G22.3033-013 Exposure-Resilient Cryptography 7 February 2007

Lecture 4

Lecturer: Yevgeniy Dodis *Scribe: Marisa Debowsky*

1 Abstract

In this lecture, we give a high-level tour of DL-Based Threshold Cryptography. We begin with distributed generation of discrete log keys. We describe two protocols: (1) parallel Feldman and its (limited) security properties – good for adaptive Pedersen VSS, but not when the simulator needs to force the outcome (due to rushing), and (2) parallel Pedersen followed by the first protocol. Next, we describe BLS signatures and their threshold variant. Finally, we describe ElGamal encryption and its threshold variant, which uses an HVZK proof of equality of discrete logs.

2 Distributed generation of discrete log keys

Suppose $SK = x$ is the secret key and $PK = y = g^x \bmod p$ is the public key.

Feldman's Protocol. Each player does (possibly adaptive) Feldman VSS of x_i (where y_i is public); also, players hold the Shamir sharing of x_i, say, x_{ij}. If some P_i refuses, then set $x_i = 0$. Define:

$$y = \prod_{\text{participating } i} y_i.$$

Implicitly, $x = \sum_i x_i$, and the Shamir share $\overline{x_j}$ belonging to player P_j will be $\sum_i x_{ij}$. Add up all the shares:

$$\begin{array}{rcccc} & x_1 & x_{11} & \cdots & x_{1n} \\ & x_2 & x_{21} & \cdots & x_{2n} \\ & \vdots & \vdots & \ddots & \vdots \\ + & x_n & x_{n_1} & \cdots & x_{nn} \\ \hline & x & \overline{x_1} & \cdots & \overline{x_n} \end{array}$$

In summary: jointly, the players generate the secret key x and a public key $y = g^x$. They each have a Shamir secret share $\overline{x_j}$. Notice that $g^{\overline{x_j}}$ is public as well.

Properties of Feldman's Protocol. We assume, as usual, that $t = t_p < \frac{n}{2}$.

1. The simulator can arrange to know the discrete log of y. (This is because the simulator controls a majority of players.)
2. Any t (or fewer) bad guys do not know the discrete log of y, even if they collect their shares.
3. We can strengthen the previous property: for any challenge $\overline{y}$, the simulator can influence y such that

$$\log_g \overline{y} = \log_g y + \Delta,$$

where S knows Δ. (This property implies the previous, under the assumption that DL is hard.)

The problem with this protocol will be that we can't force Δ to be zero, which will motivate our introduction of a second protocol in a few minutes.

Proof of Property 1 S behaves honestly: she gets the shares x_i of bad players from the good players' shares (because the adversary is in the minority) and gets the shares x_i of the good players directly. Thus, S gets all the secrets. □

Proof of Property 3 For simplicity, assume a static adversary. (It will turn out that we can ensure this property for an adaptive adversary with probability 1/2, which will suffice for our applications.) Pretend that $\overline{y} = g^{x_i}$ for some honest player P_i; the rest you do honestly (as before). So the simulator can compute

$$\Delta = \sum_{j \neq i} x_j.$$

Then fake Feldman VSS using the Feldman VSS simulator.

For adaptive adversaries, choose a "single inconsistent player" (SIP) P_j at random (i.e., a player who doesn't know $\mathrm{DLog}(y_j)$). The simulator will fail with probability at most t/n (which is less than 1/2). □

What's good about this protocol? It's good enough to jointly generate h for the Pedersen commitment in the adaptive Pedersen VSS scheme. (See previous lectures.)

What's bad about this protocol? S can't embed $y = \overline{y}$ directly. That is, S can't force the output to be exactly what she wants. We can see this shortcoming as follows: each P_i commits to g^{x_i} right away. Thus:

$$y = \left(\prod_{i=\text{good}} y_i \right) \cdot \left(\prod_{i=\text{bad}} y_i \right)$$

But at the beginning, the **rushing adversary** forces the good guys to commit first. This means S is no longer in control, as the adversary has the freedom to choose her y_i's afterwards.

Before giving a correct protocol, though, let's be explicit about our objectives:

- Privacy: Our objective is that the simulator on input y should be able to force the output y.
- Robustness: No t players should be able to disrupt the process.

