Lattices and Homomorphic Encryption, Spring 2013 Instructors: Shai Halevi, Tal Malkin

Attribute-based Encryption & Delegation of Computation

April 9, 2013 Scribe: Steven Goldfeder

We will cover the ABE and delegation schemes from [GVW13].

Attribute based Encryption(ABE)

Motivation: Say we wanted to extend Facebook's permission structure so that the audience of a post is determined by the post's content. One way of realizing this is for the poster to tag each post with attributes (e.g. food, sports, cryptography). Users then subscribe to receive posts based on the posts' attributes. For example, a user may sign up to receive posts tagged with "food". The user can have a more complicated policy as well–e.g. subscribe to "food" posts that are also tagged with at most two of the attributes "sweet", "sour", "bitter", "salty".

Now suppose that we want to enforce this attribute-based subscription with cryptography. Each post is encrypted, and each user has a policy P and a key k_P that only allows them to decrypt posts that satisfy their policy.

Another example use case is someone sends an email to an organization's email list, but only wants users with specific attributes (e.g. administrators) to be able to read the email.

Definition 1 (ABE)**.** *Formally, we define an ABE scheme by the following four procedures:*

- $(pp, msk) \leftarrow \texttt{Setup}(\$)$: Generates public parameters pp that are used to encrypt(includes the set of attributes etc.) and the corresponding master secret key msk.
- $ct_x \leftarrow \texttt{Encrypt}(M; pp, x)$: Encrypt message M with respect to the attribute-string x and the public parameters pp. There is a predefined set of attributes S, and $x \in \{0,1\}^{|S|}$ denotes the presence/absence of these attributes, i.e., each bit of x corresponds to an attribute such that:

$$x_i = \begin{cases} 1 & \text{if the associated attribute is present} \\ 0 & \text{otherwise} \end{cases}$$

  Note: The notation ct_x is used to indicate that we encrypt relative to x. In the schemes that we will see we will not attempt to hide x (but in other cases this may be a requirement).

- $sk_p \leftarrow \texttt{KeyGen}(p; msk)$: Generates a key for policy p.

  Note that $p : \{0,1\}^{|S|} \to \{0,1\}$ is a function such that on input of an attribute-string x,

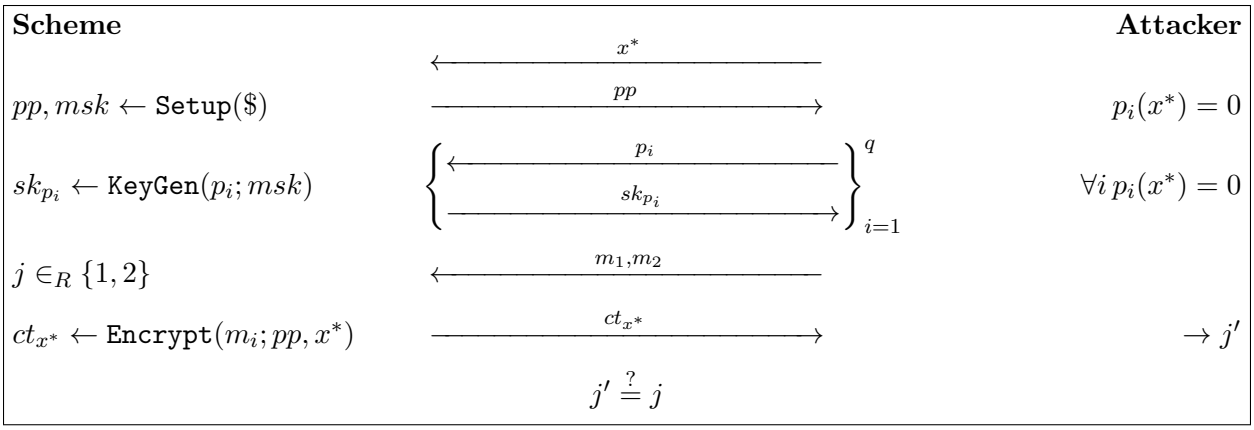
$$p(x) = \begin{cases} 1 & \text{if } x \text{ satisfies the policy} \\ 0 & \text{otherwise} \end{cases}$$

- $m/\perp \leftarrow \texttt{Decrypt}(ct_x; sk_p)$: Recovers m if $p(x) = 1$. Outputs nothing if $p(x) = 0$.

