SAND REPORT
SAND2004-2198
Unlimited Release
Printed June 2004

An Introduction to Software Obfuscation

Philip L. Campbell

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

An Introduction to Software Obfuscation

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

SAND2004-2198
Unclassified Unlimited Release
Printed June 2004

An Introduction to Software Obfuscation

Philip L. Campbell

Networked Systems Survivability & Assurance Department
Sandia National Laboratories
P. O. Box 5800
Albuquerque, New Mexico 87185-0785

Abstract

Obfuscation protects software by making the code more difficult to understand. We review a collection of obfuscation techniques. We then consider what would constitute a theory of obfuscation. Several possibilities that could lead to such a theory are explored.

Table of Contents

List of Figures

List of Tables

1. Introduction

This paper introduces a software protection approach known as " obfuscation." As the name implies, obfuscation protects software by obscuring, by making the code more difficult to understand.

A theory of obfuscation has yet to be developed.[1] So at the moment we have to be satisfied with a list of techniques that appear to obfuscate. The bulk of this paper is therefore a list, developed by Collberg [3][2], of such techniques. As one would expect in a field without a theory, Collberg's list is not exhaustive. This is highlighted by the inclusion later in this paper of obfuscation techniques described by several other researchers. The collection of techniques described in this paper seem adequate to introduce the concept of obfuscation.

This paper is organized as follows:

In Section 2. "Context" on page 2, we first provide context by explaining the three general approaches to software protection, based on Collberg [3].

In Section 3. "Threat Model" on page 2, we present a general obfuscation threat model.

In Section 4. "What Does Obfuscated Code Look Like?" on page 3, we show by example how obfuscated code differs from encrypted code, exemplifying along the way the use of deception and complexity for obfuscation.

In Section 5. "A Taxonomy of Obfuscation Techniques" on page 6, which constitutes the bulk of this paper, we present Collberg's taxonomy of obfuscation techniques referred to in the previous paragraph.

In Section 6. "Other Obfuscation Techniques" on page 25, we present several obfuscation techniques by current researchers, namely Wroblewski, Wang, Hohl, and Ng.

In Section 7. "Secure Function Evaluation" on page 32, we note briefly an alternative to obfuscation.

In Section 8. "Towards a Theory of Obfuscation" on page 33, we present ideas on what a theory of obfuscation would look like.

And in Section 9. "Conclusions" on page 37, we present conclusions.

---

1. As Loureiro notes, " [Obfuscation's] major drawback is the lack of theoretical foundations in order to establish precise definitions of security, and accordingly to be able to quantify the security of the underlying transformations" ([5], page 21).

2. For ease of reference we refer to Collberg, Thomborson, and Low's paper as though Collberg were the sole author.

2. Context

There are three general approaches to software protection, according to Collberg [3], when the software owner does not have physical control of the executing machinery[3]:

1. server-side execution,
2. encryption, and
3. obfuscation.

Server-side execution protects software by never letting it leave the owner's control.

Encryption enables the owner to relinquish some control but not all. Encryption protects software by changing the bits comprising the code so that a key is needed to determine the nature of each bit. However, encrypted code must be decrypted before it is executable. This means that the code is available in plaintext on the executing machine. And this means that encrypted code is intended to be executed only within the owner's sphere of trust. In other words only people the owner trusts should be allowed to have the decryption key.

Obfuscation, on the other hand, protects software not by keeping it under the owner's control and not by encrypting it but rather by making the code hard to understand. Obfuscation uses deception and complexity for this purpose. Obfuscated code can run anywhere (assuming portability). Obfuscated code is unlike encrypted code in that the obfuscated code remains in plaintext.

Unfortunately there is at present no theory that would provide a path to determining the quantitative strength of current obfuscation techniques. Current qualitative comparison of techniques is based on little more than intuition. Even worse, since obfuscation as it is usually understood involves deceiving a human, it is likely that as the field develops so will the humans! So today's obfuscation may not be tomorrow's.

A summary of these three approaches is shown in Table 1.

3. Threat Model

Any defense, such as obfuscation, is meaningful only in terms of an offense. Accordingly we provide a Threat Model. In this paper we presume that the adversary has possession of the obfuscated code in its entirety. We also presume that the adversary has the capability to execute the code, step-by-step if desired. The adversary's initial goal is to understand both what the code is doing and how it is doing it. We presume that understanding the code is of interest to the adversary only within some span of time. The goal of

3. Note that the approaches that Collberg describes are a proper subset of possible approaches.

Table 1 Three Approaches to Software Protection

Approach	Description	Advantages	Disadvantages
Server-side execution	Never let the code leave the owner's machine.	The code is always in the owner's control.	The code cannot traverse the web.[a]
Encryption	Transform the bits constituting a code.	The code is unintelligible.	The code should only be allowed to execute on machines owned by people trusted by the owner of the code.
Obfuscation	Transform the code so that it is hard to understand.	The code remains in cleartext.	There is no theory of strength.

a. If the code accepts input from outside then it may be possible for the adversary to corrupt the server via that input.

obfuscation is therefore to deny the adversary that understanding within that span of time. We do not specify the length of that time span, just as we did not specify what kinds of automated tools are at the adversary's disposal, and thus our Threat Model remains general.

4. What Does Obfuscated Code Look Like?

Encrypted code might look something like that shown in Figure 1.

```
zl d R jQ53sjeluD @ sl dpw. R
      rgbuj& i#jwkmu*wd
s@a#
      jfvpiz d3efkhvn#s
```

Figure 1 Encrypted code (properly formatted)

The corresponding obfuscated code, on the other hand, might look like that shown in Figure 2.

```
if ( ( (7*y*y)-1 ) == (x*x) )
     printf ("hello");
else
     printf ("goodbye");
```

Figure 2 Obfuscated code

The code in Figure 2 could pass for normal code. There are no glaring signs that it is obfuscated. This is not coincidence. However, what prints when the code in Figure 2 runs? If you answer that it obviously depends on the values of x and y, then you have fallen into the obfuscator's trap: you will waste time chasing down the values for x and y. If the obfuscator[4] can keep you in that or in similar traps for enough time then, based on our general Threat Model, the obfuscator benefits. The ideal for the obfuscator is to make the code look like normal code but upon inspection appear so complex that you eventually give up the task and call off the attack. This is the goal of obfuscation, within the constraint that the functionality of the original code remains in the obfuscated code.

Figure 3 shows input from which the obfuscator could have generated the code for Figure 3.

```
if ( false )
     printf ("hello");
else
     printf ("goodbye");
```

Figure 3 An input for Figure 2

In this example the obfuscator exploits the fact that $7y^2 - 1 \neq x^2$, for all integer values of x and y. The strength of this particular obfuscation technique depends upon the obscurity of that fact. Collberg's note that this fact is " well-known" ([3], page 23) may actually be bad news as far as its value for obfuscation is concerned.

4. We use the term " obfuscator" to denote a person who is most likely using a tool to protect software via the application of obfuscation.

However, consider the code in Figure 4.

```
if ( ( (6*y*y)-1 ) == (x*x) )
      a += 2;
else
      a = a + 2;
```

Figure 4 The predicate is unimportant

In this example the predicate is sometimes true and sometimes false but it looks suspicious enough and enough like the predicate in Figure 2 that perhaps you would decide that you again have to figure out values for x and y. Perhaps the predicate is always true or always false for certain ranges of x and y? The predicate is complex enough that making those determinations would require significant time. But the values for x and y again do not matter because the effect of the code in both arms of the conditional is the same. The syntax of the statements in the arms in this example is simple enough to conclude their equivalence by inspection. However, imagine how difficult that conclusion would be to make in the face of an arbitrary amount of additional complexity.

