

Intellectual Property Protection Using Obfuscation

Stephen Drape

Oxford University Computing Laboratory

September 2009

Contents

- Definitions of Obfuscation
- Obfuscation Survey
- Evaluating Obfuscation Tools
- Problems with the Automation of Obfuscations



What is Obfuscation?

An obfuscation is:

- a program transformation
- used to make programs “harder to understand”
- a technique for protecting intellectual property
- **not** encryption

The area of obfuscation needs more research

Two Important Papers

-  Christian Collberg, Clark Thomborson, and Douglas Low.
A taxonomy of obfuscating transformations.
Technical Report 148, Department of Computer Science,
University of Auckland, July 1997.
-  Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang.
On the (im)possibility of obfuscating programs.
In Proceedings of the 21st Annual International Cryptology
Conference on Advances in Cryptology, pages 1–18.
Springer-Verlag, 2001.

Possible Definitions of Obfuscation

We will look at the following definitions:

- Collberg *et al.*
- Barak *et al.*
- Assertion definition
- Slicing definition
- Attack model definition

Definition of Collberg *et al.*

The transformation $P \xrightarrow{\mathcal{T}} P'$ is an **obfuscating transformation**, if P and P' have the same **observable behaviour**:

- If P fails to terminate or terminates with an error condition, then P' may or may not terminate.
- Otherwise, P' must terminate and produce the same output as P .

and \mathcal{T} makes P' “hard to understand”.

Definition of Barak *et al.*

An obfuscator \mathcal{O} is a “compiler” which takes as input a program P and produces a new program $\mathcal{O}(P)$ such that for every P :

- **Functionality** — $\mathcal{O}(P)$ computes the same function as P .
- **Polynomial Slowdown** — the description length and running time of $\mathcal{O}(P)$ are at most polynomially larger than that of P .
- **“Virtual black box” property** — “Anything that can be efficiently computed from $\mathcal{O}(P)$ can be efficiently computed given oracle access to P ”.

Then a family of functions is created which is unobfuscatable in the sense that there is no way of obfuscating programs that compute these functions.

Assertion Definition

- In the thesis of Drape, obfuscations were defined for abstract data-types which consist of a series of operations and a list of assertions that the operations satisfy.
- If we obfuscate an operation then it should be harder to prove that the obfuscated operation satisfies an assertion.
- This formed the basis for the definition of an **assertion obfuscation**.

Slicing Definition

- Program slicing is a tool used to aid program comprehension.
- A program slice consists of the parts of a program that potentially affect the values of some variables computed at some point of interest.
- We can produce **slicing obfuscations** which decrease the effectiveness of a program slicer for particular variables.
- Obfuscations which are particularly good slicing obfuscations create dependencies between the variables of interest.

Attack Model Definition

- When designing obfuscations we should know which attack or analysis tools we are aiming to defend against — *i.e.* an **attack model**.
- We can use analysis metrics as a guide to the quality of the obfuscations.
- From an obfuscation perspective, a “good” obfuscation should decrease the effectiveness of an analysis.

Classifications

We can classify obfuscations into various categories:

- Layout obfuscations
 - renaming, formatting changes, removing comments
- Control-Flow Obfuscations
- Data Obfuscations
- Language Dependent Obfuscations

Layout obfuscations are not considered to be very strong but, nevertheless, they are used in many obfuscation tools.

Control-Flow Obfuscations

Control-Flow Obfuscations aim to change the structures of conditionals, jumps and loops. Examples are:

- **Using opaque predicates**
- **Loop Fusion and Fission**
- **Changing Loop conditions**
- **Control-Flow Flattening**

Opaque predicate example

We denote p^T (resp. p^F) to denote a predicate p whose value is known to be true (resp. false) at compile time. We can use the predicate p to obfuscate a statement S as follows:

$$S \Rightarrow \text{if } (P^T) \{S\}$$
$$S \Rightarrow \text{if } (P^F) \{S_{bug}\} \text{ else } \{S\}$$
$$S \Rightarrow \text{if } (P^?) \{S\} \quad \text{else } \{S_{copy}\}$$

Opaque predicate example

We denote p^T (resp. p^F) to denote a predicate p whose value is known to be true (resp. false) at compile time. We can use the predicate p to obfuscate a statement S as follows:

$$\begin{aligned} S &\Rightarrow \text{if } (P^T) \{S\} \\ S &\Rightarrow \text{if } (P^F) \{S_{bug}\} \text{ else } \{S\} \\ S &\Rightarrow \text{if } (P^?) \{S\} \quad \text{else } \{S_{copy}\} \end{aligned}$$

Predicates can be based on known mathematical identities, for example $n^2 + 1 \neq 0 \pmod{7}$ for all integral n . But these identities are either quite easy prove or are so complicated that they stand out from the rest of the program.

