Lecture Notes, Week 11

1 Feige-Fiat-Shamir Signatures

A signature scheme has a lot in common with the "non-interactive interactive" proofs introduced in lecture notes week 10. In both cases, there is only a one-way communication from Alice to Bob. Alice signs a message and sends it to Bob. Bob then verifies it without further interaction with Alice. If Bob hands the message to Carol, then Carol can also verify that it was signed by Alice.

Not surprisingly, the "non-interactive interactive proof" ideas can be used to turn the Feige-Fiat-Shamir authentication protocol of lecture notes week 10 into a signature scheme. The signature scheme we present here is based on a slightly simplified version of the aforementioned protocol in which all of the v_i's in the public key are quadratic residues, and n is not required to be a Blum integer, only a product of two distinct odd primes. The public verification key is $(n, v_1, \ldots, v_k)$, and the private signing key is $(n, s_1, \ldots, s_k)$, where $v_j = s_j^{-2} \bmod n$ $(1 \le j \le k)$.

To sign a message m, Alice simulates t rounds of the protocol in parallel. She first chooses random $r_1, \ldots, r_t \in \mathbf{Z}_n - \{0\}$ and computes

$$x_i = r_i^2 \bmod n \ (1 \le i \le t).$$

Next she computes $u = H(mx_1 \cdots x_t)$, where H is a suitable cryptographic hash function. She chooses $b_{1,1}, \ldots, b_{t,k}$ according to the first tk bits of u, that is,

$$b_{i,j} = u_{(i-1)*k+j} \ (1 \le i \le t, \ 1 \le j \le k).$$

Finally, she computes

$$y_i = r s_1^{b_{i,1}} \cdots s_k^{b_{i,k}} \bmod n \ (1 \le i \le t).$$

The signature is

$$s = (b_{1,1}, \ldots, b_{t,k}, y_1, \ldots, y_t).$$

To verify the signed message (m, s), Bob computes

$$z_i = y_i^2 v_1^{b_{i,1}} \cdots v_k^{b_{i,k}} \bmod n \ (1 \le i \le t).$$

Bob checks that each $z_i \neq 0$ and that $b_{1,1}, \ldots, b_{t,k}$ are equal to the first tk bits of $H(mz_1 \cdots z_t)$.

When both Alice and Bob are honest, it is easily verified that $z_i = x_i$ $(1 \le i \le t)$. In that case, Bob's checks all succeed since $x_i \neq 0$ and $H(mz_1 \cdots z_t) = H(mx_1 \cdots x_t)$.

To forge Alice's signature, an impostor must find $b_{i,j}$'s and y_i's that satisfy the equation

$$b_{1,1} \ldots b_{t,k} \preceq H(m(y_1^2 v_1^{b_{1,1}} \cdots v_k^{b_{1,k}} \bmod n) \ldots (y_t^2 v_1^{b_{t,1}} \cdots v_k^{b_{t,k}} \bmod n)).$$

where "$\preceq$" means string prefix. It is not obvious how to solve such an equation without knowing a square root of each of the v_i^{-1}'s and following essentially Alice's procedure.

2 Secret Splitting

2.1 Simple two-part secret splitting

There are many situations in which one wants to grant access to a resource only if a sufficiently large group of agents cooperate. For example, a store safe might require both the manager's key and the armored car driver's key in order to be opened. This protects the store against a dishonest manager or armored car driver, and it also prevents an armed robber from coercing the manager into opening the safe. A similar 2-key system is used for safe deposit boxes in banks.

We might like to achieve the same properties for cryptographic keys or other secrets. For example, if k is the secret decryption key for a cryptosystem, one might wish to split k into two *shares* k_1 and k_2. By themselves, neither k_1 nor k_2 reveals any information about k, but when suitably combined, k can be recovered. A simple way to do this is to choose k_1 uniformly at random and then let $k_2 = k \oplus k_1$. Both k_1 and k_2 are uniformly distributed over the key space and hence give no information about k. However, combined with XOR, they reveal k, since $k = k_1 \oplus k_2$.

