Software Tamper Resistance Based on the Difficulty of Interprocedural Analysis

Toshio Ogiso*, Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji

School of Information Science, Japan Advanced Institute of Science and Technology
1-1 Asahidai, Tatsunokuchi-machi, Nomi-gun, Ishikawa 923-1292, JAPAN
E-mail: {t-ogiso,y-sakabe,soshi,miyaji}@jaist.ac.jp

Abstract. Software obfuscation is a promising approach for protection of intellectual property rights of software in untrusted environments. Unfortunately most of previous obfuscation techniques do not have a theoretical basis and thus it is unclear how effective they are. Therefore in this paper we propose new software obfuscation techniques, which are based on the difficulty of interprocedural analysis. 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 obfuscation tool. In this paper we also describe the implementation and discuss the experiments results.

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.

In such situations, malicious users can analyze software programs distributed over a network and can 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, 6, 8, 10, 14]. Unfortunately most of previous software obfuscation techniques do not have a theoretical basis and hence it is unclear how effective they are.

* The author is currently with Ministry of Land, Infrastructure and Transport.

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 [14]. However, their approach is limited to the *intraprocedural analysis* [1]. Since a program consists of many procedures[1] 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 [7, 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 [15][2].

Function pointers are a powerful tool for 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 [15].
2. A theoretical basis can be provided for our obfuscation techniques because the essence of them is a new complexity problem 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 have been proved to be NP-hard so far [9, 12, 15], our complexity problem is appropriately adapted for software obfuscation and completely new.
   The NP-hardness of interprocedural analysis of programs obfuscated by our techniques means that the complexity of interprocedural analysis is expected to be exponential of the program sizes and is almost intractable. Therefore we can hardly expect the precise analysis of the programs.

In addition to use of function pointers, we propose two novel obfuscation techniques (of course they are not found in [4]) to impede interprocedural analysis in this paper. The fundamental idea of the techniques is to increase the number of

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

[2] Collberg et al. briefly mentioned interprocedural obfuscation in the presence of pointers [4]. However, their technique does not use function pointers, but aggregate data structures (e.g., `struct` data type in C) and (ordinary) pointers. In addition, since their technique mainly focuses on the difficulty to resolve dynamic data structures, it is not obvious whether or not it can be used to obfuscate arbitrary programs. Furthermore, a theoretical basis for the technique is not provided in their work.

unrealizable paths [9] of programs. Therefore, they reduce the precision of static analysis and make the obfuscated programs harder to understand.

We have already implemented a prototype tool for our software obfscation 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 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 shall 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 Related Work

In this section, we discuss existing software tamper-resistance approaches. Due to space constraints, we shall mention only representative work.

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 in a prescribed time limit in order to protect mobile agents against attacks mounted by malicious hosts [8].

Unfortunately previous obfuscation techniques share a major drawback that they do not have a theoretical basis and they often based their tamper resistance of software upon the difficulty that human users experience when the users try to tamper the software. Therefore, it is still unclear how effective the approaches 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 [14]. However, their approach is limited to the *intraprocedural analysis*[3]. Since a program consists of many procedures in general,

[3] Wang addressed interprocedural software obfuscation in the subsequent work [13]. The technique uses function pointers, which is in common with ours. However, as compared with our approaches, Wang's approach has the following three drawbacks: (i) Approaches other than function pointers to explicitly impede interprocedural analysis are not considered, (ii) Wang does not theoretically analyze the complexity of interprocedural analysis in the presence of function pointers, and finally (iii) Although obfuscation using function pointers was implemented, it is not evaluated

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 [7, 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, 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.

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]. Note that although similar kinds of problems have been considered and proved to be NP-hard so far [9, 12, 15], the complexity problem presented in this section is appropriately adapted for software obfuscation and completely new.

Now we are ready to consider Theorem 1 defined below:

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 sketch: 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.

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 [12], for example.

