Program obfuscation. A small overview.

Jaak Pruulmann-Vengerfeldt

January 7, 2006

There is quite a lot of reasons why one should consider using thing called (program) obfuscation. One can find a lot of implementations of obfuscation. Yet, there is relatively small amount of theoretical results related to obfuscation and obfuscators. This report gives an overview of existing important results. As the real world is a big tradeoff, the report also contains a section describing program obfuscation from the practical point of view.

1 Definitions, sort of.

At first we should define concepts we are going explore. One can say that the lack of consistent and widely accepted definition is somewhat the problem with obfuscation. It tends to be that everyone who has something to do with obfuscation is required to define his or her own specific variant of obfuscation.

The goal of obfuscation is hide the secrets inside some given program while preserving it's functionality. Informally, **obfuscator** $\mathcal{O}$ is an efficient, probabilistic "compiler" that transforms a program P into a new program $\mathcal{O}(P)$ such that:

- The input/output behaviour of $\mathcal{O}(P)$ is the same as P
- "Anything that can efficiently computed from $\mathcal{O}(P)$ can be efficiently computed given oracle access to P."
- For practical reasons, the obfuscated program$\mathcal{O}(P)$shouldn't be much less efficient than the original program. We allow it to be somewhat less efficient

There are some other definitions of obfuscation. Eg, informally: "hacker should not be able to recover complete source code of the input program from the obfuscated program.". Consequently, if the hacker manages to recover the source code of program P from the obfuscated program, then we say that obfuscator **completely failed** on the program P. One might argue that it implies that there are obfuscators that cannot completely fail on some programs (consider an obfuscator that removes comments from source code. If there is some secret value only in comments, then hacker cannot learn it from "obfuscated" program). On the other hand there are programs on which *every* obfuscator fails. Consider a program that prints it's own source.

Learnability. We say that a program is **learnable** if we can understand how it works internally just by looking at it's inputs and outputs. Obviously there is no point in obfuscating such a program. Good example of learnable programs are *quines,* the programs that are reproducing their source code – only thing a malicious hacker has to do with obfuscated quine is to run it and to collect the source code it outputs.

It's useful to define some notions of security also. Depending on security level some cryptographic construction such as encryption, digital signature, hash function or obfuscator, provides, we can divide them into two categories. Boaz Barak calls those **well-defined security** and **fuzzy security.**

Well-defined security means that cryptographic construction has some (well defined) properties that must hold to consider that construction secure. Well-definedness allows us to prove if some particular construction or implementation of some construction has required properties or not. So, well-defined security is a prerequisite of proved security.

On the other hand, fuzzy security is about intuition or vague claims about security. It's usually a process in which someone claims something about security of some construction, then someone uses the construction, someone breaks it, it will be revised and improved and everything starts again.It is never formally specified what we mean if we say "it's secure" and therefore it cannot be proven secure.

These concepts are important because obfuscation and obfuscators are traditionally fuzzy security – it is hard to define clearly the "resilience" of obfuscation.

2 Motivation

It's a bit strange that there is not that much of theoretical works on obfuscation, because obfuscation is important ingredient in many important constructs.

First and probably most widely known use is protection of intellectual property. There are algorithms and methods that are useful and probably quite hard to patent – be it because of the time it takes to patent something or because of method being so easily modified so that patenting it would be pointless. With computer programs it is sometimes also important to protect the program against modifications such as removing some kind of access control or advertising. Average software vendor would very much like to sell his/her product as black box that does something but doesn't allow it's user to tamper with it's inside. Obvious solution would be sell software with hardware that runs it (as Microsoft tries to do with Xbox) but it's not really acceptable within todays world of general purpose computers. A bit modified case of protection of intellectual property is protecting some watermark.

It's known that if we have generic obfuscator, we could easily construct homomorphic cryptosystems out of public key encryption. Homomorphic cryptosystem is defined as follows. Let us have some messages x and y, and some encryption function E. Now, we call this encryption scheme homomorphic if for every x,y and for some operator $*$, given$E(x)$and $E(y)$ we are able to calculate $E(x * y)$ without really finding out x or y. It's obvious that if we could securely obfuscate the function that decrypts $E(x)$ and $E(y)$, applies $*$ to them and encrypts the result, we could construct homomorphic encryption scheme out of every public key encryption scheme and for every operation we need.

Controlled delegation of authority and access. Suppose someone wants to calculate something and needs some secret ingredient. It might be some delicate personal data for statistics or some secret password for accessing something. One could write a program that controls access for that particular data or secret but it obviously would contain that secret. So, if we could obfuscate that program, we could securely delegate authority and access.