Remark: In some sense, we can view a public key encryption scheme as a simple case of ABE with a singleton policy x. That is, $S = \{0,1\}^n$ consists of all the possible user-names in the system, and each user $u \in S$ gets a key corresponding to policy $p_u(x) = 1$ iff $x = u$. (The same description in fact applies to *identity-based encryption.*)

Security : Intuitively, in order for the scheme to be secure, no colluding group of key holders can learn anything about m as long as for all of them, $p_i(x) = 0$. We formalize security by means of a game between the scheme and an attacker who asks for certain key(s) and then is attempts to decrypt a message that was encrypted under a policy that he does not have a key for.

The game proceeds as follows: The attacker begins by sending x^*, the set of attributes that he wants to attack. The scheme then runs `Setup` and obtains the public parameters, pp, and the masters secret key, msk. It sends pp to the attacker. Then, the attacker asks for the key to a policy $p_i$and the scheme gives him a key as long as this won't allow him to encrypt x^* encryptions(i.e. $p_i(x^*)$ must equal 0.). He repeats this for as many rounds as he'd like. We denote the number of rounds by q. At any point after he commits to x^*(could be before, in middle or after the q queries), the attacker sends two messages m_1, m_2. The scheme chooses one of them at random and encrypts it with respect to the policy x^*, and sends the ciphertext to the attacker. The attacker now tries to determine which message was encrypted. Graphically:



The attacker wins if $\forall i\, p_i(x^*) = 0$ and $j' = j$. By guessing, the attacker wins with probability $\frac{1}{2}$. The scheme is secure if for any probabilistic-polynomial-time attacker A,

$$Pr[A \text{ wins}] < \frac{1}{2} + \epsilon$$

where ϵ is negligible in the (implicit) security parameter.

Delegation of Computation

Motivation: I have a complicated function that I want to compute, but only a weak device, say a cell phone. I am willing to pay a server to compute the function for me on inputs of my choice, and send me back a proof that the result is correct. The proof must be simple enough that I can verify it on my weak device.

Definition 2. *(Delegation scheme) Formally, we define a delegation scheme by the following four procedures:*

- $(pp, msk) \leftarrow \texttt{Setup}(p)$: Pre-process the function $p : \{0,1\}^m \rightarrow \{0,1\}$. This may be as expensive as computing p, but it is only done once. The public parameters pp are given to the server. The secret msk is kept on your "weak" device.

- $(ex, chk) \leftarrow \texttt{Encode}(x; pp, msk)$: Encode x an input to p, generating an encoded version, ex, and some check parameters. This step must be "cheap" (computable on your weak device). The encoded input is sent to the server, and the check parameters are kept secret.
- $(y, prf) \leftarrow \texttt{Compute}(ex; pp)$: This is done by the server. The server claims $y = p(x)$ and provides prf, a proof of the correctness of y.
- Yes/No $\leftarrow \texttt{Verify}(y, prf; msk, chk)$: Verify the proof. This must be "cheap".

Correctness: Intuitively, we don't want the server to be able to cheat. So it shouldn't be able to come up with the wrong answer and a convincing proof. We can formalize it by means of a game as we did for ABE, this is in the homework.

Claim: We can construct a "sound" delegation scheme from ABE.

Proof: Say $p : \{0,1\}^n \to \{0,1\}$. We can make this assumption since any program that outputs more than one bit can be expressed as a series of programs with one-bit outputs. We define the function $p'(x, b) = p(x) \oplus b$. To pre-process the function p, we first run $(pp, msk) \leftarrow \texttt{ABE.Setup}(\$)$. We then run $sk_{p'} \leftarrow \texttt{ABE.KeyGen}(p', msk)$. The server is given pp and $sk_{p'}$.

