# Notes for Lecture 14

## 1 Applications of Pairings

### 1.1 Recap

Consider a bilinear $e : G \times G' \to G_2$. We have $e(aP, bQ) = e(P, Q)^{ab}$. We call this an asymmetric pairing when $G \neq G'$. In a symmetric pairing, $G' = G$, you can pair an element by itself, which you can't do in asymmetric pairings.

We'll be looking at symmetric pairings today. Recall our example of 3 party key exchange, with three users Alice, Bob and Charlie. There was no obvious way to incorporate Charlie into the basic two party key agreement protocol. But with pairings, Alice can send $aP$, Bob can send $bP$, and Charlie can send $cP$. The key is $e(P, P)^{abc}$, which all the users can compute by pairing the other two users' messages and then raising the result to their own secret exponent.

This launched a whole subfield of cryptography called pairing-based crypto. Today we'll sample some of the results in this area.

### 1.2 Short Signatures

Lots of signatures we've seen require megabyte-sized signatures to get security against $2^{128}$ time adversaries. It turns out that pairings give the shortest signatures that we know of.

Before we see the scheme, we recall the Decision Diffie Hellman Assumption (DDH). The assumption is that $(P, aP, bP, abP)$ and $(P, aP, bP, cP)$ are computationally indistinguishable for random $a, b, c$.

We claim that DDH is easy in groups $G$ with a symmetric pairing. We can simply pair $e(aP, bP)$ and check if it's equal to $e(P, cP)$. Equality holds if $c = ab$, but holds with very small probability if $c$ is randomly chosen.

Computational Diffie Hellman Assumption (CDH): Given $P, aP, bP$, computing $abP$ is hard. It's unclear what pairing helps you do with respect to this task, and people believe that it's still hard to compute $abP$ even if a pairing is available (in elliptic curve groups).

An elliptic curve group with a pairing gives us a potential gap-DH group, which is a group where DDH is easy and CDH is hard. Consider the following signature scheme based on a gap-DH group $G$:

- $\mathsf{Gen}()$ : Pick a random $a \leftarrow \mathbb{Z}_p$, and let $Q = aP$, where $P$ is a fixed generator of the group. $a$ is the secret key and $Q$ is the public key.
- $\mathsf{Sign}(sk, m)$ : Assume a hash function $H : M \to G$. The signature is $\sigma = a \cdot H(m)$.
- $\mathsf{Ver}(pk, m, \sigma)$ : The inputs include $P, aP, m, a \cdot H(m)$. Instead of $m$, we look at $H(m)$, which we can think of as some $bP$. Then $aH(m) = abP$. Then this looks like a Diffie Hellman instance, so if I'm in a gap-DH group, I can check if I've determined the right solution.

Security intuitively follows from the CDH hardness assumption.

But what hash function should we use? We can't use $H(m) = mP$, since if you know $m$ you can compute $amP = m \cdot (aP)$. Remember the points on the elliptic curve satisfy $y^2 = x^3 + ax + b$. Let's take our message and treat it as the $x$ value. $H(m) = (m, y)$ where $y = m^3 + Am + B$. The relationship between operations in the $x$ coordinate and group operations is complicated and there's no way to run this attack. There's a security proof for this that we won't go over.

Signatures are just 1 group element, so if you want security against $2^{128}$ time adversaries, you can set your group size to be $2^{256}$, so you can get signatures of 256 bits.

## 1.3 Identity-Based Encryption

For everything we've seen so far, public keys have been long strings of bits corresponding to various algebraic values. In Identity-Based Encryption, we want public keys to be your email address. There is going to be a set-up algorithm run by some trusted third party.

- $\mathsf{Setup}() \to (msk, mpk)$. The master public key $mpk$ is public, but only the trusted third party knows $msk$.
- $\mathsf{Enc}(mpk, id, m) \to c$. Think of $id$ as the person's email address.
- $\mathsf{Extract}(msk, id) \to\!\!\to sk_{id}$. This algorithm is run by the trusted third party and outputs a secret key for user with identity $id$.
- $\mathsf{Dec}(sk_{id}, c) \to m$.

Once you know the master public key, you can send messages to anyone else without having to interact with the trusted third party.

Note that this protocol relies on users being able to authenticate their identities to the trusted third party. Intuitively, for security we want that if your identity is not $id$, you cannot distinguish messages intended for $id$.

Consider the following basic scheme.

- $\mathsf{Setup}() \to (msk, mpk)$. Generate a 2 by $n$ grid, and for each cell in the grid generate a public key, secret key pair. The master public key is the set of all public keys generated. The master secret key is the set of all secret keys generated.
- $\mathsf{Enc}(mpk, id, m) \to c$. Do an $n$ out of $n$ secret sharing of $m$. Basically just use random shares that XOR to $m$. Then you encode $c_i \leftarrow \mathsf{Enc}(pk_{i,id_i}, sh_i)$. $c = \{c_i\}_{i \in [n]}$.
- $\mathsf{Extract}(msk, id) \to\!\!\to sk_{id}$. Select the secret keys $\{sk_{i,id_i}\}_{i \in [n]}$ (select the grid boxes corresponding to the identity, meaning that for each column, read off a bit of the identity to choose between the top cell and the bottom cell).
- $\mathsf{Dec}(sk_{id}, c) \to m$. To decrypt, you decrypt each $c_i$ and recombine the secret shares to recover $m$.