Or consider the code in Figure 5.

```
goto 1;
if ( ( (6*y*y)-1 ) == (x*x) )
      a += 2;
else
      a = a + 2;
1:
```

Figure 5 The If statement is unimportant

In this case the entire IF statement is unimportant because the goto directs control around it: the IF statement never executes.

As you may suspect, with obfuscation anything goes. There are no holds barred. If the obfuscator can deceive you into spending enough time trying to figure out extraneous complexities, then the obfuscator benefits. Deception usually is a matter of distracting you so that you focus your attention away from whatever is important. Meanwhile the complexity holds your attention. To succeed, the deception itself must not be noticed; the additions should blend in with the original code: they should be " stealthy."

5. A Taxonomy of Obfuscation Techniques

A number of obfuscation techniques have been identified. The only taxonomy of obfuscation techniques of which we are aware is the one by Collberg [3].[5] However, a general method to determine the strength of these techniques has not yet surfaced. We believe that the removal of information, such as comments, is effective to some degree and irreversible as well, but we can only make guesses about most of the rest. In a similar vein, we intuitively presume that applying additional techniques will result in additional obfuscation, but this may not be the case. There is no theory to guide us here.

Collberg presents obfuscation techniques as " transformations," emphasizing the change in the code that an application of the techniques will effect. The notation used in the presentation of Collberg's taxonomy is first presented in Table 2.

Table 2 Notation (Sheet 1 of 2)

Notation	Meaning	Examples

P
Q
R
	A predicate. In Java this is a boolean expression. In C this is an expression that evaluates to 0 (meaning false) or non-zero (meaning true).	if (P) ...
		if (Q) ...
		if (R) ...

P^T
Q^T
R^T
	A predicate that always evaluates to true, independent of the values of any variables and functions used in the predicate.
	The first example at the right is of little value because it is obvious by inspection that it is P^T. However, it is not so obvious that the second is also P^T.
	The second predicate, shown above in Figure 2, is an example of what Collberg calls an " opaque" predicate. A predicate qualifies for this if " it has some property q which is known a priori to the obfuscator but which is difficult for the deobfuscator to deduce" ([3], page 10).
		if (1) ...
		
		if ((7 * y * y - 1) != (x*x)) ...

P^F Q^F R^F	A predicate that always evaluates to false.	if (0) ...
$P^?$ $Q^?$ $R^?$	A predicate that sometimes evaluates to true and sometimes evaluates to false.	int i, j; read (i, j); if (i > j) ...

5. Viega & McGraw summarize " simple tricks" for obfuscation into three bullets: " 1. Add code that never executes, or that does nothing. 2. Move code around. 3. Encode your data oddly." ([13], pp. 422-3). They note that " these techniques amount to applying poor programming techniques" (ibid.) and that obfuscation is a " relatively uncharted area" ([13], page 74).

Table 2 Notation (Sheet 2 of 2)

Notation	Meaning	Examples
<name>$^{=<value>}$	A variable whose runtime value is known at compile time to be <value>.	int i = 0 , j = 1; $i^{=0}$ $j^{=1}$
S<integer>	A series of statements, the precise nature of which is not important.	S1; S2;

We present the structure of Collberg's taxonomy in Table 3.

Table 3 Collberg's Taxonomy of Obfuscation Transformations (Sheet 1 of 3)

Transformation	
Layout	Scramble identifiers
	Change formatting
	Remove comments

Table 3 Collberg's Taxonomy of Obfuscation Transformations (Sheet 2 of 3)

Transformation		
Control	Aggregation	Inline method
		Outline statements
		Interleave methods[a]
		Clone methods
		Block loop
		Unroll loop
		Loop fission
	Ordering	Reorder statements
		Reorder loops
		Reorder expression
	Computations	Insert dead or irrelevant code
		Extend loop condition
		[Convert] reducible to non-reducible flow graphs
		Remove library calls and programming idioms
		Table interpretation
		Add redundant operands
		Parallelize code

Table 3 Collberg's Taxonomy of Obfuscation Transformations (Sheet 3 of 3)

Transformation		
Data	Storage & Encoding	Change encoding
		Promote scalars to objects
		Change variable lifetimes
		Split variables
		Convert static data to procedure
	Aggregation	Merge scalar variables
		Modify inheritance relations: factor class, false refactor classes, or add bogus class
		Split, merge, fold, flatten arrays
	Ordering	Reorder instance variables
		Reorder methods
		Reorder arrays
Preventive	Targeted	(Specific to a particular deobfuscator)
	Inherent	Add aliased formals [i.e., parameters] to prevent slicing
		Add variable dependencies to prevent slicing
		Add bogus data dependencies
		Use opaque predicates with side-effects
		Make opaque predicates using difficult theorems

a. i.e., interleave procedures/ subroutines/functions.

In Table 4 we again present Collberg's taxonomy, but this second presentation is augmented with examples. For pedagogical purposes the examples are presented in a high-level pseudo-code- sometimes to look like C and sometimes to look like Java- thereby sidestepping advanced issues involving compiler optimizations and decompilers. Most of the examples are Collberg's. Some

of the examples are adaptations of Collberg's examples. The examples we have invented are denoted with a superscripted dagger (†).

Table 4 Collberg's Taxonomy (with Examples) (Sheet 1 of 12)

Transformation			Example	
			Before	After
Layout		Scramble identifiers	†`int i, j, key;`	`int m1, function5, m2;`

		Change formatting	†`for ( i = 0 ; i < n ; i++ )`
			`{`
			`  j += i;`
			`}`
				`for (i=`
				`0;i`
				`  <n;i++`
				`  ){ j+=`
				`i;}`

		Remove comments	†`// keep your eye on variable i`	
Control	Aggregation	Inline method	Replace the call to a method with the body of the method. The grouping of statements into a method represents information about those statements as a group. Inlining the method removes that information. In the very simple example below we presume that there is something important about the statements in method g for the programmer to take the time to make them into a separate method. By inlining those statements we have removed the aggregation and thus some of the information as well.	

			†`class d`
			`{`
			`  void f ()`
			`  {`
			`  g ();`
			`  }`
			`  void g ()`
			`  {`
			`  S1;`
			`  }`
			`}`
				`class d`
				`{`
				`  void f ()`
				`  {`
				`  S1;`
				`  }`
				`}`

		Outline statements	Replace one or more contiguous basic blocks[a] with a call to a method (and define the new method so that it contains those basic blocks). This transformation works by inserting extraneous information in the form of the construction of a new method that is not present in the original code. The adversary must expend effort to determine that the information represented by this new method is extraneous.	

Table 4 Collberg's Taxonomy (with Examples) (Sheet 2 of 12)

Transformation			Example	
			Before	After

Control	Aggregation	Interleave methods[b]	†`f ( x );`
			`g ( y );`
			`...`
			`private f (int i) { ... }`
			`private g (int j) { ... }`
				`h ( 0, x );`
				`h ( 1, y );`
				`private h ( int k, int ij )`
				`{`
				`  if ( k == 0 )`
				`  f ( ij );`
				`  else`
				`  g ( ij );`
				`}`

		Clone methods	Duplicate methods, then arrange inheritance such that syntactically different calls execute the same code.	