To evaluate p on some input $x \in \{0,1\}^n$, I encrypt two random messages, one under $x'_0 = (x, 0)$ and the other under $x'_1 = (x, 1)$, and send x and the two ciphertexts to the server.

The server can decrypt only one of the two ciphertexts since either $p'(x, 0) = 1$ (if $p(x) = 1$) or $p'(x, 1) = 1$ (if $p(x) = 0$). The server sends me the random message that it could decrypt, and I determine $p(x)$. The reason that we define p' and add the bit b is because without this, the server can always claim that it cannot decrypt. In our construction, though, the server can always decrypt exactly one of the two messages, and thus must return one decryption.

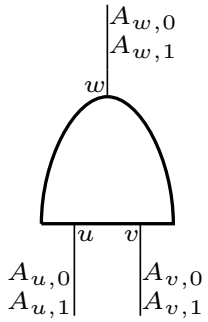
We have thus shown that ABE is "stronger" than delegation. Next we will show how to do delegation from LWE, and then show that this construction actually extends to get ABE.

Delegation Scheme Construction

We begin with a broken "linear-scheme" and then fix it by adding noise. Consider a fan-in-2 circuit for computing p. With each wire, we associate two linear functions (e.g. expressed by matrices):

$$L_{A_0}(\vec{s}) = \vec{s}A_0, \quad L_{A_1}(\vec{s}) = \vec{s}A_1$$

Consider one specific gate (say an AND gate) with input wires u, v and output wire w.



Similarly to Yao circuits, we want to give out a "translation table" so that if you know $L_{A_{u,b}}(s)$ and $L_{A_{v,c}}(s)$, then you can learn $L_{A_{w,(b \text{ AND } c)}}(s)$ (but you should not be able to learn s.).

For every wire w, choose $A_{w,0}$ and $A_{w,1}$. For every gate $G_{u,v,w}(b,c)$ where b and c are the input bits, choose four pairs of matrices such that for every pair of bits b, c we have matrices $R_{b,c}, R'_{b,c}$ that satisfy:

$$bc \in \{0,1\}^2 : A_{u,b}R_{b,c} + A_{v,c}R'_{b,c} = A_{w,G(b,c)}.$$

Continuing the illustration with our AND gate above, we have:

$$\left.\begin{array}{l} 00 : (R_{0,0}, R'_{0,0}) \text{ such that } A_{u,0}R_{0,0} + A_{v,0}R'_{0,0} = A_{w,0} \\ 01 : (R_{0,1}, R'_{0,1}) \text{ such that } A_{u,0}R_{0,1} + A_{v,1}R'_{0,1} = A_{w,0} \\ 10 : (R_{1,0}, R'_{1,0}) \text{ such that } A_{u,1}R_{1,0} + A_{v,0}R'_{1,0} = A_{w,0} \end{array}\right\} \text{output} = 0$$

$$\left.\begin{array}{l} 11 : (R_{1,1}, R'_{1,1}) \text{ such that } A_{u,1}R_{1,1} + A_{u,1}R'_{1,1} = A_{w,1} \end{array}\right\} \text{output} = 1$$

Say you know two vectors $\vec{v_0} = \vec{s} \cdot A_{u0}$ and $\vec{v_1} = \vec{s} \cdot A_{v1}$ (for the same s). Then you can compute

$$\vec{w_0} = \vec{u_0}R_{01} + \vec{v_1}R'_{01} = \vec{s}A_{u0}R_{01} + \vec{s}A_{v1}R'_{01} = \vec{s}A_{w0}$$

Similarly, for any other pair of the form $\vec{s}A_{ub}$ and $\vec{s}A_{vc}$.

Note: You need to know the bits b, c, so that you can compute using the appropriate pairs of matrices (R_{bc}, R'_{bc}).