Pedersen's Protocol. [1] We wish to have our n players generate jointly a (secret key, public key) pair, namely (x, g^x), which satisfies the above privacy and robustness constraints. At the end of the protocol, the players will have the secret x and will have agreed on the published public key g^x.

Begin by doing a secret sharing of $x = \sum_i x_i$ (like in Feldman's protocol): $x_i \mapsto (x_{i1}, x_{i2}, \ldots, x_{in})$. In aggregate,

$$\begin{array}{cccc} & x_1 & x_{11} & \cdots & x_{1n} \\ & x_2 & x_{21} & \cdots & x_{2n} \\ & \vdots & \vdots & \ddots & \vdots \\ + & x_n & x_{n_1} & \cdots & x_{nn} \\ \hline \\ & x & \overline{x_1} & \cdots & \overline{x_n} \end{array}$$

Similarly, do a secret sharing of $r = \sum_i r_i$:

$$\begin{array}{cccc} & r_1 & r_{11} & \cdots & r_{1n} \\ & r_2 & r_{21} & \cdots & r_{2n} \\ & \vdots & \vdots & \ddots & \vdots \\ + & r_n & r_{n_1} & \cdots & r_{nn} \\ \hline \\ & r & \overline{r_1} & \cdots & \overline{r_n} \end{array}$$

Publish $g^{\overline{x_i}} h^{\overline{r_i}}$ and $g^x h^r$.

Do Parallel Pedersen VSS, which is information-theoretic. At the end of Pedersen, player P_i commits to $g^{x_i} h^{r_i}$ (and we have a perfectly hiding commitment to r).

We want to "open up" g^x, which will be our public key. Right now, the public information is $g^{\overline{x_i}} h^{\overline{r_i}}$ for each i and $g^x h^r$, and each player privately knows one g^{x_i}. The "opening" works like this: player i knows x_i and r_i, so she can generate g^{x_i} and g^{r_i}. And we've already published $g^{\overline{x_i}} h^{\overline{r_i}}$ for each i, so now each player can check that $c_i = g^{x_i} h^{r_i}$ for each i. Make c_i public.

If every player is honest, they we're done: simply publish $g^x = \prod g^{x_i}$. The trouble is that Player j could have lied and produced a fake pair "g^{x_j}" and "h^{r_j}" whose product checks out. To avoid this scenario: run a fresh parallel Feldman protocol on each x_i and r_i. This leaks $g^{x_i} = c_i'$ and $h^{r_i} = c_i''$, respectively. Then players check that $c_i' c_i'' = c_i$. Finally, $\prod y_i = y$.

The security of this protocol relies on the security of the Feldman protocol called as a subroutine; for security against a static attacker, run static Feldman; against an adaptive attacker, use adaptive Feldman.

Let's argue the soundness and privacy of this protocol. In the simulated attack game, the goal of the simulator will be to force an outcome $y*$ of his choice.

If you're the simulator, S, then you will know $\mathrm{DLog}_h g$. From the security of Pedersen, S extracts the share x_i from each cheating player and chooses x_i honestly for honest players. Then S chooses a single inconsistent player (SIP), i^*, which is not yet corrupted.

[1] "If I suffered, then you guys have to suffer a little bit as well. I spent all this energy figuring this out. Not to teach it would be a crime." –YD

S computes $\Delta = \sum_{i \neq i^*} x_i$. Now he pretends that $g^{x_i^*} = \frac{y^*}{g^\Delta}$ and $h^{r_i^*} = \frac{c_i}{g^{x_i^*}}$. (Remember that each player was committed to $g^{x_i} h^{r_i}$; for good guys, S opens the commitment honestly, and for those players, S computes an r_i which is consistent because he knows the discrete log.)

Next, fake parallel Feldman VSS. (See Lecture 3 for details.)

If y is correct, $y = g^{\Sigma(\text{ opened } x_i)}$ and then $\sum x_{ij} = \overline{x_i}$.

Modulo a failure event, S succeeds. A failure event here means that a bad guy opens $FVSS(x_i)$ to an adversarily-chosen $\tilde{x_i}$ and opens $FVSS(r_i)$ to $\tilde{r_i}$. Then the bad guy sets x_i, r_i, $\tilde{x_i}$, and $\tilde{r_i}$. But then S sees $g^{x_i} h^{r_i} = g^{\tilde{x_i}} h^{\tilde{r_i}}$, and he knows discrete logs, thereby giving a contradiction.