As a special case of previous construction, we could build public key cryptosystems from symmetric (private key) systems. For private key we use natural private key of the system we use as the base system. Public key is partially evaluated encryption function that has the very same private key as seed. We convert binary function $E(K,x)$ into unary function $E_K(x)$ and obfuscate it. The result is our public key.

Secure obfuscator might allow us to securely remove random oracles. Although it's known that it will not work in general, there might be applications for which obfuscation of some family of pseudo random functions is secure enough.

3 Theoretical results

The most important result concerning obfuscation is [1]. The essence of it is that there exists a family of programs for which every obfuscator fails completely but which is non-learnable. The obvious corollary is that general obfuscator that would work on every class of non-learnable programs does not exist.

Authors observe that most of interesting applications rely on the intuition that obfuscated program is a "virtual black box". That is, everything that one can efficiently calculate from obfuscated program, one should be able to efficiently compute given just oracle access to the program.

Their main result is that it's impossible to achieve obfuscation in that sense. Having access to some program that implements some function is always better (one can do more) than having oracle access to that function.

It does not directly mean that the obfuscation in some relaxed sense could not be possible. One could either relax the blackbox-requirement. Other alternative is to choose some smaller class and to prove that for this we can construct a blackbox. There already exists such attempts, namely regarding point functions [2] and some specific access control systems [3].

So, the theoretical state of art is that there is no known good general obfuscation method. There are ways and ideas what to study next. And there are some vague results in very specific areas or very specific security models.

4 Practical point of view

It's important to understand that in spite of lack of ideal obfuscator, obfuscation remains to be the most practical method to protect architecture independent code that is widespread nowadays, such as Java or .Net bytecode. There are obfuscating transformations that make the code harder to understand and sometimes this is enough.

So, although the theoretical base of obfuscation is quite weak, there are articles that try to describe, classify and evaluate different (practical) methods of obfuscation. Some of those works are said to be not theoretical enough, and some others claim clearly that having enough time and resources, every obfuscation (that are considered) is reversible. Most of those articles are written before formal proof of impossibility of obfuscation. One of most influential works is by Thomborson et al. [1]. They describe obfuscating transformations considering obfuscations on Java bytecode. This allows one to exploit different properties and relations of Java language and JVM bytecode such as the object-orientedness of Java (the fact that data and code is divided into classes) or as the different paradigms of JVM-assembler and Java language.

Obfuscating transformations. We call transformation $P \overset{\tau}{\rightarrow} P'$an obfuscating transformation if

- τ transforms source program P into obfuscated program P'
- P and P' have the same observable behaviour, except when program does not halt or yields an error.
- τ is effective, that is P' is more unintelligible than P

Observable behaviour is the work of program that is observable in the usual sense – having some specific input, the program gives out some specific answer. Obfuscated program might do some extra work as create some temporary files, but this should be hidden from user.

4.1 Classification of obfuscating transformations

There is a lot of different classifications but most cited work is [1]. They classify obfuscations based on the object of obfuscation. They observe three classes:

- **layout obfuscation.** This class consists of the simplest of obfuscations. Obfuscations that remove comments, remove indenting of source code, rename identifiers and also inline and outline functions are considered to be layout obfuscations.
- **control-flow obfuscation.** Most of the control-flow obfuscations (and some data obfuscations) depend on opaque predicates. Informally, opaque predicate is predicate whose value is known to obfuscator but is hard to evaluate for deobfuscator. One could have an opaque predicate that is always true. We could use such a predicate to create "false branches". It's important that opaque predicate has to be opaque to compiler as well, because otherwise it could be optimised out. Finding such predicates is primary task in control-flow obfuscation.
- **data obfuscation.** The main goal in data obfuscation is making understanding of the purpose and usage of data fields in program as hard as possible. There are several ways to do it: transformations in storing and encoding of data, aggregating data and ordering of data. These include obscure indexing of arrays, sharing the same memory slot for different purposes (changing the lifetime of variable and using it for different things at different moments or even coding multiple logical values into same field if value spaces allow it). There is also a little amount of information in access patterns and the ordering of variables (that usually corresponds to the declaration order in source file) that we might want to hide.

