 and engaged in control of Electricity and Telecommunication System.

Yusuke Sakabe received the B.E. of Engineering from the Nagoya Institute of Technology in 2000. He is currently a student of master course of the Japan Advanced Institute of Science and Technology. His research interests include tamper-resistant softwares.

Masakazu Soshi received his B.E. and M.S. degrees from University of Tokyo, in 1991 and in 1993 respectively, and his Ph.D. degree from University of Electro-Communications in 1999. He worked as an associate for University of Electro-Communications from 1997 to 1998. Since 1999, He has been an associate of Japan Advanced Institute of Science and Technology (JAIST). His research interests include theoretical analysis of access matrix models, anonymous communication, and development of security architectures in general.

Atsuko Miyaji received the B.Sc., the M.Sc., and Dr. Sci. degrees in mathematics from Osaka University, Osaka, Japan in 1988, 1990, and 1997 respectively. She joined Matsushita Electric Industrial Co., LTD from 1990 to 1998 and engaged in research and development for secure communication. She has been an associate professor at JAIST (Japan Advanced Institute of Science and Technology) since 1998. She has joined the computer science department of University of California, Davis since 2002. Her research interests include the application of projective varieties theory into cryptography and information security. She received IPSJ Sakai Special Researcher Award in 2002. She is a member of the International Association for Cryptologic Research, the Institute of Electronics, Information and Communication Engineers and the Information Processing Society of Japan.