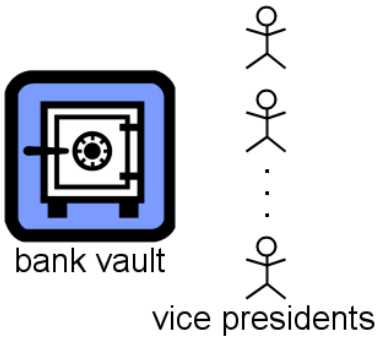
Lecture 30 — Fun Topics

1 Secret Sharing

1.1 Secret sharing background

Consider a bank vault locked with a combination lock, and n bank vice presidents that need to be able to open the vault. The combination to the lock is some number s.



There are a bunch of different ways we can give s to the VPs. We could:

Give s to each VP. This is called a 1-out-of-n access structure, because any 1 of the n VPs can access the vault. Some potential problems with this could be:

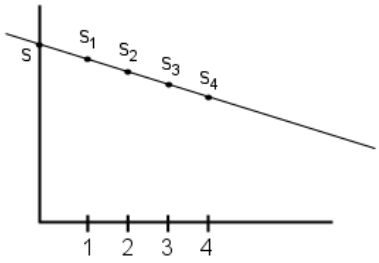
- Danger to the VPs' families. A would-be bank robber might kidnap the family of a VP and demand the VP turn over s to get them back.
- We might not be sure we can trust all the VPs.

The second issue subsumes the first; this is analagous to the privacy and coercion resistance properties desired in an electronic voting scheme, where we don't want anyone's vote to be revealed if they don't want it revealed, or even if they do.

Give a share of s to each VP so they can open the vault by pooling their knowledge. This is called an n-out-of-n access structure, because it takes all n of the n VPs to open the vault. Some possible implementations of this:

- Let the shares be random but subject to the constraint that their XOR is s.
- Let the shares be random but subject to the constraint that their sum (using modulo arithmetic) is s.

2-out-of-n. By letting the VPs' shares be distinct points on a line with y-intercept s, we let any two VPs open the vault while preventing any one from doing so.



We choose a prime p greater than n and greater than the maximum value in the set from which s is chosen. Then we randomly choose a polynomial

$$q(x) = ax + s \pmod{p}$$

either by randomly choosing an a or randomly choosing any s_i and then determining the implied a. s_i refers to the share given to the ith VP, and is calculated as

$$s_i = q(i).$$

The point associated with each share is then the pair (i, s_i) treated as x-y coordinates. These calculations must be performed by a trusted dealer in an initialization step.

t-out-of-n. In the most general case, we want the following security properties:

1. Robustness: any subset of at least t of the n parties can recover s efficiently; and
2. Secrecy: any subset of at most $t-1$ of the n parties can learn nothing about s. In other words, the values collectively possessed by any $t-1$ parties are independently distributed from s, so we have security regardless of the adversary's computing time.

In a t-out-of-n secret sharing scheme, we write the mapping from the secret s to the n shares as

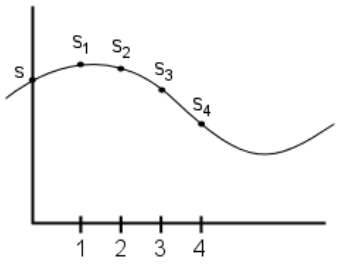
$$s \rightarrow (s_1, \ldots, s_n),$$

and for a valid secret sharing scheme this mapping must be efficiently computable.

1.2 Shamir's t-out-of-n secret sharing scheme

The t-out-of-n secret sharing scheme introduced by Adi Shamir in his 1979 paper "How to Share a Secret" is a generalization of the above line-based 2-out-of-n scheme. A line, which is fully determined by any two distinct points, is a polynomial of degree 1. In fact any polynomial of degree $t-1$ is fully determined by any t distinct points. Furthermore, when we are working in a finite field like $\mathbb{Z}_p$, it is possible, given the value of a $t-1$-degree polynomial at each of any t distinct field elements, to efficiently interpolate the value of the polynomial at any other field element.

Thus the generalization of the line method that Shamir proposed was to use a $t-1$ degree polynomial $q(x)$ with $q(0) = s \pmod{p}$.