			`class d`
			`{`
			`  f() { S1; }`
			`}`
			
			`d a = new d(); // declares and instantiates a to be of type d;`
			`a.f();`
				`class e { g(){S1;} }`
				`class d extends e { f(){g()} }`
				`d a = new d();`
				`e b = new e();`
				`a.f(); // executes S1;`
				`a.g(); // executes S1;`
				`b.g(); // executes S1;`

		Block loop	†`for ( int i = 0 ; i < n ; i++ )`
			`  b += a[i];`
				`int k = ...; // k >= 0;`
				`for ( int j=0 ; j<n/k ; j++ )`
				`  for ( int i=j*k; i<j*k+k ; i++ )`
				`  b += a[i];`
				`for ( int i=(n/k)*k ; i<n ; i++ )`
				`  b += a[i];`

		Unroll loop	†`for ( int i = 0 ; i < n ; i++ )`
			`  b += a[i];`
				`b = a[0] ;`
				`if ( n > 0 ) b += a[1];`
				`if ( n > 1 ) b += a[2] ;`
				`for ( int i = 3 ; i < n ; i++ )`
				`  b += a[i];`

Table 4 Collberg's Taxonomy (with Examples) (Sheet 3 of 12)

Transformation			Example	
			Before	After

Control	Aggregation	Loop fission	†`int c = 0;`
			`for ( int i = 0 ; i < n ; i++ )`
			`  c += a[i] + b[i];`
				`int c = 0, d = 0;`
				`for ( int i = 0 ; i < n ; i++ )`
				`  c += a[i];`
				`for ( int i = 0 ; i < n ; i++ )`
				`  d += b[i];`
				`c += d;`

	Ordering	These transformations depend upon there being information in the original, lexical order.		
		Reorder statements	In this example we presume that the obfuscator assumes that the code that the adversary is accustomed to seeing proceeds through triply-nested loops using index variables i, j, and k, and in that order. Changing the order therefore implies some unusual kind of computation. The obfuscator hopes that the adversary notices the unusual index ordering and will waste time trying to find out the unusual nature of this usual computation	

			†`for ( int i = 0 ; i < n ; i++ )`
			`  for ( int j= 0 ; j < n ; j++ )`
			`  for ( int k = 0 ; k < n ; k++ )`
			`  z *= a[i][j][k];`
				`for ( int k = 0 ; k < n ; k++ )`
				`  for ( int i = 0 ; i < n ; i++ )`
				`  for ( int j= 0 ; j < n ; j++ )`
				`  z *= a[i][j][k];`

		Reorder loops	`for ( int i = 0 ; i < n ; i++ )` `  a[i] += j;`	`for ( int i = n-1 ; i >= 0 ; i-- )` `  a[i] += j;`
		Reorder expression	†`area = 2 * pi * r * r;`	`area = r * 2 * r * pi;`

Table 4 Collberg's Taxonomy (with Examples) (Sheet 4 of 12)

Transformation			Example	
			Before	After

Control	Computations	Insert dead or irrelevant code	†Insert dead code:
			
			S1;
				if (P^F)
				{
				S2; // dead code;
				}
				else
				{
				S1;
				}

			Insert irrelevant code:
			
			†for (int i = 0 ; i < n ; i++)
			{
			S1;
			}
				int j = 83; // 83 is prime and thus highly suspicious and invites investigation, don't you think?; this comment placed here by your friendly obfuscator;[c]
				for (int i = 0 ; i < n ; i++)
				{
				S1;
				j += 7; //another suspicious number?;
				}
				// no use of j here;

		Extend loop condition	int i = 1;
			while (i < 100)
			{
			S1;
			}
				int i = 1, j = 100;
				while ((i<100) && ((j*j*(j+1) * (j+1)) % 4 == 0)[T])
				// $x^2(x+1)^2$ mod 4 = 0 for all positive integer values of x;
				{
				S1;
				j *= i+3;
				}
				// no use of j here;

Table 4 Collberg's Taxonomy (with Examples) (Sheet 5 of 12)

<table>
<tr><th colspan="3" rowspan="2">Transformation</th><th colspan="2">Example</th></tr>
<tr><th>Before</th><th>After</th></tr>
<tr><td rowspan="3">Control</td><td rowspan="3">Computations</td><td>[Convert] reducible to non-reducible flow graphs</td><td><pre>do
{
    S1;
} while (P);</pre></td><td><pre>1: if (Q?)
{
    2: S1;
    if (P) goto 3;
    goto 4;
}
else
{
    3: S1;
    if (P)
    {
        if (R?) goto 2;
        goto 1;
    }
}
4:</pre></td></tr>
<tr><td>Remove library calls and programming idioms</td><td colspan="2">It may not be possible to remove calls to library routines since the routines are called using the name of the routine. However, new calls could be added such that, for example, a call to subtract is actually a call to pow, as we show below.</td></tr>
<tr><td></td><td><pre>x = pow(a, b); // x = a^b;</pre></td><td><pre>float subtract (int i, int j)
{
    return (pow(i, j));
}
...
x = subtract (a, b);</pre></td></tr>
<tr><td></td><td></td><td>Table interpretation</td><td colspan="2">Convert a subset of the target code such that the subset must be unconverted prior to its execution. In its simplest form this requires calling a method, possibly embedded in the full program somewhere, that at runtime unconverts the converted code. This could be done in Java by using the equivalent of an additional virtual machine: the original bytecodes would be converted, then, at runtime, the additional virtual machine would unconvert them and pass them on to the normal virtual machine. This approach is akin to encryption and to Aucsmith [2].</td></tr>
</table>

Table 4 Collberg's Taxonomy (with Examples) (Sheet 6 of 12)

Transformation			Example	
			Before	After
Control	Computations	Add redundant operands	`x = x + v;` `z = m + 1;`	`x = x + v * i=1;` `z = m + (j=2k / k ) / 2;`
		Parallelize code	This can obscure the control flow if the sequence in which the parallel elements execute is determined at runtime. Two routines that could run in either order are an example of where this transformation could be applied.	

Data	Storage & Encoding	Change encoding	`int i = 1;`
			`while ( i < 1000)`
			`{`
			`  ... a[i] ... ;`
			`}`
				`int i = 11;`
				`while ( i < 8003 )`
				`{`
				`  ... a[(i-3)/8] ... ;`
				`  i += 8;`
				`}`

		Promote scalars to objects	`int i = 1;`
			`while ( i < 9 )`
			`{`
			`  ... a[i] ... ;`
			`  i++;`
			`}`
				`myInt () extends Object`
				`{`
				`  int k;`
				`  myInt (int j) { k=j; }`
				`  public inc () { k++; }`
				`  public int get() {return k;}`
				`}`
				`...`
				`myInt i = new myInt ( 1 );`
				`while ( i.get() < 9 )`
				`{`
				`  ... a[i.get()] ... ;`
				`  i.inc();`
				`}`

Table 4 Collberg's Taxonomy (with Examples) (Sheet 7 of 12)

Transformation			Example	
			Before	After

Data	Storage & Encoding	Change variable lifetimes	`// i is not globally defined;`
			`f ()`
			`{`
			`  int i = 10;`
			`  ... i ...; // no call to g;`
			`}`
			`g ()`
			`{`
			`  // i is not defined here;`
			`  int k = 20;`
			`  ... k ... ; // no call to f;`
			`}`
				`int i;`
				`f ()`
				`{`
				`  i = 10;`
				`  ... i ...;`
				`}`
				`g ()`
				`{`
				`  i = 20;`
				`  ... i ... ;`
				`  // k is changed to i throughout routine;`
				`}`