4.1.1 Subclasses of control-flow obfuscation

The goal of control-flow obfuscation is making it as hard as possible to understand the real control-flow in program using various methods. It appears that all methods divide into three subclasses:

- **control-flow aggregation.** Most important ways of aggregation are inlining, outlining, interleaving and cloning. The basic idea is to break the procedural abstraction that the programmer put into the program. So, the parts that programmer did put together are scattered around the code and parts that were separate before are integrated into one procedure.
- **control-flow computation.** These are divided into three categories. a) hiding the right control-flow with insignificant statements, such as opaque predicates that always are true. This category also contains dead code that is inserted just to confuse adversary. b) insertion of such objectcode that has no match in higher level language. There is a goto instruction in Java virtual machine assembler but there is no goto in Java language, for example. Also, table interpretation, one of the most effective obfuscating transformations is in that category. Table interpretation is about realising some part of the program in new language that is interpreted runtime. This is quite effective, but it has a performance penalty in order of magnitude, so one must consider using this technique carefully. And c) parallelising the program.
- **control-flow ordering**. Ordering transformations are a lot like aggregation transformations. The basic idea is that two actions that are close in source code are likely to be related. To hide such relations in the code, one wants to reorder declarations, blocks, statements etc.

It's important to remember that any control-flow obfuscation is potential overhead and one should use them wisely.

4.1.2 Additional options

As we can be quite sure that if one wants to obfuscate his/her program, then there is someone trying to deobfuscate it. Logical way to do it would be use some automatic deobfuscator that tries to detect and remove obvious obfuscations. So there are some **preventive transformations** that are designed to confuse the machine and not the human being. There are two kinds of preventive transformations. First kind is against some known techniques of deobfuscation. Others are designed to use known vulnerabilities in specific deobfuscators.

4.2 Attacks against obfuscation

As we are trying to protect some program, it's clear someone tries to deobfuscate it. So, it's wise to think about the ways one could deobfuscate our program.

Methods of deobfuscation by program analysis are either static or dynamic. Static analysis allows us to reconstruct control-flow graph of the program by exploring the behaviour of the program on various inputs. There are ways how to make program more resilient to static analysis. Dynamic analysis on the other hand is not about studying the representation of program but running it in some specific environment that allows us to watch programs real behaviour, values that emerge, (system) calls that are made and relations between data fields. It's clear that the code that was added by obfuscation will not be run by dynamic analysis and so the adversary won't spend it's time on it.

In general, almost every obfuscation is about adding extra code within non-necessary extra code. So, the adversary just has to simplify the program and remove parts that are never used. As adversary knows, that many of obfuscations use opaque predicates, it's first task is to identify and evaluate such predicates. It's harder if those predicates have structure that resembles the structure of those that are naturally found in the program.

To find code segments that are somewhat related, there is technique called slicing. Slicer divides program into small segments that are independent in some sense and should be good building blocks to reconstruct the original program. Of course, there are also ways to make slicer's life more complicated. Usual approach is to add fake relations between variables and code segments such as aliases for the same memory field an so on.

4.3 Effectiveness of obfuscation

Obviously there are better and not so good ways to obfuscate programs. Different articles offer different ways to evaluate different methods. [1] lists three of them, afterwards the same authors added fourth. The distinctive qualities are:

- **potency** expresses the clearlessness or how hard it is to understand the obfuscated program. Potency can be low, medium or high.
- **resilience** measures how good a transformation is against automated attack. It's really a combined measure of programmer effort and automated deobfuscator effort. Resilience is can be from trivial up to one-way.
- **cost** evaluates the extra resources one needs when applying a transformation. Cost can be dear (exponential growth in resource usage), costly (polynomial), cheap (linear) or free (constant).
- **stealth** says how easy it is to distinguish between original code and code added by transformation. Obviously it depends on program that is obfuscated. Naturally predicates that resemble those that are in original program are harder to detect either manually or automatically.

Generally those are modified versions of metrics known from software engineering disciplines that allow one estimate the complexity of programs. It's important to understand that these are good to compare complexity of two programs (in some specific sense) but it's hard to measure "absolute" complexity. This is so mainly because it's hard to estimate how complex something is for human mind. Some researchers have even done experiments using real crackers (or students) to get empirical estimations on some methods effectiveness.

References

[1] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudish, Amit Sahai, Salil Vadhan and Ke Yang. On the (im)possibility of obfuscating programs. Crypto 2001.

[1] C. Thomborson, C. Collberg, D. Low. A Taxonomy of Obfuscating Transformations. 1997.

[2] Hoeteck Wee. On Obfuscating Point Functions. STOC 2005.

[3] Benjamin Lynn, Manoj Prabhakaran, Amit Sahai. Positive Results and Techniques for Obfuscation. Eurocrypt 2004.

[4] Jarno Raid. Tarkvara sogastamine (Obfuscation of Software). Tartu 2005.