If you have $id' \neq id$ and you have $sk_{id'}$, you can't learn anything about $m$ encrypted for user $id$. Essentially, at least one share must be hidden. However, this scheme has no protection against collusion between two or more users. Suppose one user is the all 0's and the other user is the all 1's. Then they have all the secret keys, and can break everyone else's secret keys.

This motivates defining collusion-resistant IBE. We can have some bounded collusion-resistant system where I can handle up to 1000 users colluding. However, what we really is that if everyone in the world gets together to collude, they still can't decrypt messages for you. This is called Fully Collusion-Resistant IBE.

Here is such a scheme built from pairings:

- $\mathsf{Setup}()$. This algorithm chooses a random $a \leftarrow \mathbb{Z}_p$, and sets $msk = a$. Then $Q = aP$ is set to be $mpk$.
- $\mathsf{Enc}(mpk, id, m)$. To encrypt, choose a random integer $d \leftarrow \mathbb{Z}_p$. Then output $(dP, e(Q, H(id))^d \cdot m)$. We assume messages are already group elements in $G_2$ (we don't care how this is done).
- $\mathsf{Extract}(msk, id)$. The secret key is just $aH(id)$.

- $\mathsf{Dec}(aH(id), (dP, e(Q, H(id))^d \cdot m))$. To decrypt, you compute $\frac{e(Q,H(id))^d \cdot m}{e(dP,aH(id))}$. Using $Q = aP$ and multilinearity, we see this gives $m$.

Security intuition is that $H(id)$ acts as a public key, and learning secret keys for other public keys doesn't let you learn the secret key of that specific user.

## 1.4 Broadcast Encryption

The setup is we have some content distributor $A$. Think of $A$ as XM radio or a TV channel. We have a bunch of users, $B, C, D, E, F$. At any given time, a subset of them have subscribed to $A$'s content. She has some message that she wants to broadcast (think of this as a song) over public airwaves, but she only wants the set of users (say $B, D, E$) to be able to decrypt and listen to the song.

The naive solution is to have each user own a public key/secret key pair, and have the public message be a concatenation of the ciphertexts encrypting the message to each user in the broadcast set. The public key is the concatenation of all the users' public keys. This is obviously infeasible for services like Netflix. Ideally we want the ciphertext to be $O(1)$, and the public key to be $O(1)$. The simplest solution we just saw has both of these at $O(N)$. It turns out that with pairings we have an $O(1)$ ciphertext size and $O(N)$ public key size. You can actually run instances of this pairing-based scheme in parallel to get $O(\sqrt{N})$ for both public key size and ciphertext size.

## 1.5 Multi-Party Key Exchange

To build this, we can hope to construct something like a tri-linear map. This would have the property that $e : G \times G \times G \to G_3$, where $e(aP, bP, cP) = e(P, P, P)^{abc}$. It turns out that if we have a multilinear map, where $e : G^k \to G_k$, we can build a number of things. For example (not a comprehensive list at all):

- Broadcast encryption with $O(1)$ ciphertext and public key size.
- Multiparty non-interactive key agreement for $k - 1$ users.
- Obfuscation. This is a tool that hides everything about a program but still allows for evaluation. Note obfuscation is different from garbled circuits, since with garbled circuits you need input labels and it's only one-time.
- Fully Homomorphic Encryption. This is an encryption allows you to compute on ciphertexts directly without leaking knowledge of the messages.

The remarkable thing is that for all these things a 3-linear map suffices.

Can we get a 3-linear map from elliptic curves? Recall the torsion group can be written as $E[p] \equiv \mathbb{Z}_p \times \mathbb{Z}_p$. So say $P, Q$ spans $E[p]$. Any point $R = aP + bQ$. I can think of the Weil pairing as $e(R, S) = \mu_p^{det(\begin{bmatrix} a & b \\ c & d \end{bmatrix})}$ where $\mu_p$ is the primitive $p$th root of unity and $S = cP + dQ$. We see that a 3-linear map presents issues, because this matrix becomes $\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$. It's unclear how to think of the determinant of this non-square matrix. It turns out you can use structures in algebraic geometry where the torsion group can be a three dimensional space. Now everything has three coordinates, and you can define a pairing like this with a determinant of a 3 by 3 matrix, but we have no idea how to compute this efficiently. And there are actually some reasons to believe this is not possible.

With lattices you can actually build something that looks like a multilinear map (called a graded encoding scheme), though the security for these constructions is still poorly understood. Next time we'll be talking about lattice-based crypto, but only schemes with well-understood security guarantees.