Indeed, the one-time pad cryptosystem of lecture notes week 1 can be viewed as an instance of secret splitting. Here, Alice's secret is her message m. The two shares are the ciphertext c and the key k. Neither by themselves gives any information about m, but together they reveal $m = k \oplus c$.

2.2 Multiple shares

Secret splitting generalizes to more than two shares. Imagine a large company that restricts access to important company secrets to only its five top executives, say the president, vice-president, treasurer, CEO, and CIO. They don't want any executive to be able to access the data alone since they are concerned that an executive might be blackmailed into giving confidential data to a competitor. On the other hand, they also don't want to require that all five executives get together to access their data, both because this would be cumbersome and also because they worry about the death or incapacitation of any single individual. They decide as a compromise that any three of them should be able to access the secret data, but not one or two of them operating alone.

A (τ, k) *threshold secret splitting scheme* splits a secret s into shares $s_1, \ldots, s_k$. Any subset of τ or more shares allows s to be recovered, but no subset of shares of size less than τ gives any information about s.

Shamir's scheme

Shamir proposed a threshold scheme based on polynomials. A *polynomial* of *degree* d is an expression

$$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_d x^d.$$

where $a_d \neq 0$. The numbers $a_0, \ldots, a_d$ are called the *coefficients* of f. A polynomial can be simultaneously regarded as a function and as an object determined by its vector of coefficients.

Interpolation is the process of finding a polynomial that goes through a given set of points.

Fact Let $(x_1, y_1), \ldots, (x_k, y_k)$ be points, where all of the x_i's are distinct. There is a unique polynomial $f(x)$ of degree at most $k - 1$ that passes through all k points, that is, for which $f(x_i) = y_i$ $(1 \leq 1 \leq k)$.

f can be found using Lagrangian interpolation. This statement generalizes the familiar statement from high school geometry that two points determine a line.

Interpolation also works over finite fields, for example, $\mathbf{Z}_p$ for prime p. That is, any k points with distinct x coordinates determine a unique polynomial of degree at most $k-1$ over $\mathbf{Z}_p$. Of course, we must have $k \leq p$ since $\mathbf{Z}_p$ has only p distinct coordinate values in all.

Here's how Shamir's (τ, k) secret splitting scheme works. Let Alice (also called the *dealer*) have secret s. She constructs a polynomial of degree at most $\tau - 1$ as follows: She sets $a_0 = s$, and she chooses $a_1, \ldots, a_{\tau-1} \in Z_p$ at random. Share s_i is the point (x_i, y_i), where $x_i = i$ and $y_i = f(i)$ $(1 \leq i \leq k)$[1].

Theorem 1 *s can be reconstructed from any set T of τ or more shares.*

Proof: Suppose $s_{i_1}, \ldots, s_{i_\tau}$ are τ distinct shares in T. By interpolation, there is a unique polynomial $g(x)$ of degree $d \leq \tau - 1$ that passes through these shares. By construction of the shares, $f(x)$ also passes through these same shares; hence $g = f$ as polynomials. In particular, $g(0) = f(0) = s$ is the secret. ■

Theorem 2 *Any set T' of fewer than τ shares gives no information about s.*

Proof: Let $T' = \{s_{i_1}, \ldots, s_{i_r}\}$ be a set of $r < \tau$ shares. There are in general many polynomials of degree $\leq \tau - 1$ that interpolate the points in T'. In particular, for each $s' \in \mathbf{Z}_p$, there is a polynomial $g_{s'}$ that interpolates the shares in $T' \cup \{(0, s')\}$. Each of these polynomials passes through all of the shares in T', so each is a plausible candidate for f. Moreover, $g_{s'}(0) = s'$, so each s' is a plausible candidate for the secret s. One can show further that the number of polynomials that interpolate $T' \cup \{(0, s')\}$ is the same for each $s' \in \mathbf{Z}_p$, so each possible candidate s' is equally likely to be s. Hence, the shares in T' give no information at all about s. ■

2.3 Extensions

Several variations on secret sharing have been studied. I mention two briefly but do not go into details.