Thus, the protocol is sound.

3 Threshold BLS Signatures

Before describing the this signature scheme, let's be explicit about the attack model we will use in the proof of security. Players jointly generate the keys. As usual, the attacker can repeatedly ask up to t of the servers for computations. We'll say that the attacker wins if, after q queries to the servers, she can output a signature σ on a message m that is different from messages $m_1, \ldots, m_q$. For robustness: if a user wants to get a signature of m and there are at least t_r good servers, then the user *will* get $\mathsf{Sig}(m)$.

The **Boneh, Lynn and Shacham (BLS) Signature** will follow the same idea as RSA signatures, just in a different algebraic domain (in which distributed key generation is the same as DL).[2]

Take a DL-based group G and a second group G_T in which $|G| = |G_T| = q$. Assume $g \in G$ is a generator of G and $g_T \in G_T$ is a generator of G_T, and suppose that G and G_T are the particular sorts of groups for which there exists a map $e : G \times G \to G_T$ with the following properties:

1. $e(g, g) = g_T$,
2. $e(g^a, h^b) = e(g, h)^{ab}$.

A map satisfying these properties is called a bilinear map.

The BLS Signature has, in its setup, $SK = x$, $PK = g^x = y$, and system parameters $\{g, g_T, G, G_T\}$. Define $\mathsf{Sig}(M) = H(m)^x$, where H is a hash function, usually modeled as random oracle. The information known to the verifier is the DDH-tuple $\{g, y = g^x, H(m), \sigma = H(m)^x\}$, and the verifier should accept σ if

$$\text{DLog}_g(y) = \text{DLog}_{H(m)}(\sigma).$$

But how can he do it? Here, we use $e(\cdot, \cdot)$; namely, define

$$Ver(m, \sigma) = \left[e(g, \sigma) \stackrel{?}{=} e(y, H(m)) \right].$$

The verification equation should be satisfied if $\sigma = \mathsf{Sig}(m)$ because e is bilinear. Explicitly,

$$\begin{aligned} e(g, \sigma) &= e(g, H(m)^x) \\ &= e(g^x, H(m)) \\ &= e(y, H(m)) \,. \end{aligned}$$

[2] "I am going to choose the simplest possible examples because the more complicated ones are more complicated." –YD

Remark. The DDH assumption is *false* in these groups! Recall that the DDH assumption says that it's hard to distinguish $\langle g, g^a, h, g^a \rangle \approx \langle g, g^b, h, g^b \rangle$. This is not the case here. Indeed, this is precisely what we use in the verification equation.

How to Distribute. First, we need to run the distributive key generation protocol. In this case, the Feldman protocol might be enough (check this as an exercise), but surely we can use the second, Petersen-based protocol, at the end of which each player knows $x_i = SSS(x)$ and all players know the public key.

Now for signing: P_i outputs $H(m)^{x_i} = \sigma_i$. Anybody can check that σ_i is correct by doing the same verification: $e(g, \sigma_i) \stackrel{?}{=} e(y_i, H(m))$.

(We note, parenthetically, that we've changed our notation slightly here. Earlier, we had $x = \sum x_i$ and also x was Shamir's secret sharing of $\overline{x_i}$. Now, x_i is Shamir's secret sharing of x. Not linear, but t-out-of-n.)

After collecting any $t+1$ good shares σ_i, we can recover σ using Lagrange interpolation in the exponent. Namely,

$$\begin{aligned} \sigma &= \prod_{\text{the } t+1 \text{ good shares}} \sigma_i^{\lambda_i} \\ &= \prod H(m)^{\lambda_i x_i} \\ &= H(m)^{\sum \lambda_i x_i} \\ &= H(m)^x \end{aligned}$$

Lemma 1 *If there is some attacker A (controlling at most $t \leq \frac{n}{2}$ players) breaking the threshold BLS signature scheme, then there exists an attacker A' breaking the regular BLS signature scheme.*