Example Loop Transformation

We can add a bogus induction variable into the loop which can make the loop conditions more complicated. For example:

```
i = 0;  
while (i < 10)  
{  
  ...  
  i = i + 1;  
}
```

⇒

Example Loop Transformation

We can add a bogus induction variable into the loop which can make the loop conditions more complicated. For example:

```
i = 0;
while (i < 10)
{
    ...
    i = i + 1;
}
```

\Rightarrow

```
j = 1;
i = 0;
while ((i < 10) && (j < 120))
{
    ...
    i = i + 1;
    j = j + 2 * i;
}
```

We assume that the value of i is only changed at the end of the loop. The bogus variable j can be used to create fake dependencies.

Data Obfuscations

Data Obfuscations aim to change the data structures of programs.
Example data obfuscations are:

- **Variable Encoding**
- **Merging**
- **Splitting**
- **Array Transformations**

Variable Encodings

An example variable encoding, using constants a and b , is:

$$i \Rightarrow a * i + b$$

Using this transformation

$$i = 2 \quad \Longrightarrow \quad i = a * 2 \qquad j = i + 1 \quad \Longrightarrow \quad j = \frac{i-b}{a} + 1$$

Variable Encodings

An example variable encoding, using constants a and b , is:

$$i \Rightarrow a * i + b$$

Using this transformation

$$i = 2 \quad \Longrightarrow \quad i = a * 2 \quad \quad j = i + 1 \quad \Longrightarrow \quad j = \frac{i-b}{a} + 1$$

An expression such as $i++$ is both a definition and a use and so

$$i++ \quad \Longrightarrow \quad i = a * \left(\frac{i-b}{a} + 1 \right) + b$$

Variable Encodings

An example variable encoding, using constants a and b , is:

$$i \Rightarrow a * i + b$$

Using this transformation

$$i = 2 \quad \Longrightarrow \quad i = a * 2 \quad \quad j = i + 1 \quad \Longrightarrow \quad j = \frac{i-b}{a} + 1$$

An expression such as $i++$ is both a definition and a use and so

$$i++ \quad \Longrightarrow \quad i = a * \left(\frac{i-b}{a} + 1 \right) + b$$

which (using exact arithmetic) can be simplified to

$$i = i + a$$

Example Variable Splitting

Suppose that we had an integer variable x then we can split x into two variables a and b as follows:

$$a = x \operatorname{div} 10 \quad \text{and} \quad b = x \operatorname{mod} 10$$

Example Variable Splitting

Suppose that we had an integer variable x then we can split x into two variables a and b as follows:

$$a = x \operatorname{div} 10 \quad \text{and} \quad b = x \operatorname{mod} 10$$

For example, under this transformation, the statement $x++$ is transformed to:

$$\begin{aligned} a &= (10 * a + b + 1) \operatorname{div} 10; \\ b &= (b + 1) \operatorname{mod} 10; \end{aligned}$$

Example Variable Splitting

Suppose that we had an integer variable x then we can split x into two variables a and b as follows:

$$a = x \operatorname{div} 10 \quad \text{and} \quad b = x \operatorname{mod} 10$$

For example, under this transformation, the statement $x++$ is transformed to:

$$\begin{aligned} a &= (10 * a + b + 1) \operatorname{div} 10; \\ b &= (b + 1) \operatorname{mod} 10; \end{aligned}$$

These assignments are equivalent to:

```
if (b == 9) {a = a + 1; b = 0;} else {b = b + 1;}
```

Example Array Splitting

Collberg *et al.* give the following example of an **array split**:

```
int [] A = new int [10];
```



```
...  
A[i] = ...;
```

Example Array Splitting

Collberg *et al.* give the following example of an **array split**:

```
int [] A = new int [10];  
  
...  
A[i] = ...;
```

⇒

```
int [] A1 = new int [5];  
int [] A2 = new int [5];  
  
...  
if ((i % 2) == 0) A1[i/2] = ...;  
    else A2[i/2] = ...;
```

This transformation was generalised in the thesis of Drape so that the transformation works with more general data-types.

Language Dependent Obfuscations

Most of the obfuscations seen so far can be applied to a variety of language paradigms. However we can create obfuscations which rely on specific language features:

- **Exceptions**
- **Intermediate Language Commands**
- **Creating Irreducible Loops**

Intellectual Property Protection report

The project report “**Intellectual Property Protection**” written by Stefan Vogl, Emanuel Mathis, Martin Ortner and Georg Schönberger from the University of Applied Sciences of Upper Austria, Hagenberg was written as part of a semester project on behalf of Siemens.

The report considers various protection methods, including obfuscations, which aim to protect the Intellectual Property of programs. Three different programming languages (and associated lower level languages) were considered: **ANSI-C** (with MS-DOS), **Java** (with Bytecode) and **C#** (with .NET CIL).