Verifiable secret sharing. A *dealer* has a secret s which she wishes to share with a number of *players*. The dealer can of course always lie about the true value of her secret, but, as with bit commitment, the players want assurance that their shares do in fact code a unique secret. That is, whenever sufficiently many shares are assembled to reconstruct the secret, the same secret s is recovered, no matter which shares are used. In Shamir's (τ, k) threshold scheme, this will be true only if all of the shares lie on a single polynomial of degree at most $k - 1$. However, if the dealer is dishonest and gives bad shares to some of the players, the resulting shares might not lie on any polynomial of degree $k - 1$ or smaller. The players have no way to discover this until later when they try to reconstruct s.

In verifiable secret sharing, the sharing phase is an active protocol involving the dealer and all of the players. At the end of this phase, either the dealer is exposed as being dishonest, or all of the players end up with shares that are consistent with a single secret. Needless to say, protocols for verifiable secret sharing are quite complicated.

[1] $f(i)$ is the result of evaluating the polynomial f at the value $x = i$. Here we assume all arithmetic is over the field $\mathbf{Z}_p$, so we omit explicit mention of $\bmod p$.

Fault tolerance. Even if the dealer is assumed to be honest, there is still the problem of actively dishonest players. With Shamir's scheme, a share that just disappears does not prevent the secret from being reconstructed, as long as enough valid shares remain. But if a player lies about his share and presents a corrupted share, then that share might be used by the other players in reconstructing an incorrect value for the secret. A fault-tolerant secret sharing scheme should allow the secret to be correctly reconstructed, even in the face of a certain number of corrupted shares.

Of course, it may be desirable to have schemes that can tolerate dishonesty in both dealer and a certain number of players. The interested reader is encouraged to explore the extensive literature on this subject.

3 Bit-Commitment Problem

Alice and Bob want to play a game over the internet. Alice says, "I'm thinking of a bit. If you guess my bit correctly, I'll give you $10. If you guess wrong, you give me $10." Bob says, "Ok, I guess zero." Alice replies, "Sorry, you lose. I was thinking of one."

While this game may seem fair on the surface, there is nothing to prevent Alice from changing her mind after Bob makes his guess. Even if Alice and Bob play the game face to face, they still must do something to commit Alice to her bit before Bob makes his guess. For example, Alice might be required to write her bit down on a piece of paper and seal it in an envelope. After Bob makes his guess, he opens the envelope and knows whether he has won or lost. The act of writing down the bit *commits* Alice to that bit, even though Bob doesn't learn its value until later.

The *bit-commitment problem* is to implement an electronic form of sealed envelope called a *commitment* or *blob* or *cryptographic envelope*. Intuitively, a blob has two properties: (1) It is not possible to see the bit inside the blob without opening it. (2) It is not possible to change the bit inside the blob, that is, the blob cannot be opened in two different ways to reveal two different bits.

A blob is produced by a protocol $\mathbf{commit}(b)$ between Alice and Bob. We assume that b is initially private to Alice. At the end of the commit protocol, Bob has a blob c containing Alice's bit b, but he should have no information about b's value. Later, Alice and Bob can run a protocol $\mathbf{open}(c)$ to reveal the bit contained in c.

Alice and Bob do not trust each other, so each wants protection from cheating by the other. Alice wants to be sure that Bob cannot learn b after running $\mathbf{commit}(b)$, even if he misbehaves during the protocol. Bob wants to be sure that any successful run of $\mathbf{open}(c)$ reveals the same bit b', so no matter what Alice does. Note that we do *not* require that Alice tell the truth about her private bit b. A dishonest Alice can always pretend her bit was $b' \neq b$ when producing c. But if she does, c can only be opened to b', not to b.