		Split variables	`bool a, b;`
			`a = true;`
			`b = false;`
			`if ( a ) ... ;`
			`if ( b ) ... ;`
				`short a1, a2, b1, b2;`
				`a1 = 0; a2 = 1; // a = true;`
				`b1 = 0; b2 = 0; // b = false;`
				`// or: a1 = 1; a2 = 0; // a=true;`
				`// or: b1 = 1; b2 = 1; // b=false;`
				`int x = 2*a1+a2;`
				`if ((x==1) || (x==2))...; //if(a)`
				`if ( val (b1, b2) ) ... ; //if(b)`
				
				`int val ( int i, int j )`
				`{`
				`  if ( i == 0 )`
				`  return ( j );`
				`  else`
				`  return ( (j + 1) % 2 );`
				`}`

Table 4 Collberg's Taxonomy (with Examples) (Sheet 8 of 12)

Transformation			Example	
			Before	After

Data	Storage & Encoding	Convert static data to procedure	`System.out.println ("ABA");`	`myPrint(1);`
				`// myPrint(2) does not terminate;`
				`String myPrint( int k )`
				`{`
				`  int i=0; char c; String s;`
				`  while ( 1 ) {`
				`  switch ( k )`
				`  {`
				`  case (0): c='A'; k=4;`
				`  break;`
				`  case (1): c='A'; k=3;`
				`  break;`
				`  case (2): c='B'; k=2;`
				`  break;`
				`  case (3): c='B'; k=0;`
				`  break;`
				`  }`
				`  s.append(c);`
				`  if ( k>3 ) return ( s );`
				`  }`
				`}`

	Aggregation	Merge scalar variables	This transformation assumes that the combined ranges of the original variables fit within the range of the new variable. In this example two 32-bit values, x and y, are stored in one 64-bit value, z. x is stored in the least significant bits; y is stored in the most significant bits: $z = 2^{32} + (y * 2^{32}) + x$. So, if z = 4294967296 then this represents x = y = 0 since $2^{32} = 4294967296$.	

			`int x = 45, y = 100;`
			`x += 4;`
			`y += 10;`
				`long z = 433791696941;`
				`// z = 4294967296 + 429496729600 + 45`
				`  = 2^32 + (100 * 2^32) + 45;`
				`z += 4;`
				`// z = 433791696945`
				`z += 42949672960;`
				`// z = 433791696945 + 42949672960 =`
				`  z + 10 * 2^32;`

Table 4 Collberg's Taxonomy (with Examples) (Sheet 9 of 12)

<table>
<tr><th colspan="3" rowspan="2">Transformation</th><th colspan="2">Example</th></tr>
<tr><th>Before</th><th>After</th></tr>
<tr><td rowspan="3">Data</td><td rowspan="3">Aggregation</td><td>Modify inheritance relations: factor class, false refactor classes or add bogus class</td><td colspan="2">This is accomplished by factoring (splitting up a class) or false refactoring (combining two classes that have no common behavior to form a third that acts as the new superclass of the original two classes) or by adding a bogus class. In the example below both factoring and adding a bogus class are used. We create a new parent class, c, to minimize the lexical effect of this change on other parts of the code.
The reason that this transformation is an obfuscation is that it adds extraneous information that was not present in the original code. Programming constructs such as classes denote information, so splitting a class into two classes adds information. The obfuscator hopes that the adversary will waste time trying to ferret out that information. Whatever information the adversary arrives at should be unrelated to the original code.</td></tr>
<tr><td></td><td><pre>†class a extends b
{
      void f () {...}
      void g () {...}
}</pre></td><td><pre>class c extends b
{
      void f () {...}
}
class a extends c
{
      void g () {...}
      void h () {...} // bogus
}</pre></td></tr>
<tr><td>Split, merge, fold, flatten arrays</td><td colspan="2">Shown below is an example of splitting an array making multiple arrays out of one. Merging arrays is the inverse operation, making one array out of many. Folding (flattening) involves increasing (decreasing) an array's dimensions. Note that splitting and folding add extraneous information and merging and flattening remove existing information.</td></tr>
<tr><td></td><td><pre>int a[9];
... a[i] ... ;</pre></td><td><pre>int a1[4], a2[4];
if ((i % 2) == 0)
      ... a1[i/2] ...
else
      ... a2[i/2] ...</pre></td></tr>
</table>

Table 4 Collberg's Taxonomy (with Examples) (Sheet 10 of 12)

Transformation			Example	
			Before	After
Data	Ordering	These transformations refer to declarations. Their effectiveness depends upon there being information in that original, lexical ordering. Changing the ordering is intended to remove that information.		
		Reorder instance variables	As an example assume that there are several pairs of variables where each pair works in tandem and is independent of the other pairs. Mixing up the declaration order may remove some of this information.	

			†`int i, j;`
			`int k, l;`
			`int m, n;`
				`int l, m, k;`
				`int i;`
				`int n, j;`

		Reorder methods	As an example assume that the order of the appearance of the methods in a given class is from general to specific. Mixing up that order forces the adversary to determine the generality or specificity of each method.	
		Reorder arrays	As an example assume that the order of the declaration for a given collection of arrays is the order in which operations are performed on the arrays and that execution ordering is important in some way. Mixing up that order forces the adversary to determine that execution ordering.	
Preventive	Targeted		The purpose is to " explore known problems in current deobfuscators" ([3], page 24). Collberg gives an example from HoseMocha that adds " extra instructions after every return statement in the source program" which does not change the behavior of the program but causes the Mocha decompiler to crash.	
	Inherent	The purpose is to " make known automatic deobfuscation techniques difficult" ([3], page 24).		
		Add aliased formals [i.e., parameters] to prevent [i.e., inhibit] slicing	`{ ... f ( &i ); ... }` `f ( int *j ) { ... }`	`{ ... f ( &i, &i ); ... }` `f ( int *j, int *k ) { ... }`

		Add variable dependencies to prevent [i.e., inhibit] slicing	`f ()`
			`{`
			`  int x = 1;`
			`  x = x * 3;`
			`}`
				`f ()`
				`{`
				`  int x = 1;`
				`  if ( P^F)`
				`  x++;`
				`  x = x + y^=0;`
				`  x = x * 3;`
				`}`

Table 4 Collberg's Taxonomy (with Examples) (Sheet 11 of 12)

Transformation			Example	
			Before	After
Preventive	Inherent	Add bogus data dependencies	In the example below the direction of the loop is changed, using the example given in the "Reorder loop" transformation above. However, the adversary could automatically note that there are no loop-carried dependencies and could thus automatically convert the direction of the loop. But this would not be possible to do, at least not easily, if a loop-carried dependence is added.	

			`for ( int i = 0 ; i < n ; i++ )`
			`  a[i] = i;`
				`int b[(n-1)*(n-1)/2];`
				`for ( int i = n-1 ; i >= 0 ; i-- )`
				`{`
				`  a[i] = i;`
				`  b[i] += b[i*i/2];`
				`}`

		Use opaque predicates with side-effects	In this example if the adversary removes one but not both of the predicates, k will overflow and crash the executable, assuming that an int is stored in 32 bits using 2s complement.	