Thus we have the following (broken) delegation scheme:

- `Setup`(p): Choose two matrices $A_{w_i,0}$, $A_{w_i,1}$ for every wire in the circuit for p. Then, for every gate with input wires u, v and output wire w, solve the appropriate linear systems to find a pair of matrices (R_{00}, R'_{00}) such that $A_{u0}R_{00} + A_{v0}R'_{00} = A_{w,G(0,0)}$. (If this system is under-defined, then choose a random solution.) Similarly, choose the pairs (R_{01}, R'_{01}), (R_{10}, R'_{10}), and (R_{11}, R'_{11}).

  The public parameters are all the R's. The secrets are the $A_{w_i,b}$'s corresponding to the input and outputs wires.

- `Encode`$(x \in \{0,1\}^n; msk)$: Chose a random vector $\vec{s}$. For input wire w_i that corresponds to an input bit $x_i \in \{0,1\}$, compute $\vec{v_i} = \vec{s}A_{w_i,x_i}$. The encoding includes x and all the $\vec{v_i}$. The check information is $\vec{s}$.

- `Compute`$(ex; pp)$: For every wire in the circuit, compute the bit on that wire using the bits of x. For every gate G with input bits b, c,input vectors $\vec{u_b}, \vec{v_c}$,and output bit $d = G(b,c)$, we compute the output vector:

  $$\vec{w_d} = \vec{u_b}R_{bc} + \vec{v_c}R'_{bc}$$

  Return y,the bit on the output wire, and the corresponding vector as proof.

- `Verify`$(y, \overrightarrow{prf}; msk, chk)$: Check that $\overrightarrow{prf} = \vec{s} \cdot A_{w_{\text{out}},y}$.

Why is this scheme broken? The server can begin by answering honestly, and accumulate enough information to cheat on all subsequent queries. How? It can solve a linear system of equations to express any vector corresponding to $x_1 = 1$ and a linear combination of some previous such vectors. Similarly for the output wire. After many inputs with $x_1 = 1$ and output= 1, given any new input with $x_1 = 1$, you can compute a 1-vector that passes verification.

Fixing the broken scheme

Step 1 : Replace $\vec{s}A$ by $\vec{s}A + \vec{e}$ where the A matrices are $\in \mathbb{Z}_q^{n \times m}$. The noise distribution as well as n, q, m are chosen as appropriate D-LWE parameters. This solves the attack, because $\vec{s}A + \vec{e}$ is pseudorandom, so the server only sees things that look random. Whereas before it had a linear function, now whatever it sees looks random.

But how can you verify? Again, let $d = G(b, c)$. Then we have:

$$\begin{aligned}\vec{w}_d = \vec{u}_b R_{b,c} + \vec{v}_c R'_{b,c} &= (\vec{s}A_{u,b} + \vec{e}_u)R_{b,c} + (\vec{s}A_{v,c} + e_v)R'_{b,c} \\ &= \vec{s}A_{w,d} + \underbrace{\vec{e_u}R_{b,c} + \vec{e_v}R'_{b,c}}_{\vec{e'}}\end{aligned}$$

But even if $\vec{e}_u$ and $\vec{e}_v$ are small, $\vec{e'}$ is not! So the verifier cannot just check that the output vector is "close" to $\vec{s}A_{w_{\text{out}},y}$.

Step 2 : Use small R's. If we could somehow make sure that all R's are small, then the errors will not grow too much with every gate. So if we started from small $\vec{e}_i$'s at the inputs, then we could still get a "rather small" error-vector in the output.

Recall that for every gate we have four constraints of the form $A_{u,\star}R + A_{v,\star}R' = A_{w,\star}$. These are linear constraints (in the entries of R and R') and finding small solutions to linear constraints is the SIS problem, which we believe to be hard. So how can we do that? We will generate the A's together with trapdoors!