These ideas should become clearer in the protocols below.

3.1 Commitment using symmetric cryptography

A naïve way to use a symmetric cryptosystem for bit commitment is for Alice to commit b by encrypting it with a private key k to get a blob $c = E_k(b)$. She later opens it using the decryption function $D_k(c)$. Unfortunately, Alice can easily cheat if she can find a "colliding triple" (c, k_0, k_1) with the properties that $D_{k_0}(c) = 0$ and $D_{k_0}(c) = 1$. She just "commits" by sending c to Bob. Later, she can choose whether to open it to 0 or to 1 by sending Bob k_0 or k_1. This isn't just a hypothetical problem. Suppose Alice uses the most secure cryptosystem of all, a one-time pad, so $D_k(c) = c \oplus k$. Then she can easily find a colliding triple by choosing $k_0 = c$ and $k_1 = c \oplus 1$.

The protocol of Figure 1 tries to make it harder for Alice to cheat by making it possible for Bob to detect most bad keys.

	Alice		Bob
	To **commit**(b):		
1.		$\xleftarrow{r}$	Choose random string r.
2.	Choose random key k.		
	Compute $c = E_k(r \cdot b)$.	$\xrightarrow{c}$	c is commitment.
	To **open**(c):		
3.	Send k.	$\xrightarrow{k}$	Let $r' \cdot b' = D_k(c)$. Check $r' = r$. b' is revealed bit.

Figure 1: Bit commitment using cryptosystem.

For many cryptosystems (e.g., DES), this protocol does indeed prevent Alice from cheating, for she will have difficulty finding any two keys k_0 and k_1 such that $E_{k_0}(r \cdot 0) = E_{k_1}(r \cdot 1)$. However, for the one-time pad cryptosystem, she can cheat as before: She just takes c to be random and lets $k_0 = c \oplus (r \cdot 0)$ and $k_1 = c \oplus (r \cdot 1)$. Then $D_{k_b}(c) = r \cdot b$ for $b \in \{0, 1\}$, so the revealed bit is 0 or 1 depending on whether Alice sends k_0 or k_1 in step 3.

We see that not all secure cryptosystems have the properties we need in order to make the protocol of Figure 1 secure. We need a property analogous to the strong collision-free property for hash functions.

3.2 Commitment using hash functions

The analogy between bit commitment and hash functions described above suggests a bit-commitment scheme based on hash functions, as shown in Figure 2.

	Alice		Bob
	To **commit**(b):		
1.		$\xleftarrow{r_1}$	Choose random string r_1.
2.	Choose random string r_2.		
	Compute $c = H(r_1 r_2 b)$.	$\xrightarrow{c}$	c is commitment.
	To **open**(c):		
3.	Send r_2.	$\xrightarrow{r_2}$	Find $b' \in \{0, 1\}$ such that $c = H(r_1 r_2 b')$. If no such b', then fail. Otherwise, b' is revealed bit.

Figure 2: Bit commitment using hash function.

The purpose of r_2 is to protect Alice's secret bit b. To find b before Alice opens the commitment, Bob would have to find r_2' and b' such that $H(r_1 r_2' b') = c$. This is akin to the problem of inverting H and is likely to be hard, although the one-way property for H is not strong enough to imply this. On the one hand, if Bob succeeds in finding such r_2' and b', he has indeed inverted H, but he does

so only with the help of r_1—information that is not generally available when attempting to invert H.

The purpose of r_1 is to strengthen the protection that Bob gets from the hash properties of H. Even without r_1, the strong collision-free property of H would imply that Alice cannot find c, r_2, and r_2' such that $H(r_2 0) = c = H(r_2' 1)$. But by using r_1, Alice would have to find a new colliding pair for each run of the protocol. This protects Bob by preventing Alice from exploiting a few colliding pairs for H that she might happen to discover.