			`{ ... S1; ... S2; ... }`	`int k = 0;`
				`f ()`
				`{`
				`  k += 2147483647;`
				`  // 2147483647 = 2^31-1`
				`  return (P^T)`
				`}`
				`g ()`
				`{`
				`  k -= 2147483647;`
				`  return (P^T)`
				`}`
				`...`
				`{`
				`  ...`
				`  if ( f()^T )`
				`  S1;`
				`  if ( g()^T )`
				`  S2;`
				`  ...`
				`}`

Table 4 Collberg's Taxonomy (with Examples) (Sheet 12 of 12)

<table>
<tr><th colspan="3" rowspan="2">Transformation</th><th colspan="2">Example</th></tr>
<tr><th>Before</th><th>After</th></tr>
<tr><td rowspan="2">Preventive</td><td rowspan="2">Inherent</td><td rowspan="2">Make opaque predicate using difficult theorems</td><td colspan="2">This example makes use of the Collatz problem. According to Collberg, " A conjecture says that the loop [in the Collatz problem] will always terminate. Although there is no known proof of this conjecture, the code is known to terminate for all numbers up to 7 * 10^{11}. Thus this obfuscation is safe (the original and obufscated code behave identically [except for some time delay in the obfuscated version, of course]), but difficult to deobfuscate" ([3], page 26).</td></tr>
<tr><td>{ ... S1; ... S2; ... }</td><td>{
...
S1;
int n = random (1, 2147483647);
// 2147483647 = 2^{31}-1
do
n = ((n % 2) != 0) ? (3 * n
+ 1) : (n/2);
while (n > 1);
S2;
...
}</td></tr>
</table>

a. A "basic block" is a sequence of statements such that if the first statement executes then every other statement in the block will execute.

b. i.e., interleave procedures/ subroutines/functions.

c. Adding misleading comments is not in Collberg's taxonomy, perhaps because it would be so hard to automate effectively.

As we noted above, neither Collberg nor anyone else that we know of has arrived at a theory that would determine the strength of an obfuscation. This is a limiting situation. If it is not clear how strong a given obfuscation technique is, if it is not even clear how strong the technique is relative to another technique, then obfuscation is still a curiosity and not ready for the marketplace, though this has not stopped a number of companies from offering obfuscators (and deobfuscators) for sale.

Although Collberg has not provided a theory of strength, he has provided categories that such a theory might use. Collberg considers that the " quality" of a given obfuscation transformation is a function of what he refers to as " potency," " resilience," and " cost." Collberg defines potency as a measure of the strength of a given transformation against a human de-obfuscator. Resilience is a measure of the strength of a given transformation against an automated de-obfuscator. And cost is a measure of both the anticipated increased execution time and increased code size of a given transformation (and not the time or space required to perform the

obfuscation). Collberg provides a scale for each category, as shown in Table 5 (see [3], Table 2).

Table 5 Performance

Measure		Description
Potency	low	(No description provided.)
	medium	
	high	
Resilience	trivial	
	weak	
	strong	
	full	
	one-way	The effect of the transformation cannot be undone.
Cost	free	Requires a constant amount of resources: O(1).
	cheap	Requires a linear amount of resources: O(n).
	costly	Requires a polynomial amount of resources: $O(n^p)$, where $p > 1$.
	dear	Requires an exponential amount of resources: $O(p^n)$, where $p > 1$.
Quality		Quality is a function[a] of Potency, Resilience, and Cost.

a. There is no point in asking *what* function quality is a function of since we do not have values for the measures.

Using intuition Collberg then applies the performance categories shown in Table 5 to the taxonomy of obfuscation transformations

shown in Table 3, which we show in Table 6.

Table 6 Quality of Collberg's Transformations (Sheet 1 of 2)

Transformation			Quality		
			Potency	Resilience	Cost
Layout		Scramble identifiers	medium	one-way	free
		Change formatting	low		
		Remove comments	high		
Control	Aggregation	Inline method	medium	one-way	free
		Outline statements		strong	
		Interleave methods[a]	Depends on the quality of the opaque predicate[b].		
		Clone methods			
		Block loop	low	weak	free
		Unroll loop			cheap
		Loop fission			free
	Ordering	Reorder statements		one-way	
		Reorder loops			
		Reorder expression			
	Computations	Insert dead or irrelevant code	Depends on the quality of the opaque predicate and the nesting depth at which the construct is inserted.		
		Extend loop condition			
		[Convert] reducible to non-reducible flow graphs			
		Remove library calls and programming idioms	medium	strong	Depends on issues outside the scope of this table.
		Table interpretation	high		costly
		Add redundant operands	Depends on the quality of the opaque predicate and the nesting depth at which the construct is inserted.		
		Parallelize code	high	strong	costly

Table 6 Quality of Collberg's Transformations (Sheet 2 of 2)

Transformation			Quality		
			Potency	Resilience	Cost
Data	Storage & Encoding	Change encoding	Depends on the complexity of the encoding function.		
		Promote scalars to objects	low	strong	free
		Change variable lifetimes			
		Split variables	Depends on the number of variables into which the original variable is split.		
		Convert static data to procedure	Depends on the complexity of the generated function.		
	Aggregation	Merge scalar variables	low	weak	free
		Modify inheritance relations: factor class, false refactor classes, or add bogus class	medium	Depends on issues outside the scope of this table.	
		Split, merge, fold, flatten arrays	Depends on issues outside the scope of this table.	weak	free (fold is cheap)
	Ordering	Reorder instance variables	low	one-way	free
		Reorder methods			
		Reorder arrays		weak	
Preventive	Targeted	(Specific to a particular deobfuscator)		trivial	
	Inherent	Add aliased formals [i.e., parameters] to prevent slicing	medium	strong	
		Add variable dependencies to prevent slicing	Depends on the quality of the opaque predicate.		
		Add bogus data dependencies	medium	weak	cheap
		Use opaque predicates with side-effects			free
		Make opaque predicates using difficult theorems	Depends on issues outside the scope of this table.		

a. i.e., interleave procedures/ subroutines/functions.

b. See Table 2 on page 6 for an explanation of "opaque predicates."

6. Other Obfuscation Techniques

In this Section we present the obfuscation techniques presented by several other people, namely Wroblewski, Wang, Hohl, and Ng. These particular researchers represent what we believe to be a sample of the current work in obfuscation. They also span the spectrum from addressing very low level code (Wroblewski) to addressing very high level code (Ng). In Table 7 we show how these approaches fit within Collberg's taxonomy presented in Section 5.. It is significant that for as many transformations as Collberg has fleshed out there appear to be a few more. We have no way now of knowing how many more there might yet still be.

Table 7 Relationship to Collberg's Taxonomy

Name	Techniques	
	From Collberg's Taxonomy	Not Included in Collberg's Taxonomy
Wroblewski	control ordering	replacement; complex insertion
Wang	(Collberg has one mention of aliasing but it is a severe subset of Wang.)	flattening; aliasing
Hohl	split variables	conversion of control flow elements into value-dependent jumps; deposited keys
Ng		intention obfuscation

6.1 Wroblewski

Wroblewski [16] operates on assembly language code and uses four obfuscation techniques:

1. reordering of instructions and blocks,
2. replacement,
3. simple insertion, and
4. complex insertion.[6]

The first and third techniques fit in Collberg's " control ordering" category, with a little stretch. The other two techniques are not explicitly in Collberg's taxonomy.

6. Wroblewski does not name his two insertion types.

Instructions and blocks that share no dependencies can be reordered. If dependencies are shared, then additional control structure will need to be added in the form of jumps to preserve the original ordering. Code that has no relevance to the current context can always be inserted. This is simple insertion. If for a sequence of statements there is an equivalent sequence, then that equivalent sequence can replace the original one. For example, the following two code fragments (from [2])

```
temp = a;
a = b;
b = temp;
```

and

```
a = a ⊕ b;
b = a ⊕ b;
a = a ⊕ b;
```

where a, b, and temp are all of the same scalar type and ⊕ denotes XOR, are functionally equivalent but not equivalent in terms of the ease with which the general programmer will understand that they both swap values.