1.2.1 Secrecy

This scheme is secure because for any subset of fewer than t points, any hypothesized y-intercept for the $t-1$-degree polynomial is consistent with the observed points, and equally likely. Therefore no information is gained from any such subset.

1.2.2 Robustness (Lagrange interpolation)

Given any set I of parties pooling their shares, where $|I| \geq t$ and I is represented as a set of participant indices, the polynomial q can be efficiently recovered using Lagrange interpolation, the formula for which is

$$q(x) = \sum_{i \in I} s_i \underbrace{\prod_{j \in I - \{i\}} \frac{x-j}{i-j}}_{\lambda_i(x)}. \tag{1}$$

To understand this formula, consider the product term, which we call $\lambda_i(x)$. It is a function of i and x, as suggested by the notation. The value of $\lambda_i(x)$ for $x \in I$ is

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

because in the top case, the numerator and the denominator of the fraction are always the same, yielding 1, and in the bottom case, the numerator for one of the terms of the product is 0, yielding 0. As a result, if we evaluate $q(x)$ on an x in I, the sum in Equation (1) expands to

$$0 \cdot s_1 + 0 \cdot s_2 + \cdots + 1 \cdot s_x + \cdots = s_x,$$

as it should be. If we evaluate q on 0 to get s, the interpolation expression simplifies to

$$\begin{aligned} s &= q(0) = \sum_{i \in I} s_i \underbrace{\prod_{j \in I - \{i\}} \frac{-j}{i-j}}_{\lambda_i \equiv \lambda_i(0)} \\ &= \sum_{i \in I} \lambda_i s_i \end{aligned} \tag{2}$$

Note that $\lambda_i(0)$, or λ_i as we call it, is just a function of what indices are in the index set I, presumably public information. Thus s is a *linear* function of the s_is, with publicly known coefficients. This linearity has many useful implications.

1.2.3 Linearity of Lagrange interpolation

Because of the linearity of Lagrange interpolation as mentioned above, if we have a valid Shamir sharing

$$s \to (s_1, \ldots, s_n)$$

and also a valid Shamir sharing

$$t \to (t_1, \ldots, t_n),$$

then it is the case that

$$s + t \to (s_1 + t_1, \ldots, s_n + t_n),$$

or in other words if each party adds their share of s and their share of t, then they have a share of $s + t$. This property holds not just for the sum of two secret/sharing pairs, but in fact any linear function of any number of secret/sharing pairs. One useful application is reblinding sharings: if

$$\begin{aligned} s &\to (s_1, \ldots, s_n) \\ 0 &\to (t_1, \ldots, t_n), \end{aligned}$$

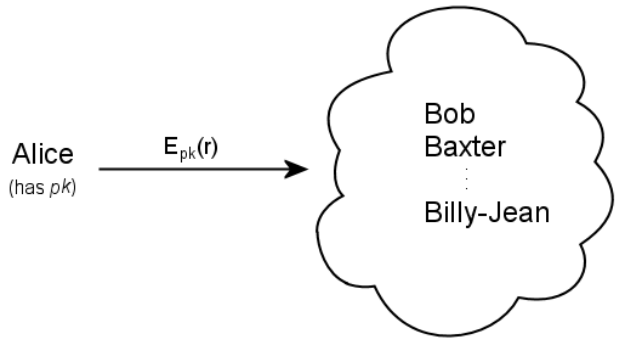
then $s \to (s_1+t_1, \ldots, s_n+t_n)$ is a valid sharing of s, independent of the previous sharing $(s_1, \ldots, s_n)$.

- Imagine you are one of the vice-presidents of the bank and you want to go on vacation, leaving your key share with an underling so your underling can act as your proxy and assist in opening the vault in your absence. However, when you return, you want to revoke the underling's ability to open the vault, so when you get back you reblind the sharing of the vault combination and distribute that reblinding, but not to the underling. You tell the other VPs to forget their old shares, making your underling's share the only share of its set and thereby useless. However, you can't force the other VPs to forget their old shares, so $t - 1$ of them could collude with the underling to open the vault.
- If you have a cluster of machines that share some access code, the sharing could be reblinded daily, and since it is possible to ensure that uncompromised machines really do erase old shares, this method could guarantee that an adversary gains nothing as long as there is no day on which at least t machines are compromised *at the same time*.