3.3 Commitment using pseudorandom sequence generators

A pseudorandom sequence generator (PRSG) maps a "short" random seed to a "long" pseudorandom bit string. For a PRSG to be cryptographically strong, it must be difficult to correctly predict any generated bit, even knowing all of the other bits of the output sequence. In particular, it must also be difficult to find the seed given the output sequence, since if one knows the seed, then the whole sequence can be generated. Thus, a PRSG is a one-way function and more. While a hash function might generate hash values of the form yy and still be strongly collision-free, such a function could not be a PRSG since it would be possible to predict the second half of the output knowing the first half.

I am being intentionally vague at this stage about what "short" and "long" mean, but intuitively, "short" is a length like we use for cryptographic keys—long enough to prevent brute-force attacks, but generally much shorter than the data we want to deal with. Think of "short"=128 or =256 and you'll be in the right ballpark. By "long", we mean much larger sizes, perhaps thousands or even millions of bits. In practice, we usually thing of the output length as being variable, so that we can request as many output bits from the generator as we like and it will deliver them. Also, in practice, the bits are generally delivered a block at a time rather than all at once, so we don't even need to announce in advance how many bits we want but can go back as needed to get more.

There are many ways to use a PRSG G for bit commitment. One such way is shown in Figure 3. Here, ρ is a security parameter that controls the probability that a cheating Alice can fool Bob. We let $G_\rho(s)$ denote the first ρ bits of $G(s)$.

	Alice		Bob
	To **commit**(b):		
1.		$\xleftarrow{r}$	Choose random string $r \in \{0,1\}^\rho$.
2.	Choose random seed s. Let $y = G_\rho(s)$. If $b = 0$ let $c = y$. If $b = 1$ let $c = y \oplus r$.	$\xrightarrow{c}$	c is commitment.
	To **open**(c):		
3.	Send s.	$\xrightarrow{s}$	Let $y = G_\rho(s)$. If $c = y$ then reveal 0. If $c = y \oplus r$ then reveal 1. Otherwise, fail.

Figure 3: Bit commitment using PRSG.

Assuming G is cryptographically strong, then c will look random to Bob, regardless of the value of b, so he will be unable to get any information about b.

The purpose of r is to protect Bob against a cheating Alice. Alice can cheat if she can find a triple (c, s_0, s_1) such that s_0 opens c to reveal 0 and s_1 opens c to reveal 1. Such a triple must satisfy the following pair of equations:

$$\left.\begin{array}{rcl} c & = & G_\rho(s_0) \\ c & = & G_\rho(s_1) \oplus r. \end{array}\right\} \tag{1}$$

It is sufficient for her to solve the equation

$$r = G_\rho(s_0) \oplus G_\rho(s_1) \tag{2}$$

for s_0 and s_1 and then choose $c = G_\rho(s_0)$.

One might ask why Bob needs to choose r? Why can't Alice choose r, or why can't r be fixed to some constant? If Alice chooses r, then she can easily solve (2) and cheat. If r is fixed to a constant, then if Alice ever finds a triple (c, s_0, s_1) satisfying (1), she can fool Bob every time. While finding such a pair would be difficult if G_ρ were a truly random function, any specific PRSG might have special properties, at least for a few seeds, that would make this possible. For example, suppose $r = 1^\rho$ and $G_\rho(\neg s_0) = \neg G_\rho(s_0)$ for some s_0. Then (2) could be solved by taking $s_1 = \neg s_0$. By having Bob choose r at random, r will be different each time (with very high probability), and a successful cheating Alice would be forced to solve (1) in general, not just for one special case.