If the effect of the code on the current context can be later undone, then code that changes the current context can also be inserted, along with, at some other point in the program, the code that will undo that addition. This is complex insertion.

Note that Wroblewski presumes more analysis on a larger scale than does Collberg.

6.2 Wang

Wang [15] uses two techniques:

1. "flattening"[7] and
2. aliasing.

Collberg does not include flattening in his taxonomy. Collberg includes one mention of aliasing, but it is a severe subset of the technique Wang uses.

Flattening, as Wang describes it, is the process of converting the control structure of a procedure to a "universal" structure, as shown

7. Not to be confused with Collberg's array flattening.

in Figure 6 (taken from Wang, Figure 4.4, page 66).

```
while ()
{
            switch ()
            {
                  <procedure body>
            }
}
```

Figure 6 Universal Control Structure

The blocks[8] of the procedure become the statements of the switch. The statements that are at the same time both inside of the while statement and outside of the switch statement control the variable that determines which statement in the switch is executed next. For example, consider the procedure shown in Figure 7.

```
int a, b;
a = 1;
b = 2;
while ( a < 10 )
{
      b = a + b;
      if ( b > 10 )
            b--;
      a++;
}
use(b);
```

Figure 7 Sample Procedure

8. A "block" is a sequence of instructions for which both of the following two conditions hold: (1) no instruction in the sequence is the target of a jump except possibly the first instruction, and (2) there is no jump instruction in the sequence except possibly the last instruction. That is, blocks are defined such that (a) jump targets are always at the beginning of a block and (b) jump instructions are always at the end of a block.

An equivalent universal structure for the code in Figure 7 is shown in Figure 8.

```
int swVar = 1;
while ( swVar < 7 )
{
      switch (swVar)
      {
            case (1):
                  a = 1; b = 2;
                  swVar = 2; break:
            case (2):
                  if (!(a < 10)) swVar = 6;
                  else swVar = 3;
                  break;
            case (3):
                  b = b + a;
                  if (!(b > 10)) swVar = 5;
                  else swVar = 4;
                  break;
            case (4):
                  b--; swVar = 5;
                  break;
            case (5):
                  a++; swVar = 2;
                  break;
            case (6):
                  use (b); swVar = 7;
                  break;
      }
}
1:
```

Figure 8 Universal Structure Equivalent of Figure 7[a]

a. Wang uses goto statements instead of a switch in this example.

Note that the control-flow in Figure 7 has been converted into the data-flow in Figure 8. The code in Figure 8 is not difficult to decipher because the values assigned to swVar are hardcoded. However, consider replacing statements such as " swVar = 2;" with " swVar = g[g[5] +g[g[23]]];" where g is a global integer array whose values are changed every so often during execution. (And now imagine adding several levels of indirection, as in g[g[***i]], where *i happens at the moment to be g[2] and **i happens to be &i.)

The other technique that Wang uses is aliasing: using more than one name for the same memory location. In general resolving aliases

is undecidable. Several examples are shown in Figure 9.

```
Global and local reference aliasing:
      int *i = ...;
      main() { f(&i); ... }
      f (int **j) {
            int *k = *j;
            <k and i now point to the same location>
            ...
      }

Parameter aliasing:
      f (&i, &i);

Aliasing through return values:
      f () {
            int *i = ..., *j = ...;
            j = g ( &i );
            <i and j now point to the same location>
            ...
      }
      int *g (int **a ) { return ( *a ); }

Aliasing through side effects:
      f () {
            int *i = ..., *j = ...;
            g ( &i, &j );
            <i and j now point to the same location>
            ...
      }
      g ( int **a, int **b ) { a = b; }
```

Figure 9 Alias Examples

6.3 Hohl

In this early paper Hohl [4] suggests the use of the following techniques:

1. " variable recomposition,"
2. " conversion of control flow elements into value-dependent jumps," and
3. " deposited keys."

Variable recomposition is what Collberg would call " split variables." The second approach approximates what Wang would call " flattening." The third approach gets values at runtime from some external source, thereby hindering static analysis. For example, the code could fetch the values that determine control flow. Riordan & Schneier use a similar approach [9].

6.4 Ng

Ng [7] uses one technique:

1. intention obfuscation.

This technique does not appear in Collberg's taxonomy.

Ng operates within the context of agents and he is interested in the " intention" of the code: what is the information that the owner wants to know? The easiest way to understand what Ng is talking about is to give an example. Consider the shopping agent shown in very high-level pseudo-code in Figure 10.

```
What is the price of your apples?
```

Figure 10 A Shopping Agent

We presume that the " intention" of the agent's owner (i.e., what the owner of the agents wants to know) is self-evident from the code in Figure 10, that the price of apples is really the information that is wanted. This intention could be a little obscured- or, as Ng

would say, the entropy would be increased- if the agent owner sent the agent shown in Figure 11 instead.

```
What is the price of your apples and the
          price of your oranges?
```

Figure 11 A Second Shopping Agent

The intention would be further obscured if the owner sent agents as shown in Figure 12 to additional stores, stores from which the agent owner would not consider buying (perhaps their quality is low).

```
What is the price of your pears and the
          price of your grapefruits?
```

Figure 12 A Third Shopping Agent

Even if an adversary were to receive the results of all of the agents, that adversary would still not be able to determine the intention of the agent owner.

Note that Ng is operating at a level above the code, at what he might call the " intention" level.

7. Secure Function Evaluation

Another approach to the problem of software protection is what is known as " secure function evaluation" or " computing with encrypted functions." [9] For example the scheme that Sander & Tschudin [12] present uses additive and mixed multiplicative homomorphic properties of the Goldwasser-Micali probabilistic encryption scheme. That is, if we let " E(x)" denote the encryption of data item x, then there are efficient algorithms to compute E(x + y) given E(x) and E(y) and to compute E(xy) given E(x) and y. These

9. There are generally two types of problems involved with the protection of data as opposed to function. One type, known as " secure multi-party computation" or " hiding data from an oracle," is typified by Yao's " Millionaire's Problem" [17]: two millionaires want to find out who is wealthier but do not want to reveal to the other their wealth. The other type of problem is known as " computing with encrypted data" [11] (see also [1]) and uses homomorphic encryption schemes: E(x) op E(y) = E(x op y), where " E(x)" represents the encrypted value of data item x and " op" represents an operation. The code owner encrypts x and y, obtaining E(x) and E(y) respectively, and sends E(x) and E(y) to an adversary who executes op on them and sends back z = E(x) op E(y), whereupon the code owner decrypts the return value to get the plaintext result: D(z) = D(E(x) op E(y)) = D(E(x op y)) = x op y. Both types suffer from high complexity and are currently considered open research problems.

properties enable Sander & Tschudin to protect computations with polynomials. The scheme is restricted to polynomial/rational functions. Loureiro notes that whereas obfuscation lacks a theoretical foundation this approach applies to " more limited models such as circuits" and they have a " large complexity associated with each bit of output" ([5], page 27).

8. Towards a Theory of Obfuscation

As we have already noted a theory of obfuscation is not available today. If and when we have one, we will have more than intuition at our disposal when we compare two obfuscated programs. We would like to be able to determine which program provides more obfuscation. This would enable us to rank order obfuscations. The goal, unfortunately, is more grandiose: we would like to be able to determine the lower bounds in time for an adversary to break the obfuscation. In this section we consider different possibilities for such a theory.

