Deobfuscation is in NP

Andrew W. Appel*
Princeton University

August 21, 2002

Under some very general assumptions about how a program-obfuscation works, deobfuscation is NP-easy. While this does not immediately lead to a practical deobfuscation algorithm, it shows that deobfuscation is much easier than most other exact program analyses, such as the related problem of program optimization.

1 Introduction

There are two important problems in the security of mobile programs: protecting the host computer from the program, and protecting the program from the host computer. For the first problem, many successful techniques apply from virtual-memory protection to type-checking and even proof-carrying code. But the second problem seems harder: computing a result "in full sight" of the host computer while keeping the algorithms or intermediate results secret.

Protection of programs from hosts has many applications; a typical application is in "digital rights management," where it is desired (for example) to permit the decryption of a document for display on the screen but not for copying or printing. The display program must somehow hide the decryption key from the prying eyes of the host computer and operating system.

Every part of a typical personal computer is accessible to the person who has physical possession of it. This means the user can read the disk drive, can insert emulation layers to debug or analyze the execution of the program, and so on. Under such circumstances, protection is difficult indeed.

Some work in this area assumes the use of a secure processor (or coprocessor) that can do computations that are not physically accessible to the user [3, 2]. Such approaches can be made as strong as public-key encryption, and appear to have significant promise.

However, much of the work in program-protection is meant to be applicable on conventional, general-purpose PCs, and does not rely on special hardware. The algorithms used are generally called "code obfuscation;" they rearrange the protected program to make their control- and data-flow more difficult to understand. Collberg [1] presented one of the first results in systematic code obfuscation, and from his solution one can observe some important principles of code obfuscation:

- We assume that the obfuscation algorithm takes in a source program and a secret key, and produces an obfuscated program whose input/output behavior is identical to the source program. That is, $P = F(K,S)$ where F is the algorithm, K is the key, and S is the source program.

- We assume that the obfuscation algorithm itself is not secret. This assumption is analogous to the assumption made in modern cryptography, where encryption/decryption algorithms are assumed to be published, but the keys are kept secret; encryption must be secure even if the adversary knows the algorithm. This assumption is made because secrets inevitably leak. If a randomly chosen key leaks, then one can simply (and cheaply) generate a new key, but if an algorithm leaks, it's not easy to find a new algorithm. Also, it is assumed that many people will be using the algorithm (making it hard to keep secret) but few people will use any given key.

- The obfuscation algorithm should produce an obfuscated program P that is not too much bigger or smaller than the source program S. In this paper I will make the (generous) assumption that the size of the obfuscated program is at most polynomial in the size of the source, and vice versa. In real life, a constant factor is usually demanded.

- The length of the key K is polynomial in the length of S.

- The obfuscation algorithm F runs in polynomial time.

Collberg's algorithm works by intertwining some extra data-structure creation and manipulation instructions with the instructions of the source program. The data structure is a directed graph, and this graph is designed

*This work was supported by DARPA award F30602-99-1-0519.

to have some NP-complete property. Periodically (intertwined with program code), the graph is queried, and based on the query result the control flow of the original program is modified. The obfuscator knows which way the queries will come out; but because of the embedding of the NP-complete problem, it is difficult to tell from the examination of the obfuscated program which way the queries will come out. However, even though there is an NP-complete problem embedded, it is not entirely clear whether deobfuscation is really NP-hard.

2 Difficulty of program analyses

The deobfuscation problem is to simplify an obfuscated program, removing useless control- and data-flow. This is an example of a program analysis or program optimization. That is, ordinary optimizing compilers are interested in the same thing: to improve a program by removing useless work. Most of the analyses that optimizing compilers really want to do are Turing-complete. For example,

- Dead-code elimination wants to know, "does control-flow ever reach this point in the program?"
- Register allocation wants to know, "starting from this point in the program, can the value of variable x affect the eventual result computed?"
- Load/store scheduling wants to know, "are these pointers *aliased*, i.e. can they contain the same value?"

All of these problems reduce the halting problem, so they are all undecidable. Real compilers avoid the Turing tarpit by solving conservative approximations to these questions. However, these conservative approximations can never be *complete* – they never guarantee to find a nontrivial answer – and compilers cope with this by simply leaving the program (partially) unoptimized in such cases.

We are interested in a complete deobfuscator, one which positively finds a source program. At first glance, the deobfuscation problem – "is there a smaller program that does the same thing as this one" – appears to be undecidable. If we had a general deobfuscator D, we could take any instance P of the halting problem and feed it to the deobfuscator. If $D(P)$ is smaller than P, then compute $D(D(P))$. If (and only if) P does not halt, then this procedure must converge on a one-instruction infinite loop.

3 The nondeterministic algorithm

However, the deobfuscator has access not only to the obfuscated program P, but to the obfuscation algorithm F; it lacks only the key K.

Now, the deobfuscation algorithm $D(F,P)$ can be implemented in nondeterministic polynomial time:

Guess a source program S
Guess a key K
Compute $P' = F(K,S)$
Verify that $P' = P$.

Based on the assumptions, each step takes at most polynomial time in the size of P (or in the size of S). Actually, this holds even if one of the assumptions is relaxed: the obfuscation algorithm F could even be permitted to take nondeterministic polynomial time instead of deterministic polynomial time.

4 Impact of the result

Of course, the fact that deobfuscation is NP-easy does not immediately and constructively lead to useful deobfuscation programs. An NP upper bound is rather weak. But the result is still worrisome for obfuscation: in contrast to so many program analyses that are undecidable, to find that this one is not only decidable but in a complexity class almost within practical reach reduces the margin of security.

In practice, it is my suspicion that program obfuscation will not provide strong security in practice because the resources and techniques available to attackers are so numerous and powerful: debuggers, simulators, test-coverage tools, decompilers. Then, once the attacker has information about the algorithm F, it should be possible to make specialized execution-analysis tools tuned to F.

References

[1] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL '98: The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–196, January 1998.

[2] Paul England, John D. DeTreville, and Butler W. Lampson. Digital rights management operating system. U.S. Patent No. 6,330,670, December 2001. assigned to Microsoft Corporation.

[3] David Lie, Chandu Thekkath, Patrick Lincoln, Mark Mitchell, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS IX: Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, New York, November 2000. ACM Press.