On Software Protection Via Function Hiding

Tomas Sander and Christian F. Tschudin
Presented by Baogang Song
November 22, 1999

1 Introduction

If someday you find an algorithm that can solve a very important mathematical problem (e.g., a classical NP-hard problem) much faster than any other previously known algorithm. How can you turn this finding into a profitable business without revealing the algorithm? As the program generates a considerable workload you do not want everybody to send their problem instances to you. Instead, you would like to sell the software that your clients would run in the most autonomous way possible. However, this should not jeopardize your trade secret and should still let you control who licensed (and thus is allowed to run) the program. Hiding the algorithm and detecting the unlicensed copy are the primary goal of this paper.

Is that possible to distribute your program without the fear of disclosing the algorithm? There is a widespread belief in the mobile agent community that an entity which executes a given program has fully control over its execution, that the entity may potentially fully understand the program and therefore eventually can change it in any way it wants because Cleartext data can be read and changed; cleartext programs can be manipulated and cleartext messages, e.g., to the originator, can be faked. But there is no intrinsic reason why programs have to be executed in cleartext form: In the same sense that you can communicate some ciphermessage to another party without understand it, we can make the computer execute a ciperprogram without understanding it.

2 Encryption Protocols

2.1 Computing Encrypted Results

Let $E$ be a mechanism to encrypt a function $f$.

- Alice encrypts $f$ and creates a program $P(E(f))$.
- Alice sends her software $P(E(f))$ to interested parties e.g., Bob.
- Bob executes $P(E(f))$ at his input $x$ and send the result $y$ to Alice.
• Alice decrypts $y$, obtains $f(x)$ and sends this result back to Bob.

Notice that in the protocol above, Alice can charge the users on a per-usage basis. The decryption requests provide the nature hook for enforcing the collection of charges and could be handled by worldwide replicated trusted decryption and payment centers.

2.2 Fingerprinting Encrypted Functions and Their Results

Alice must deal with the situation that Bob could send the encrypted result, which is generated by unlicensed copy, to her in order to get the decrypted result. A very elegant solution is provided in this paper to fingerprint every (encrypted) result that each user generated so that each decryption request automatically identifies the program by which the result was obtained.

3 How to Hide Polynomials

3.1 Homomorphic Encryption Schemes

Definition 1. Let $R$ and $S$ be rings. We call an (encryption) function $E: R \rightarrow S$ additively homomorphic if there is an efficient algorithm PLUS to compute $E(x+y)$ from $E(x)$ and $E(y)$ that does not reveal $x$ and $y$.

Definition 2. Let $R$ and $S$ be rings. We call an (encryption) function $E: R \rightarrow S$ mixed multiplicatively homomorphic if there is an efficient algorithm MIXED-MULT to compute $E(xy)$ from $E(x)$ and $y$ that does not reveal $x$.

Lemma 1. An additive homomorphic encryption function on $\mathbb{Z}/n\mathbb{Z}$ is also mixed multiplicative homomorphic.

Proposition 1. Let $E: \mathbb{Z}/N\mathbb{Z} \rightarrow R$ be an additively homomorphic encryption scheme. Then we can hide polynomials $p$ with the help of $E$ in a program $\text{Prog}$. The program outputs $E(p(x))$ for an input $x$.

Using the homomorphic encryption scheme above one can encrypt a polynomial such that the resulting program effectively hides the function. There are several homomorphic encryption functions in the literature. In this paper the author used Goldwasser-Micali scheme: an additive homomorphic encryption scheme $E$ on $\mathbb{Z}/2\mathbb{Z}$ which applied to a one bit message. In the Goldwasser-Micali public key crypto system, Alice’s secret key are two large primes $P$ and $Q$. Her public key is the modulus $N = PQ$ and a quadratic non-residue $t$ modulo $N$ with Jacobi symbol 1. The plaintext 0 is encrypted by a random quadratic residue modulo $N$, 1 is represented by non-residue. The scheme is additive because the encrypted sum of two values $x$ and $y$ is obtained by multiplying their encrypted values modulo $N$ i.e., $E(x+y) = E(x)E(y)$. 

2
3.2 A Naive Protocol for Protecting Polynomials

Bob runs Alice’s program PROG on his private input $x$ and obtains $Prog(x) = E(f(x))$. He sends the result back to Alice for decryption. Alice decrypts the result, obtains $f(x)$ and sends the cleartext result to Bob.

The problem with this approach is that Bob might send the encrypted coefficients that he finds in the program Prog. After Alice decrypts them and sends them back, Bob can obtain the algorithm of the secret function.

3.3 A Modified Protocol for Protecting Polynomials

A better protocol is to make sure that Alice is able to detect if an element send to her was in fact produced as an output of her encrypted program. The key idea is that Alice hides additional polynomials besides the function $f$ that get simultaneously executed when PROG is run and that serve as checksums for her.

3.4 Fingerprinting Encrypted Functions

Encrypted functions offer an elegant new way to fingerprinting a program: versions for different users of the program can be encrypted with different encryption functions. This has the interesting side effect that the encrypted outputs are fingerprinting too. Alice has complete knowledge about which program copy the request for decryption has been produced.

4 Conclusion

Encrypted functions that can be executed without prior decryption give way to surprising solutions for seemingly unsolvable problems of software protection. Alice can give the program to Bob safely without fearing that the function is divulged. Nevertheless, among the other issues such as learning attack and efficiency, how to hide general programs is a big challenge. Considering only polynomial function encryption is a restriction. It will be interesting to extend these results to algebraic circuits and finally to Boolean circuits. If one succeeds to hide Boolean circuits, major problems of software protection would - at least theoretically - be solved for general programs because every Turing machine program can (for a fixed input size) be efficiently simulated by Boolean circuits.