To begin, consider the list of techniques that Collberg has provided (see Section 5. "A Taxonomy of Obfuscation Techniques" on page 6). It is logical to presume that if one of these techniques is good, then two might be better. But it is possible that some sets of techniques work against each other, weakening the set, possibly reducing the obfuscation, maybe even making the program easier to understand.We have no guide here- we are flying blind, so to speak- so we do not know what will happen. Instead of applying a set of techniques, perhaps we could iteratively obfuscate and deobfuscate, choosing obfuscation techniques at each iteration. Given the reasonable assumption that deobfuscators do not always generate the original program, this approach might mimic the " wearing out" of code that makes legacy code eventually unmaintainable. This appears to use entropy to our advantage- an unusual arrangement. For this application we would like to know the relationship between number of iterations and obfuscation strength. But again we have no guide. Worse, the inevitable bugs in the obfuscators and deobfuscators work against us. What confidence do we have that the final obfuscated product has the same input/output behavior as the original program? This line of reasoning suggests that for a theory of obfuscation we need to get " below" the techniques and look for an ideal, for a perfect obfuscation system.

Cryptography has a model of a perfect cryptosystem, namely the Vernam cipher, also known as the one-time pad: a random and never re-used key stream of zeros and ones is XORed with the input, which also consists of zeros and ones. Presuming that the key stream is random and is never re-used, then the cipher stream is perfectly secure. This system has actually been used, though it is impractical for most purposes. The value of the scheme is primarily its ideal nature. Is there a similar ideal scheme for obfuscation that would serve as a starting point?

Yes, as a matter of fact, there is such an ideal scheme for obfuscation. Unfortunately it is even less practical than the Vernam cipher. Here is the scheme: ask the adversary to run all possible programs for n steps- the length of our program- on all possible inputs, reporting the output whenever a program halts;[10] when the input and program we want halts, then we have our answer. At some point thereafter we tell the adversary to stop charging us for compute cycles. We could call this " Vernam obfuscation," to suggest that this is a perfect method of obfuscation. Surprisingly there is no obfuscation involved! Our program is run as is. We protect our

program by hiding it amongst many other programs. Although this is a completely impractical approach it does provide us with a starting point.

We can make Vernam obfuscation increasingly efficient[11] by limiting the size and the range of the input or the number of programs executed, but as we do so we provide the adversary with more information, narrowing the adversary's search space. Does this process narrow the adversary's search space faster than what we gain in efficiency? What is the shape of the function that describes the trade-off between efficiency and security?

Unfortunately, the Vernam obfuscation approach seems to lack a "workload advantage." We want to show two results simultaneously. First, we want to show that for a given program with n steps that the adversary has to pay an exponential price to break the obfuscation. That is, the adversary has to consider m^n programs, where $m > 1$. We could call this "possible program explosion." Second, we want to show that for the same program we have to pay only a linear price, or maybe only a polynomial price, to create the obfuscation and execute the resulting obfuscated program. If we can show these two results, then we have the adversary "over a barrel," as the expression is, and we are on our way to a theory of obfuscation.

Perhaps we could construct Vernam obfuscation by building a program that uses p instructions for each instruction in the original program. Rivest [10] presents a cryptographic approach, called "winnowing," that provides privacy via integrity that is similar to this. The idea of winnowing, in the extreme, is that the sender sends both a zero and a one for each bit in the message. For each zero and for each one, the sender includes a Message Authentication Code (MAC) such that for each pair of bits only one MAC will authenticate. The receiver recovers the message by "winnowing," by discarding as "chaff" the bit in each pair that does not authenticate. The message, like an obfuscated program, is in plaintext, but the message is afforded privacy because it is hidden, in a sense: the adversary does not know which bit of each pair is part of the message. So the adversary has to consider all 2^n possible messages. This is the workload advantage. Can we make this approach work for obfuscation?

We could begin by considering an approach that for simplicity uses two instructions for each of the n instructions in the original program. We obfuscate by adding a "phony" instruction for each real instruction. The adversary is forced to winnow the real instructions from the phony ones.

The problem we have that the winnowing message-sender does not have is state: state persists between instructions. The adversary in the winnowing case has to consider all 2^n possible messages because either the zero or the one of each pair can be in the message. But this is not the case with our program. For example, the following shows pairs of instructions for each step in the obfuscated program:

---

10. Of course any reasonable adversary would run the programs using dovetailing so that he is not caught trying to complete a program that does not halt.
11. Or perhaps it is better to say, "decreasingly inefficient."

step i: load register 1 from address ...

(another instruction that does not use or load register 1)

step i+1: load register 1 from address ...

(another instruction)

The adversary knows that the instruction in step i from the original program is not the load, simply because that load is immediately overwritten in the next step.[12] The first load is "dead" code.

Our task is to construct a program such that any (or at least enough) of the instructions at step i could be part of a program that uses any (or at least enough) of the instructions at step i+1 (or some subsequent step), for any (or at least enough) i in the range of n. This is the first piece of the puzzle.

The second piece of the puzzle is the use of a key. Like cryptography the same obfuscation algorithm operating on the same program should produce a different obfuscated program given a different key. Using Kerckhoff's assumption, the security of the scheme should rest as much as possible on the security of the key alone. If we can align obfuscation with that assumption, then we can use results from cryptography to help with a theory of obfuscation.

Assuming the simplest approach, namely that the obfuscator adds one instruction for each instruction in the original program, the key could be used to determine whether the original or the phony (i.e., the added instruction) instruction comes first, as suggested in Table 8.

12. Unless, of course, the instruction at step i in the original program is not the equivalent of a no-op.

Table 8 Initial Part of an Obfuscated Program

Step	Sample Key Bit [a]	Instruction Sequence
0	0	(real instruction 0)
		(phony instruction 0)
1	1	(phony instruction 1)
		(real instruction 1)
2	1	(phony instruction 2)
		(real instruction 2)
3	0	(real instruction 3)
		(phony instruction 3)
...	...	...

a. If the i^{th} bit of the key is 0, then the real instruction for the i^{th} pair is the first in the pair, otherwise it is the second in the pair.

We presume that the key would be necessary to extract the results of program execution. This would suggest that a superset of the output of the original program should be sent back as a result of execution to the obfuscator's computer. Perhaps that superset could be something like a trace [13]. Although this suggests that this approach could provide execution integrity, and privacy of execution, code, and data,[13] it is not clear what output should be generated.

The key could, like a one-time pad, have as many bits as the program has instructions and thus be just as random as keys for the Vernam cipher. Unlike the Vernam cipher the key never has to leave the owner's control.

Unfortunately we have not addressed how we get a sufficient possible program explosion via this approach. That is, which

13. This approach is too loosely defined to determine if it precludes the adversary from violating execution integrity by returning bogus output. Since some of the instructions inserted for obfuscation will execute, perhaps they could also serve to generate a result that provides a check on execution integrity.

instructions do we use for the phony ones?

Is either puzzle piece possible? To our knowledge these are open questions.

However, before we leave this topic, consider another twist. Rivest points out that what we consider "chaff" could actually be another message.[14] In fact there could be m messages all interleaved in some random way known only to the sender.[15] Each of the m recipients, using its unique key, can separate its wheat (the bits in the message intended for that recipient) from the chaff (the bits intended for some other recipient, or the bits that really are chaff, included to confuse the adversary). As the broadcast stream continues, some messages complete while new ones begin as the quantity of true chaff waxes and wanes. Applying this to obfuscation, can we combine two programs such that it is infeasible for the adversary to untangle them?