Proof of Lemma. Suppose we have such an A. Then $A'(y)$ runs A using a simulator which forces $PK = y$. Notice that at this stage, S knows the share x_i for every corrupted player. (Moreover, for adaptive corruptions, S will learn or cook up the share x_i for the new corrupted player. For good guys, he doesn't know shares because he doesn't know the global x.) Now A' needs to simulate the signing oracle for A. If A initiates $\mathsf{Sig}(m)$, then A' gets it from its own signing oracle. Then A' can compute σ_i for every corrupted i, and using Lagrange interpolation, compute σ_i for all honest players as well (using σ). (This is essentially reverse Lagrange interpolation.) Finally, when given a forgery (m, σ), just output it. □

4 Threshold ElGamal Encryption

Recall the basic ElGamal Encryption setup: we let $x = SK$, $y = g^x = PK$. (Here, we need groups where DDH is believed to be true; in particular, we can't use the groups from above, but a usual prime order subgroup of $\mathbb{Z}_p^*$ will be fine. Suppose that $|G| = p$ is prime and and that q is another prime dividing $p-1$.) Define $Enc(m) = (g^r, y^r \cdot m)$ where $r \in \mathbb{Z}_q$ and $Dec(A, B) = B/A^x$ because

$$\frac{B}{A^x} = \frac{y^r \cdot m}{(g^r)^x} = \frac{(g^x)^r \cdot m}{g^{rx}} = m.$$

For threshold encryption, we have to be a little bit careful about the attack model. Naturally, you would expect an attacker to interact with servers to do decryption. However, in that case, the underlying encryption scheme should be CCA-2 secure. ElGamal is only CPA secure. Therefore, we consider a weaker attack model. In this model, the attacker chooses two messages, m_0 and m_1; the challenger sets ciphertext

C to be $Enc(m_b)$. The challenger honestly interacts with servers to decrypt the ciphertext C. The attacker, who controls at most t servers, has to guess the bit b after interacting with the challenger in the decryption process.

This is a privacy attack. The robustness property is similar to other signatures; after interacting with at least t_r honest servers, with a ciphertext, you should always be able to decrypt.

Given a ciphertext (A, B), what we need to compute is A^x.

$$\begin{aligned} A^x &= A^{\sum \lambda_i x_i} \\ &= \prod (A^{x_i})^{\lambda_i}, \end{aligned}$$

where the sum and product are taken over the i's of good players. Player P_i sends $\sigma_i = A^{x_i}$ to the user. How do you check that A^{x_i} is correct? Unlike in the previous example, there's no nice bilinear map, since with a bilinear map, you can break the semantic security of ElGamal. We consider three different approaches.

1. You can do Reed-Solomon decoding in the exponent. There is a big problem here, which is that nobody knows how to do it; it's an open problem, and likely hard.

2. Instead, you might use honest verifier zero knowledge (HVZK) proofs. The server is going to prove that
$$\text{DLog}_A(\sigma_i) = \text{DLog}_g(y_i).$$
The proof is actually simple. We want to show, generally, that $\text{DLog}_A(B) = \text{DLog}_g(h)$ (a proof of the equality of discrete logs). The protocol will be:
   - Prover sends $G = g^r$, $H = h^r$ to the Verifier.
   - Verifier sends challenge c to the Prover.
   - Prover sends $z = r + cx$, and the Verifier checks that $g^z = G \cdot g^c$, $A^z = H \cdot B^c$.

   Notice that in this application, honest verifier is fine because the challenger is honest.

3. Alternatively, you could condense to non-interactive ZK using the Fiat-Shamir heuristic. This will be full NIZK in the random oracle model. Namely, in the HVZK proof above, set $c = H(g^r, h^r,$ "context info"$)$ and use this c as if V sent c. Here, H is modeled as random oracle, and the "context info" is a unique identifier to ensure one doesn't reuse the 'same' oracle in different applications. For ElGamal encryption, the "context info" is the ciphertext $C = (A, B)$.

References

[1] D. Boneh, B. Lynn, H. Shacham. Short Signatures from the Weil Pairing. In *Proc. ASIACRYPT '01*, pages 514–532, 2001.

[2] M. Abe, S. Fehr. Adaptively Secure Feldman VSS and Applications to Universally-Composable Threshold Cryptography. In *CRYPTO '04, Lecture Notes in Computer Science vol. 3152*, pages 317–334, 2004.

[3] R. Canetti, R. Gennaro, S. Jarecki, H. Krawczyk, T. Rabin. Adaptive Security for Threshold Cryptosystems. In *CRYPTO '99, Lecture Notes in Computer Science vol. 1666*, pages 98–115, 1999.