4 Bit-Commitment Schemes

The three bit-commitment protocols of the previous section all have the same form. We abstract from these protocols a cryptographic primitive, called a *bit-commitment scheme*, which consists of a pair of *key spaces* $\mathcal{K}_\mathcal{A}$ and $\mathcal{K}_\mathcal{B}$, a *blob space* $\mathcal{B}$, a *commitment* function

$$\mathbf{enclose} : \mathcal{K}_\mathcal{A} \times \mathcal{K}_\mathcal{B} \times \{0,1\} \to \mathcal{B},$$

and an *opening* function

$$\mathbf{reveal} : \mathcal{K}_\mathcal{A} \times \mathcal{K}_\mathcal{B} \times \mathcal{B} \to \{0,1,\phi\},$$

where ϕ means "failure". We say that a blob $c \in \mathcal{B}$ *contains* $b \in \{0,1\}$ if $\mathbf{reveal}(k_A, k_B, c) = b$ for some $k_A \in \mathcal{K}_A$ and $k_B \in \mathcal{K}_B$.

These functions have three properties:

1. $\forall k_A \in \mathcal{K}_A, \forall k_B \in \mathcal{K}_B, \forall b \in \{0,1\}$, $\mathbf{reveal}(k_A, k_B, \mathbf{enclose}(k_A, k_B, b)) = b$;
2. $\forall k_B \in \mathcal{K}_B, \forall c \in \mathcal{B}, \exists b \in \{0,1\}, \forall k_A \in \mathcal{K}_A$, $\mathbf{reveal}(k_A, k_B, c) \in \{b, \phi\}$.
3. No feasible probabilistic algorithm that attempts to distinguish blobs containing 0 from those containing 1, given k_B and c, is correct with probability significantly greater than 1/2.

The intention is that k_A is chosen by Alice and k_B by Bob. Intuitively, these conditions say:

1. Any bit b can be committed using any key pair k_A, k_B, and the same key pair will open the blob to reveal b.
2. For each k_B, all k_A that successfully open c reveal the same bit.
3. Without knowing k_A, the blob does not reveal any significant amount of information about the bit it contains, even when k_B is known.

A bit-commitment scheme looks a lot like a symmetric cryptosystem, with $\mathbf{enclose}(k_A, k_B, b)$ playing the role of the encryption function and $\mathbf{reveal}(k_A, k_B, c)$ the role of the decryption function. However, they differ both in their properties and in the environments in which they are used. Conventional cryptosystems do not require condition 2, nor do they necessarily satisfy it. In a conventional cryptosystem, it is assumed that Alice and Bob trust each other and both share a secret key k. The cryptosystem is designed to protect Alice's secret message from a passive eavesdropper Eve. In a bit-commitment scheme, Alice and Bob cooperate in the protocol but do not trust each other to choose the key. Rather, the key is split into two pieces, k_A and k_B, with each participant controlling one piece.

A bit-commitment scheme can be turned into a bit-commitment protocol by plugging it into the generic protocol given in Figure 4.

	Alice		Bob
	To $\mathbf{commit}(b)$:		
1.		$\xleftarrow{k_B}$	Choose random $k_B \in \mathcal{K}_B$.
2.	Choose random $k_A \in \mathcal{K}_A$.		
	Compute $c = \mathbf{enclose}(k_A, k_B, b)$.	$\xrightarrow{c}$	c is commitment.
	To $\mathbf{open}(c)$:		
3.	Send k_A.	$\xrightarrow{k_A}$	Compute $b = \mathbf{reveal}(k_A, k_B, c)$. If $b = \phi$, then fail. If $b \neq \phi$, then b is revealed bit.

Figure 4: A generic bit commitment protocol.

Each of the protocols of section 4 can be regarded as in instance of the generic protocol. For example, we get the protocol of Figure 1 by taking

$$\mathbf{enclose}(k_A, k_B, b) = E_{k_A}(k_B \cdot b),$$

and

$$\mathbf{reveal}(k_A, k_B, c) = \begin{cases} b & \text{if } k_B \cdot b = D_{k_A}(c) \\ \phi & \text{otherwise.} \end{cases}$$