1.3 Threshold El Gamal

We might want to send an encrypted message, in a public key setting, to group of n recipients such that they can decrypt the message if and only if at least t of them work together. The naïve method of just having the recipients share a private key with Shamir secret sharing would work, but only as a one-shot deal. After one message, all the members of the recipient group would know the private key, and thenceforth any one of them would be able to decrypt incoming messages by themself, which is not what we want.

However, we can take advantage of mathematical properties of El Gamal to create a scheme that does *something like* sharing the private key, but in the shared decryption process, the recipients



learn the message but nothing more—in particular, not the private key. The setting here can be visualized as one sender using a public key to send an encrypted message to a "cloud" of recipients.

The scheme starts out with a trusted dealer choosing a private key x, a public key $y = g^x$, and a sharing of the private key $x \to (x_1, \ldots, x_n)$, and then giving the shares to the participants. As in normal El Gamal, a ciphertext $E_{pk}(m)$ is a pair $(c, d) = (g^r, y^r \cdot m)$ for some r. As a reminder, normal El Gamal decryption is

$$\frac{d}{c^x} = \frac{y^r m}{g^{rx}} = \frac{y^r m}{y^r} = m.$$

We want to use the shares of the private key x to jointly raise c to the xth power without learning anything about x, and we use the fact that share recovery is linear. We want perfect secrecy against $t-1$ malicious parties, but if there are t or more than all bets are off.

For a set of participant indices I, decryption works like this.

1. Each participant i computes and broadcasts (within the group of participants) $z_i = c^{x_i}$.
2. Everyone computes, on their own, the λ-coefficients of Lagrange interpolation, which they can do by just knowing the indices in I. Remember from Equation (2) that $x = \sum_{i \in I} \lambda_i x_i$.
3. Everyone computes, on their own:

$$\begin{aligned} z &\equiv \prod_{i \in I} z_i^{\lambda_i} \\ &= \prod_{i \in I} (c^{x_i})^{\lambda_i} \\ &= c^{\sum_{i \in I} \lambda_i x_i} \\ &= c^x \end{aligned}$$

In an honest but curious model, that is, if everyone followed the protocol (if not, any wrong z_i could wreck everything, though we could require z_is be accompanied by zero-knowledge proofs), the above equalities hold.

A further improvement can be made to get rid of the reliance on a trusted dealer. Everyone picks a random value and then acts as a dealer, distributing shares of that value to all participants.

Thus each participant ends up with a share of n different numbers, each generated by a different participant. Each participant then adds all the shares they've received, and the result is that everyone has a share of one random value: the sum of everyone's values. If at least one person chose their value truly at random, and distributed shares of it correctly (for which we can use zero-knowledge proofs) then the sum is in fact random.

2 Software Obfuscation

2.1 The goal

Sometimes it's desirable to obfuscate software so hard-to-disassemble binaries can be distributed, or source code can be distributed without revealing any "secret sauce" used in the program. The goal is to transform a program so it can be run, but that's all that can be done with it.

Restricting our discussion for simplicity's sake to deterministic, stateless programs that always terminate, let us signify obfuscation as a mapping from program P to obfuscated program $\boxed{P}$. This process must satisfy the following two conditions:

1. It must be behavior-preserving ($\boxed{P}$ must compute the same function as P), and
2. $\boxed{P}$ must be like a black box (one can only supply input and see output), which is the reason for the letter-in-box notation.

Let us in turn make these two conditions more precise.

2.1.1 Behavioral equivalence

Let us say that two programs P and Q are behaviorally equivalent, $P \equiv Q$, if no efficient algorithm can distinguish between the two given only oracle access. In other words,

$$\forall \text{ efficient } A, A^P \sim A^Q.$$

Here, oracle access means input-output access only. For P and Q to be considered behaviorally equivalent, their running times and sizes may be different, but by at most a polynomial factor in each case. Our behavior-preserving condition above then becomes $P \sim \boxed{P}$.

2.1.2 Virtual black box

The virtual black box condition means that anything that can be learned from looking at $\boxed{P}$ can be learned by just having oracle access to P. Note that there are two ways to look at a program, as a function and as a string of bits, and here we are looking at $\boxed{P}$ as a string of bits and P as a function.

