CS 276 Lecture 30: Secret Sharing and Program Obsfucation

Scribe: Daniel Chen

May 9, 2006

1 Secret Sharing

Imagine a bank vault that can only be opened by a combination code s. The bank has n vice presidents and we want the vice presidents to be able to open the vault if necessary. This problem is called secret sharing. There are several forms of secret sharing:

1.1 1-out-of-n Secret Sharing

The obvious way of solving our problem would be to give the combination code s to all vice presidents. Then, any one of the vice presidents would have all the information necessary to open the vault. In practice, however, this is dangerous, because if one of the vice presidents' son gets kidnapped, and the ransom is the combination code s, the security of the vault can be easily compromised. This brings us to the next form of secret sharing:

1.2 n-out-of-n Secret Sharing

Alternatively, we could have a secret sharing scheme such that all n vice presidents are needed to open the vault, and that the vault cannot be opened when less than n vice presidents agree. This is called n-out-of-n secret sharing. One obvious way to do this would be to give s_1 to the first vice president, s_2 to the second, and so on, and let $s = s_1 \oplus s_2 \oplus \cdots \oplus s_n$. Alternatively, we could let $s \equiv s_1 + s_2 + \cdots + s_n \mod p$ for a prime p. However, this sort of scheme seems to be overly restrictive, because if one of the vice presidents is on vacation, then the vault cannot be opened.

1.3 2-out-of-n Secret Sharing

Therefore, we look for a compromise: We want to find scheme such thatonly two vice presidents are needed to open a vault, but any one of the vice presidents cannot open the vault by himself is called 2-out-of-n secret sharing. One way we might think of implementing this is by using a line $q(x) \equiv ax + s \mod p$

s be the y-intercept of our secret line and let $s_1, \ldots, s_n$ be points on the line where $s_i = q(i)$. Then, it is easy to see that one of the s_i gives no information about the y-intercept, while two of the s_i completely determines the secret line and therefore reveals the y-intercept.

1.4 t-out-of-n Secret Sharing

We might want to generalize this concept so that more than 2 vice presidents are needed to open a vault. We first introduce some notation:

Definition 1.1. *$s \rightarrow (s_1, \ldots s_n)$ means that s is the secret and $s_1, \ldots s_n$ are the share allocated to the vice presidents.*

The formal properties we want for t-out-of-n secret sharing are:

1. Recovery: Any party with size greater than or equal to t can recover s.
2. Secrecy: Any party with less than t members can learn nothing about s.

An easy modification of our 2-out-of-n example will give us these desired properties: Let us have a secret polynomial $q(x) \mod p$ of degree less than or equal to $t-1$. Let $s = q(0) \mod p$ and $s_i \equiv q(i) \mod p$. Then, if we have t or more points, we can recover $q(x) \mod p$, and thus recover $s = q(0) \mod p$. Otherwise, we have no information on $s = q(0) \mod p$. This particular form of t-out-of-n secret sharing is due to Shamir. Shamir's secret sharing scheme also has the following nice properties:

Theorem 1.2. *If $s \rightarrow (s_1, \ldots, s_n)$ and $u \rightarrow (u_1, \ldots, u_n)$ are Shamir sharings, then $s + u \rightarrow (s_1 + u_1, \ldots, s_n + u_n)$ is also a valid Shamir sharing.*

Thus, we can compute any linear function of a Shamir sharing. A important special case is the folowing:

Corollary 1.3. *If $s \rightarrow (s_1, \ldots, s_n)$ and $0 \rightarrow (u_1, \ldots, u_n)$ are Shamir sharings, then $s \rightarrow (s_1 + u_1, \ldots, s_n + u_n)$ is also a valid Shamir sharing.*

This fact is used when the sharing needs a "refreshing," that is, the sharing needs to be changed while the secret remains the same.