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

with respect to function pointers in Wang's empirical study and thus its effectiveness is not clear.

the C program in Fig. 1, the size of which is obviously polynomial of the length of the formula.

```
int true() { return 1; }
int false() { return 0; }
main() {
L1:  int (*fp)(); (*v_1)(); (*~v_1)(); ···; (*v_m)(); (*~v_m)();
     int (*A[2])();

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

L3:  if (-) { v_1 = true; ~v_1 = false; } else { v_1 = false; ~v_1 = true; }
     ...
     if (-) { v_m = true; ~v_m = false; } else { v_m = false; ~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 problem to the problem in Theorem 1

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

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, fp does not point to function *true*, but to *false*, at L5 on any execution path.

Now it should be clear that 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.

Use of Function Pointers for Software Obfuscation Our software obfuscation technique uses function pointers and this is a great deviation from previous work. Function pointers are used combinedly with other obfuscation techniques discussed later in this section.

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 in a position 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 (Fig. 2). This step is repeatedly done 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.

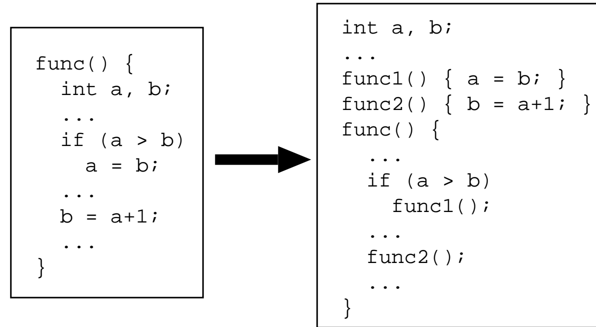

Fig. 2. Decomposition of procedures

2. Use of function pointers
   A set of procedures and decomposed ones randomly chosen are now forced to be called via function pointers. For example, 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 determine the execution paths in the presence of if-statements[4]. 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.
3. Introduction of arrays of function pointers
   Some of procedure calls are replaced with the calls via arrays of function pointers. 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. Of course a more elaborated manner of index computation may be also possible (actually some others are 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.

[4] 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 later.