For every wire w, we choose at random $A_{w,0}, A_{w,1} \in_R \mathbb{Z}_q^{n \times m}$ together with matching trapdoors. Then, we can find $R_{b,c}, R'_{b,c}$ as follows:

- Choose the entries of $R'_{b,c}$ from a discrete Gaussian over $\mathbb{Z}$

$$(R'_{b,c})_{i,j} \leftarrow D_{\mathbb{Z},\sigma} \text{ for } i, j \in \{1 \cdots m\} \qquad (\text{so } R' \in \mathbb{Z}^{m \times m}) \text{ and } \sigma \text{ is small relative to } q$$

- Compute $\Delta = A_{w,d} - A_{v,c}R'_{b,c} \in \mathbb{Z}_q^{n \times m}$ (i.e. the answer we are looking for $A_{w,d}$ minus the part we already have).
- Use the trapdoor to find $R_{b,c}$, small, such that $A_{u,b}R_{b,c} = \Delta$.

Let the columns of Δ be $\Delta = (\vec{\delta}_1 | \cdots | \vec{\delta}_m)$. Then the ith column of R must satisfy $A_{v,b}\vec{r}_i = \vec{\delta}_i$. Using the trapdoor, we choose $\vec{r}_i$ from the integers such that the above constraint is met. That is, we choose the ith column of R from a Discrete Gaussian over the coset:

$$\vec{r}_i \leftarrow D_{\mathcal{L}^{\perp}_{\vec{\delta}_i}(A_{u,b}),\sigma}$$

Alternatively, we could choose $R_{b,c} \in \mathbb{Z}_q^{m \times m}$ at random and use the trapdoor $t_{v,c}$ to solve for $R'_{b,c}$. The resulting distribution on the pair $(R_{b,c}, R'_{b,c})$ will be nearly the same, namely a Gaussian $\tilde{R} = \begin{pmatrix} R_{b,c} \\ R'_{b,c} \end{pmatrix}$ such that

$$\underbrace{\begin{pmatrix} A_{u,b} & | & A_{v,c} \end{pmatrix}}_{n \times 2m} \underbrace{\begin{pmatrix} R_{b,c} \\ \\ R'_{b,c} \end{pmatrix}}_{2m \times m} = \underbrace{\begin{pmatrix} A_{w,d} \end{pmatrix}}_{n \times m}$$

Actually, there are three different procedures that produce nearly the same (statistically close) distributions over (A_u, A_v, A_w, R, R'):

1. choose A_u, A_v, A_w at random with trapdoors. Choose R' from a Gaussian and use the trapdoor for u to choose a Gaussian R subject to $A_uR + A_vR' = A_w$.

2. Choose A_u, A_v, A_w at random with trapdoors. Choose R from a Gaussian and use the trapdoor for A_v to choose a Gaussian R' subject to $A_uR + A_vR' = A_w$.

3. Choose A_u, A_v at random. Choose R, R' from a Gaussian. Compute $A_w = A_uR + A_vR'$. (A_w is nearly uniform by the Leftover Hash Lemma since A_u and A_v are random and therefore define a universal hash function. The columns of R have high min-entropy, and thus applying the universal hash function to a high min-entropy distribution gives us a random A_w).

Parameter setting. We need to set q large enough. Recall that at each gate, the noise in the output is of the form $\vec{e'} = \vec{e_u}R_{b,c} + \vec{e_v}R'_{b,c}$. Since $R, R' \in \mathbb{Z}_q^{m \times m}$, (dimension m with entries of roughly size m)then the size of the output error:

$$||\vec{e'}|| \approx m^2||\vec{e}_v, \vec{e}_u||$$

If d is the depth of the circuit computing p, then we need $q > m^{\Omega(d)}$ (since the size of the error grows by a factor $poly(m)$ with each gate and we need the final error to be smaller than q). This means that we need to know the depth of our circuit ahead of time, and then choose the parameters based on the depth. There is no way in this scheme to handle every circuit depth with a single construction.

We also need to choose n and m relative to q so that LWE is still hard. For example if the ratio between q and m is fully exponential, say $q = 2^m$, then we can use LLL to find small elements in the lattice and break LWE. So, in our example, we would use $M = m^2$ and $Q = q^2$.Thus $Q = 2^{2\sqrt{M}}$ and we cannot use LLL anymore.

References

[GVW13] S. Gorbunov, V. Vaikuntanathan and H. Wee. Attribute-based Encryption for Circuits *Proccedings of STOC 2013.*