1.4.1 Lagrange Interpolation

So far, we haven't discussed how to recover a secret s from $(s_1, \ldots, s_n)$ in a Shamir sharing. Let I be a set of parties pooling their shares such that $|I| \geq t$ and define:

$$f(x) = \sum_{i \in I} s_i \prod_{j \in I - \{i\}} \frac{x - j}{i - j}$$

Now, lets look at the product $\prod_{j \in I - \{i\}} \frac{x-j}{i-j}$ in the summation and call it $\lambda_i(x)$. It is easy to check that:

$$\lambda_i(x) = \begin{cases} 1 & \text{if } x = i; \\ 0 & \text{if } x \in I - \{i\} \end{cases}$$

Thus, we also see that for all $i \in I$, $f(i) = s_i$. We see that $f(x)$ is a polynomial that goes through all the points (i, s_i) and that $f(x)$ is of the right degree, so it is the polynomial we want. Thus, to recover the key, we simply compute:

$$s = f(0) = \sum_{i \in I} s_i \prod_{j \in I - \{i\}} \frac{-j}{i - j}$$

If we refer to the constant product term as $\lambda_i = \prod_{j \in I - \{i\}} \frac{-j}{i-j}$, we can write this as:

$$s = f(0) = \sum_{i \in I} \lambda_i s_i$$

and thus recovery is linear as noted before.

1.4.2 Threshold El Gamal

A regular public key encryption scheme like El Gamal has a public key and a private key. Suppose we want to split the private key such that a group of recipients will only decrypt when at least t of them agree to decrypt. Recall that in ordinary El Gamal, we have a private key x and a public key $y = g^x$. An obvious way to do this would be to use the sharing $x \to (x_1, \ldots, x_n)$ and just compute x each time, but this is not secure because after one decryption, any one of the recipients would know how to decrypt future messages. In regular El Gamal, we encrypt by sending the pair $(c, d) = (g^r, y^r \cdot m)$ and we decrypt by computing $\frac{d}{c^x}$. For threshold El Gamal then, it is sufficient to be able to compute c^x. We use the sharing $x \to (x_1, \ldots, x_n)$. Suppose the set I where $|I| \geq t$ is participating in decryption. Then:

1. Participant i computes and broadcasts $z_i = c^{x_i}$.
2. Everyone computes the λ_i coefficients. (Recall that $x = \sum_{i \in I} \lambda_i x_i$.)

Then, everyone can compute:

$$z = \prod_{i \in I} z_i^{\lambda_i} = \prod_{i \in I} (c^{x_i})^{\lambda_i} = c^{\sum_{i \in I} \lambda_i x_i} = c^x$$

so we are done. Threshold cryptography is an active area of research.

2 Program Obsfucation

This is the second part of the lecture. Suppose we want to take a program, and transforming it in a way such that it still works the same, but looking at the source code does not help to learn about the original program in any way. Let us refer an obsfucated version of a program P as $\boxed{\text{P}}$. There are two properties that we want:

1. Behavioral Equivalence: $\forall$ efficient A, $A^P \sim A^{\boxed{\text{P}}}$.

2. Virtual Black Box: $\forall A \exists Sim \forall P, A(\boxed{\text{P}}) \sim Sim^P$.

One possible application of program obsfucation is to turn private key encryption into public key encryption. For example, let $P(x) = AES_k(x)$. Then $\boxed{\text{P}}$ is a public key and k is a private key.

Another application of program obsfucation is to only decrypt non-confidential messages. Suppose $P(c)$ decrypts c and returns the message only if it does not start out with "Confidential: ". Then, we could give $\boxed{\text{P}}$ to the public, and D_k to trusted people.

However, we end this lecture by noting that it is impossible to build program obsfucation. We give an example of a program that would be impossible to obsfucate. Suppose we choose α, β, γ randomly from $\{0,1\}^{256}$. We define:

$$L_{\alpha,\beta,\gamma}(P) = \begin{cases} \gamma & \text{if } P(\alpha) = \beta; \\ 0 & \text{otherwise.} \end{cases}$$

and

$$K_{\alpha,\beta}(P) = \begin{cases} \beta & \text{if } x = \alpha; \\ 0 & \text{otherwise.} \end{cases}$$

Then clearly, $L(K) = \gamma$. We also note that:

1. $L(\boxed{\text{K}}) = \gamma$.
2. $\boxed{\text{L}}(\boxed{\text{K}}) = \gamma$.
3. Therefore, given $\boxed{\text{L}}$ and $\boxed{\text{K}}$, we can learn γ.
4. However, for any simulator Sim, $Sim^{K,L}$ can learn nothing about γ. Therefore, program obsfucation is impossible.