Formally, for an obfuscator to satisfy this condition we require that

$$\forall A.\ \exists Sim.\ \forall P\ A(\boxed{\mathbb{P}}) \sim Sim^P,$$

or if we restrict A to being a predicate,

$$\forall A.\ \exists Sim.\ \forall P\ \left|Pr[A(\boxed{\mathbb{P}}) = 1] - Pr[Sim^P = 1]\right| \leq \epsilon$$

for some negligible ϵ. Sim is a simulator.

2.2 Hypothetical applications of software obfuscation

Consider a program $P(x)$ that calculates the function $AES_k(x)$; in other words, the key k is hard-coded into the program. Then we have a public key encryption scheme with $\boxed{\mathbb{P}}$ as the public key and k as the private key, because anyone can encrypt using $\boxed{\mathbb{P}}$—but not decrypt—and k can be used to decrypt (and encrypt too, which breaks the analogy somewhat, but in the public key setting the public key is assumed to be public so anyone can encrypt.) So if obfuscation is possible, then in the same way, any private-key encryption algorithm can be efficiently converted into a public-key encryption algorithm.

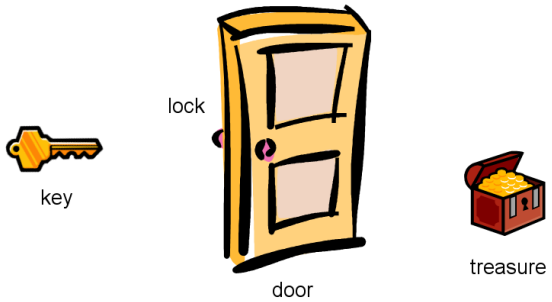
Likewise, we could use obfuscation to convert online programs into offline ones. For example, if I go on vacation and want to let my assistant decrypt messages to me unless they are marked confidential, I could leave a program running on my computer—which has my private key on it—that provides that service. However, if I wanted to just give a program to my assistant without leaving my computer on, I could write a program

$$P(c) = \begin{cases} - & \text{if } D_k(c) \text{ starts with ``confidential''} \\ D_k(c) & \text{else} \end{cases}$$

and give $\boxed{\mathbb{P}}$ to my assistant.

2.3 Impossibility of obfuscation

As one might suspect from the promised efficient conversion of any private key encryption algorithm into a public key encryption algorithm, obfuscation under the conditions given above is *impossible*.



Consider the above situation, in which all the objects shown are actually bit strings. Let α, β, and γ be 256-bit strings chosen uniformly at random. γ is a secret (the "treasure"). The key is a program, and the lock is also a program that runs the key program to determine if it's valid. Let the lock program l be defined as calculating the function

$$Lock_{\alpha,\beta,\gamma}(P) = \begin{cases} \gamma & \text{if } P(\alpha) = \beta \\ 0 & \text{else} \end{cases}$$

and let the key program k be defined as calculating the function

$$Key_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0 & \text{else} \end{cases} .$$

Note that $l(k) = \gamma$. Now suppose it were possible to obfuscate k and l to obtain $\boxed{k}$ and $\boxed{l}$.

1. $\boxed{k}$ has the same behavior as k, so $l(\boxed{k}) = \gamma$.
2. Likewise, $\boxed{l}$ has the same behavior as l, so $\boxed{l}(\boxed{k}) = \gamma$.
3. So given $\boxed{k}$ and $\boxed{l}$, one can learn γ.
4. However, a simulator with only oracle access to k and l *cannot learn* γ. In particular, one needs the "source code" of k to give to l. And any query directly to the k oracle returns 0, unless one by chance hits the correct value of α hard-coded into k, which happens with negligible probability. Likewise, any query to the l oracle returns 0, unless one by chance gives it a program which, on input α, returns the correct value of β hard coded into l, which also happens with negligible probability.

The fact that we can learn something from $\boxed{k}$ and $\boxed{l}$ that a simulator with oracle access to k and l cannot learn means that the virtual black box property is not satisfied, contradicting the assumption that k and l were obfuscated in compliance with the two conditions mentioned earlier.

This is an example of a negative result in cryptography, which is perhaps more depressing than a positive result, but is also very useful, because it lets us know not to waste time trying to create perfect obfuscation. By building up many positive and negative results, we gradually narrow down our knowledge of what exactly is cryptographically possible.