```
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

```
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

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, on which every procedure call returns to the point where the procedure was actually called [9]. Such paths are called *realizable paths*. The paths that are not realizable are called *unrealizable paths*. The difficulty of interprocedural analysis to exactly follow the realizable paths increases drastically as the size of the program becomes larger or the call graph becomes more complicated. On the other hand, if interprocedural analysis ignores the unrealizable paths, it only fails or otherwise yields imprecise analysis results [7, 9].

Based on this discussion, in this section we propose two novel software obfuscation techniques to hinder interprocedural analysis: *Mergence of procedure calls into one call* and *Additions 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.

Merge Procedure Calls into One Call This technique randomly selects multiple procedure calls, generate a new procedure, and finally accommodates the selected procedure calls into the new procedure.

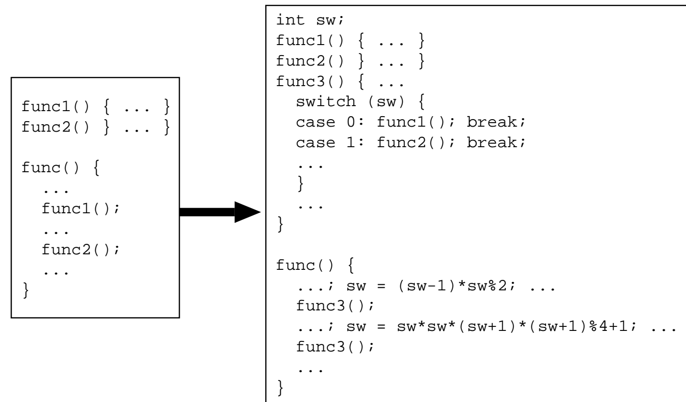

Fig. 5. Merge procedure calls into one call

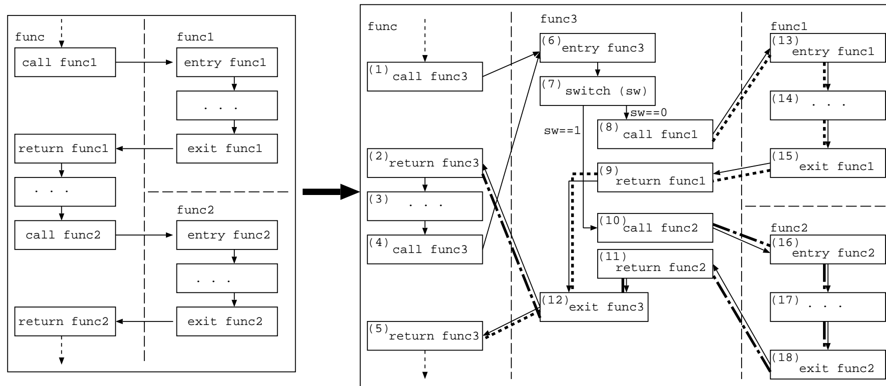

Fig. 6. Call graph change after procedure calls merged

For example, consider the transformation 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 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 '$\cdots \rightarrow$ [(15) `exit func1`] $\rightarrow$ [(9) `return func1`] $\rightarrow$ [(12) `exit func3`] $\rightarrow$ [(5) `return func3`] $\rightarrow \cdots$', and the other is '$\cdots \rightarrow$ [(18) `exit func2`] $\rightarrow$ [(11) `return func2`] $\rightarrow$ [(12) `exit func3`] $\rightarrow$ [(2) `return func3`] $\rightarrow \cdots$'. Thus this obfuscation technique complicates the structure of the call graph and increases the uncertainty of possible realizable paths on the graph, which results in the obstruction of interprocedural analysis.

Additions of Redundant Return-Statements Another obfuscation technique here can also complicate 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. Hence the same discussion in the previous technique also applies to the obfuscation technique here and demonstrates its validity.

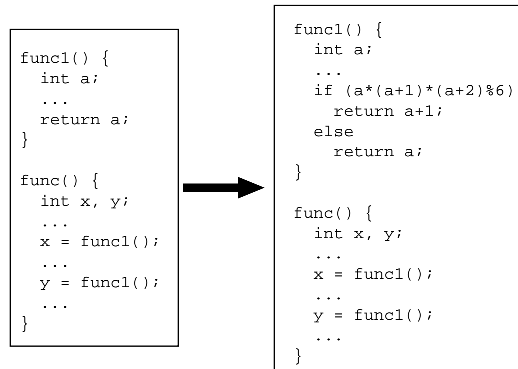

Fig. 7. Additions of redundant return-statements

4 Prototype Implementation and Experiments

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

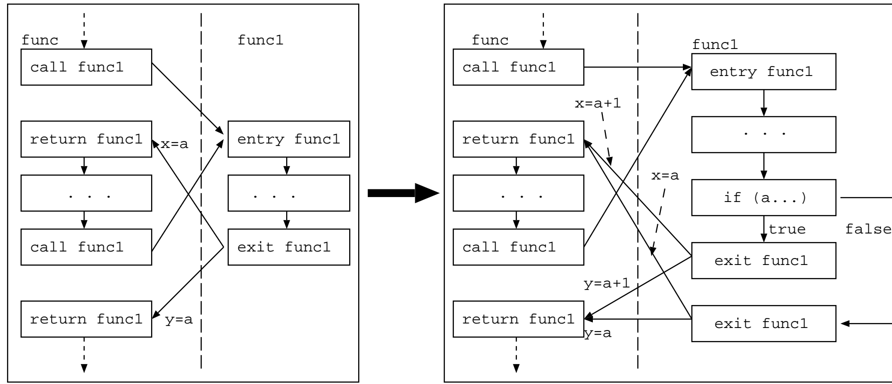

Fig. 8. Call graph change after return-statements added

4.1 Prototype Obfuscation Tool

We have implemented a prototype obfuscation tool based on our proposed technique with SUIF [2]. SUIF is an excellent 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. 9.

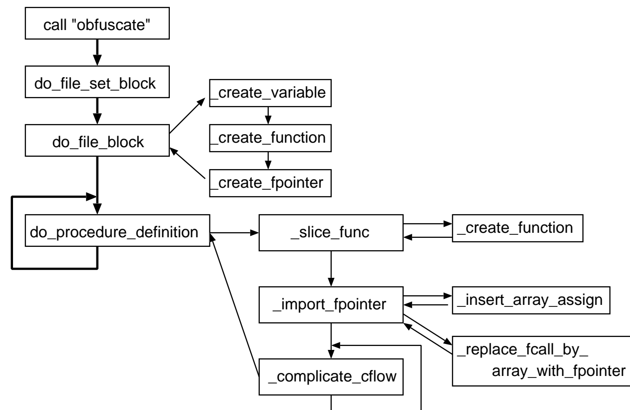

Fig. 9. Basic structure of our obfuscation tool

Now our obfuscation tool works roughly as follows. The obfuscation tool first reads a target 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. `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 severely obstruct analysis of software programs, especially interprocedural analysis [1, 9]. Thus control flow and data flow analysis become harder. Furthermore, as discussed in Sect. 3.2 and Sect. 3.2, we can expect that these results immediately lead to the difficulty of interprocedural analysis. Complication of the control flow graph also results in the difficulty of program readability [6].

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

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. Notice that it can be said from the discussions of Theorem 1 that the complexity of interprocedural analysis is expected to be exponential of the program size. Therefore these results give a good evidence that we can hardly expect precise interprocedural analysis of the obfuscated programs. It is also well-known that lower precision of static analysis impedes program understanding significantly [7].