The problem again is state. The state of a program includes the contents of some registers (at least the program counter) and some portion of memory (at least the data that is contributing to the output). We are tempted at this point to appeal to functional programming [6] because of referential transparency: a functional programming function depends only on its inputs. The same function always returns the same results given the same inputs; it is independent of state. This is a step toward messages that consist of zeros and ones that also carry no state. This suggests that instead of considering instructions as our basic unit of obfuscated programs, as we have in the discussion above, we should consider functional programming functions.[16] This is counter-intuitive because these objects are at a higher level of abstraction than instructions and thus more likely to be easier to understand.[17] But such functions are more mutually-independent and thus more amenable to winnowing. Recall that the zeros and ones in winnowing are by themselves perfectly easy to understand. Is there light ahead?

Finally we need to consider steganography. This is the study of hidden messages. The goal is for the adversary to be unaware of even the existence of a hidden message. Would this work for obfuscation? That is, could there be such things as hidden programs?

9. Conclusions

We have taken a quick tour of obfuscation, starting with its context and a general threat model, to a brief understanding of the use of deception and complexity, to Collberg's taxonomy of obfuscation transformations, to a summary of the techniques used by four

---

14. That is, one man's wheat is another man's chaff.

15. The collecting of many programs in order to protect each has some similarities to the Crowds system for anonymity of web transactions [8].

16. There is a subtlety here. What the adversary sees could consist of assembly language statements generated by a compiler, just as we have tacitly assumed in the discussion further above. But if we are using functional programming functions, then alternatively and without loss of protection what the adversary sees could consist of high-level language statements since we presume that the adversary could decompile that assembly language into a high-level language. If the obfuscation is done properly, then the high-level language form would still be too difficult to understand.

17. Indeed, the intent of functional programming is to make programs easier to understand, not harder.

representative researchers in the field, to a passing note on an alternative to obfuscation, and finally to thoughts on a theory of obfuscation.

Conventional software wisdom holds that obfuscation is the natural state of software, that like entropy, obfuscation in software tends to monotonically increase over time. Of all man-made artifacts software may be unique in that it never wears out. But oddly enough it still has to be " maintained." And even with careful attention it can still slip through our fingers into oblivion, forcing an entire re-write of the code at sometimes enormous expense. The whole field of software engineering can be seen as an effort to hold in check that constant drift. So it is curious that when we willfully attempt to push that drift along to some impenetrable state that we find ourselves without tools or gauges, without maps or signposts or even milestones, moving about in a kind of software Sargasso Sea. Perhaps the more we learn about obfuscation the more we will learn about the true nature of software.

References

[1] Mikhail J. Atallah, K. N. Pantazopoulos, John R. Rice, Eugene E. Spafford, " Secure Outsourcing of Scientific Computations." Advances in Computers, Vol. 54, pp. 215-272. 2001.

[2] David Aucsmith, " Tamper Resistant Software: An Implementation." Lecture Notes in Computer Science. Vol. 1174. Springer-Verlag, Berlin. pp. 316-333. 1996. 2 refs.

[3] Christian Collberg, Clark Thomborson, Douglas Low, " A Taxonomy of Obfuscating Transformations." Technical Report 148, Department of Computer Science, University of Auckland. 1997. 36 pages. 33 refs.

[4] Fritz Hohl, " Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts." Mobile Agents and Security. G. Vigna, editor. Lecture Notes in Computer Science, Volume 1419, pp. 92-114, 1998. Springer-Verlag , Berlin. 13 refs.

[5] Sergio Loureiro, " Mobile Code Protection." Ecole Nationale Superieure des Telecommunications. Paris, France. January 26, 2001. 142 pages.

[6] Bruce J. MacLennan, Functional Programming: Practice and Theory. Addison-Wesley, New York. 1990.

[7] Sau-Koon Ng, " Protecting Mobile Agents Against Malicious Hosts," Masters Thesis. Division of Information Engineering. The Chinese University of Hong Kong. June 2000. 123 pages. 95 refs.

[8] Mark K. Reiter, Aviel D. Rubin, " Crowds: Anonmymity for Web Transactions." ACM Transactions on Information and System Security, June 1999.

[9] James Riordan, Bruce Schneier, " Environmental Key Generation Towards Clueless Agents," Mobile Agents and Security. G. Vigna, editor. Lecture Notes in Computer Science, Volume 1419, pp. 15-24, 1998. Springer-Verlag , Berlin. 8 refs.

[10] Ronald L. Rivest, " Chaffing and Winnowing: Confidentiality without Encryption." MIT Lab for Computer Science. March 18, 1998. (http://theory.lcs.mit.edu/~rivest/chaffing.txt).

[11] Ronald L. Rivest, Leonard Adleman, and M. Dertouzos, " On data banks and privacy homomorphisms." In Demilllo, Dobkin, Jones, and Lipton, editors, Foundations of Secure Computation, pp. 169-80. New York: Academic Press, 1978.

[12] Tomas Sander, Christian F. Tschudin, " Towards Mobile Cryptography." 1998 IEEE Symposium on Security and Privacy. May 3-6, 1998, Oakland, CA. pp. 215-24.

[13] John Viega, Gary McGraw, Building Secure Software: How to Avoid Security Problems the Right Way. Addison-Wesley, 2002. 493 pages. ISBN 020172152X.

[14] Giovanni Vigna, " Protecting Mobile Agents Through Tracing." 3rd ECOOP Workshop on Mobile Object Systems, Jyvälskylä, Finland, June 1997.

[15] Chenxi Wang, " A Security Architecture for Survivability Mechanisms," Ph.D. Dissertation, University of Virginia, Department of Computer Science, October 2000. 209 pages. 82 refs.

[16] Gregory Wroblewski, " General Method of Program Code Obfuscation." (draft) Ph.D Dissertation. Wroclaw 2002. 112 pages. 77 refs.

[17] A. C. Yao, " Protocols for secure computations." IEEE Symposium on Foundations of Computer Science 82, pp. 160-4, Chicago, Illinois, 1982.

Distribution

1	0455	R. D. Pollock, 5501
1	0455	R. S. Tamashiro, 5517
1	0708	J. M. Covan, 6202
1	0784	J. M. DePoy, 5512
1	0784	R. D. Halbgewachs, 5501
1	0784	M. J. Skroch, 5512
1	0784	R. E. Trellue, 5501
1	0784	C. M. Villamarin, 5512
1	0785	W. E. Anderson
1	0785	C. L. Beaver, 5514
10	0785	P. L. Campbell, 5516
1	0785	J. D. Dillinger, 5516
1	0785	D. L. Harris, 5516
1	0785	R. L. Hutchinson, 5516
1	0785	A. J. Lanzone, 5514
1	0785	T. S. McDonald, 5514
1	0785	W. D. Neumann, 5514
1	0785	R. C. Schroeppel, 5514

1	0785	M. E. Senglaub, 5516
1	0785	J. E. Stamp, 5516
1	0785	B. P. Van Leeuwen, 5516
1	0806	L. G. Pierson, 9336
1	1030	W. F. Hossley, 12870
1	1351	J. J. Torres, 5517
1	1371	B. G. Varnado, 4142
1	1411	C. C. Battaile, 1834
1	1411	M. E. Chandross, 1834
1	9018	Central Technical Files, 8945-1
2	0899	Technical Library, 9616
1	0612	Review & Approval Desk, 9612 For DOE/OSTI