Obfuscation Techniques Considered

There were only a few obfuscation techniques that were in the tools studied in the report:

- Name obfuscation for Java and C# and token renaming for assembler language
- Shrinking (removing unnecessary information) for Java
- Confusing assembler control-flow with opaque predicates
- Adding bogus code between blocks in bytecode
- Using **switch** blocks in CIL
- String encryption

Evaluation of Techniques

- As expected, layout obfuscations were commonly used.
- String Encryption was the only technique which could be classed as a data obfuscation: string encryption is considered to be a weak obfuscation.
- Trivial opaque predicates were used (e.g. $1 \neq 2$) — it is hard to write suitable predicates in a low-level language.
- It is doubtful that these techniques would survive the compilation/decompilation process (many of these transformation could be removed by an optimising compiler).
- It is unclear whether the lack of good obfuscation techniques is due to the particular tools that were considered or is indicative of commercial tools in general.

Review of Obfuscation Techniques

Only a handful of the many obfuscation techniques discussed earlier were actually implemented in obfuscation tools. Why was this?

We will look at some of the problems associated with implementing obfuscations, such as

- **Stealth**
- **Creating Opaque Predicates**
- **Placement**
- **Choice of Variables**

Stealth

- Collberg *et al.* discuss the concept of **stealth**.
- An obfuscation has stealth if it does not “stand out” from the program.
- Currently, there is no quantitative measure of stealth as it is very context sensitive.
- The notion of stealth makes the process of creating automatic obfuscators hard.
- We may be restricted to creating more general obfuscations than can be applied to a variety of situations.

Creating Opaque Predicates

It is hard to generate suitable opaque predicates but Collberg *et al.* proposed generating predicates based on mathematical identities, however:

- Such identities are generally not stealthy
- These identities are usually true (or false) for all integers (say).
- Attackers will spend their time trying to remove these
- Maybe we should include fake predicates that are deliberately not stealthy to confuse attackers

Using invariants to create predicates

An alternative to using mathematical identities is to use program invariants.

- To obfuscate a particular point, we need to find an invariant at this point and create a predicate based on this invariant
- The predicate will be true at this program point but not necessarily at other program points.
- To remove such a predicate, an attacker needs to understand the part of the program
- The obfuscator needs to find suitable predicates — it is hard to do this automatically

Where should we place obfuscations?

When creating obfuscations we need to choose the “best” places to put them. How do we make this choice?

- We should place obfuscations at “sensitive” or “important” places in a program
- However, doing this may slow down the programs so there is a trade off between the level of obfuscation and efficiency?
- Where should we add opaque predicates?
- Where should we put fake or bogus obfuscations?

These are difficult decisions for an automatic obfuscator.

Which variables should we obfuscate?

A related problem to placement is the choice of variable.

- Should we choose to obfuscate a variable which is used frequently? (If so we may risk affecting the efficiency)
- Which variables should we split, merge or encode?
- What about arrays?
- Which variables should we use in an opaque predicate?

When creating slicing obfuscations, Majumdar *et al.* used a slicer to determine the area of the code in which obfuscations should be placed and which variable to obfuscate. Maybe we should use analysis tools before obfuscating?

What's next?

- So far we have surveyed various obfuscation techniques and considered some protection tools — however we found that the obfuscations contained in these tools were not very good.
- We would like to develop more successful obfuscations than the ones that are commonly found in protection tools.
- There are a number of issues that may hinder the successful implementation of more productive obfuscation techniques.
- What obfuscations techniques do we think can be implemented quickly?

Immediate

Here are some obfuscations which could be implemented fairly easily:

- Using attack tools to guide the creation of obfuscations
- Creating irreducible jumps in Intermediate Language
- Using intermediate level commands which are not easily translated into the source level language
- Creating malicious layout obfuscations (such as renaming variables to misleading names)

Short Term

Here are some techniques which will require further research but should be fairly straightforward to implement:

- Using invariants to create opaque predicates
- Creating simple array obfuscations
- Developing simple loop obfuscations (such as adding dummy variables or extending the scope of loop variables)
- Producing deliberately unstealthily code with the aim to focus an attacker's attention on an unimportant area of the program

Long Term

Finally, here are some techniques which require extensive research and effort before they can be successfully created:

- Developing a measure for the stealth of an obfuscation
- Creating loop obfuscations for Intermediate Language
- Automating the process of using a slicer (or other attack tools) to provide information about suitable places for obfuscation
- Obfuscating more than a single method

Summary

- Creating good obfuscations is not an easy task as there are many issues to consider
- The area of obfuscation is much more subtle than the “all-or-nothing” approach to encryption
- There are a variety of techniques but most of them are hard to implement
- There is a rich field of obfuscations that have not been discussed here: e.g. pointers, OO methods, threads
- Obfuscation needs much more research...