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

We have evaluated performance degradation due to the obfuscation, as indicated in Table 3. 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 4. Similar discussion as in the case of execution time also holds here.

Note that the results presented in this section were obtained by only one time application of our obfuscation tool to each target program. Taking into consideration trade-off of required degree of tamper-resistance and performance

Table 3. Change of execution time

	non-optimized		optimized	
program	Before Obfuscation [sec]	After Obfuscation [sec]	Before Obfuscation [sec]	After Obfuscation [sec]
RC6	0.21	0.27	0.13	0.17
MD5	0.78	1.51	0.26	0.67
jpeg2ps	0.23	0.23	0.17	0.17
Camellia	0.25	0.50	0.07	0.30
FFT	0.20	0.24	0.08	0.11
coretest	0.37	0.38	0.37	0.37

Table 4. Change of program size

	non-optimized		optimized	
program	Before Obfuscation [byte]	After Obfuscation [byte]	Before Obfuscation [byte]	After Obfuscation [byte]
RC6	9420	15696	8200	12168
MD5	18740	36824	13960	22396
jpeg2ps	24028	48988	19380	38260
Camellia	16744	31560	12280	38260
FFT	48068	92476	22228	48804
coretest	9540	16384	8892	13456

degradation, we can apply the tool to a target program arbitrary times. This is one of the greatest advantages of our obfuscation tool. Furthermore, since SUIF has enough generality to support various programming languages, it is easy to modify our tool to obfuscate other languages other than C.

5 Conclusion

Software obfuscation is a promising approach 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. The essence of our obfuscation techniques 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. 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 that obfuscates C programs according to our proposed techniques and in this paper we have described the implementation and discussed 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 graphs of obfuscated

programs are made much more complicated than original ones. They imply 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 et al. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. G. Aigner et al. An overview of the SUIF2 compiler infrastructure. Comp. Syst. Lab., Stanford Univ. http://suif.stanford.edu/.
3. D. Aucsmith. Tamper resistant software: An implementation. In R. J. Anderson ed., *Information Hiding: First International Workshop*, Vol. 1174 of *LNCS*, pp. 317–333. Springer-Verlag, 1996.
4. C. Collberg et al. A taxonomy of obfuscating transformations. Tech. Rep. 148, Dept. of Comp. Sci., the Univ. of 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.
6. H. Goto et al. Evaluation of tamper-resistant software deviating from structured programming rules. In *ACISP 2001*, Vol. 2119 of *LNCS*, pp. 145–158. Springer-Verlag, 2001.
7. M. Hind et al. Interprocedural pointer alias analysis. *ACM Trans. Prog. Lang. Syst.*, 21(4):848–894, 1999.
8. F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In G. Vigna ed., *Mobile Agents Security*, Vol. 1419 of *LNCS*, pp. 92–113. Springer-Verlag, 1998.
9. W. A. Landi. *Interprocedural aliasing in the presence of pointers*. PhD thesis, Rutgers Univ., NJ, 1992.
10. M. Mambo et al. A tentative approach to constructing tamper-resistant software. In *New Security Paradigm Workshop*, pp. 23–33, 1997.
11. T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Inf.*, 28(2):121–163, 1990.
12. E. W. Myers. A precise inter-procedural data flow algorithm. In *Conf. record of the 8th POPL*, pp. 219–230, 1981.
13. C. Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, Univ. of Virginia, 2000.
14. C. Wang et al. Software tamper resistance: Obstructing static analysis of programs. Tech. Rep. CS-2000-12, Dept. of Comp. Sci., Univ. of Virginia, 2000.
15. S. Zhang and B. Ryder. Complexity of single level function pointer aliasing analysis. Tech. Rep. LCSR-TR-233, Lab. of Comp. Sci. Research, Rutgers Univ., 1994.