COS 533: Advanced Cryptography — Princeton University
Lecture 4 (Feb 11)
Lecturer: Mark Zhandry — Scribe: Xiaoqi Chen

Notes for Lecture 4

1 Last Time

We were trying to prove the Goldreich-Levin theorem, which states that every OWF has a hardcore bit. We had to technically modify the one-way function but it's a technical detail.

Basically, what it reduces to is you're given an adversary B, that given $F(x)$ and r can compute $\langle x, r\rangle = \sum x_i r_i \mod 2$ with probability better than a half.

In the last lecture we proved that, for an adversary that achieves

$$\Pr[B(r, F(x)) = \langle x, r\rangle] \geq 3/4 + \gamma,$$

let $H(r) = B(r, F(x))$, we can find some choices of x (at least $\gamma/2$ fraction of xs) that satisfy

$$\Pr[H(r) = \langle x, r\rangle] \geq 3/4 + \gamma/2.$$

We subsequently feed the H with two random but correlated input only differ in one bit, and notice that

$$\Pr[H(r) \oplus H(r, e_i) = x_i] \geq 1/2 + \gamma,$$

i.e., the probability that both H are correct or both are wrong (which leads to the correct inner product) is $1/2 + \gamma$, good enough for us to guess each digit x_i. We can then repeat this step for every i to recover x.

2 Proof of hardcore existence continued: Recovering x from a weaker $H(r) = B(r, F(x))$

This time, we want to assume we only have $\Pr[H(r) = \langle x, r\rangle] \geq 1/2 + \varepsilon$. The problem is that given such H, the choice of correct x might not be unique, there might be two or more x that satisfy the above condition.

We have to resort to something else. In *Error Correcting Code* language, we call it list decoding, and solve all xs for a given H such that the condition holds. There are only polynomial number of xs and we hope to output all of them efficiently.

Our strategy is to use H to implement a H' such that $\Pr[H'(r) = \langle x, r\rangle] \geq 7/8$. We construct several different H' such that the choice of x is unique in H'. We can then use the analysis from the last time to recover x (since $7/8$ is above $3/4$).

First, we choose a random string $r_1, \ldots, r_k \in \{0,1\}^\lambda$ for some parameter k that we'll choose later. Now, we pretend we can magically obtain $b_i = \langle x, r_i\rangle$ (the inner product of x with all r_i), for one of the x that's consistent with H.

Now we define

$$H'(r) = Maj_{i\in[k]} \left(H(r \oplus r_i) \oplus b_i\right).$$

This is similar to what we did in the $3/4$ case, but in that case we used H two times, therefore requiring the probability to be greater than $3/4$. Here since H's probability of correctness is only weakly over $1/2$, we can't put two different value into H or we'll be in trouble. Hence we assume we magically know one x already.

Now we note, by definition we have

$$\Pr_{r_1,\ldots,r_k,\ r}[H(r \oplus r_i) \oplus b_i = \langle x, r\rangle] = \Pr[H(r \oplus r_i) = \langle x, r \oplus r_i\rangle] \geq 1/2 + \varepsilon.$$

We can pull in the XOR into the inner product; then, since $r \oplus r_i$ is just a random value, by definition H will give correct answer with probability $1/2 + \varepsilon$.

We can now set k high enough so that the majority is correct with high probability:

$$\Pr_{r_1,\ldots,r_k,\ r}[H'(r) = \langle x, r\rangle] \geq 31/32$$

(We won't bother to calculate how large k must be here.)

Why $31/32$? By Markov inequality, if we separate the probability, with $31/32$ we can show that $\Pr_{r_1,\ldots,r_k}[\Pr_r[H'(r) = \langle x, r\rangle] \geq 7/8] \geq 3/4$. Then, with the inner probability of H, using the sample from the last lecture, we can use the $3/4 + \gamma$ case to find x.

Finally, the last part of the proof is we need to show how to obtain b_is. Let $k = 2^l$ for some l. Choose random $r'_1, \ldots, r'_l \in \{0,1\}^\lambda$. For $S \subseteq \{1, \ldots, l\}, r_s = \bigoplus_{i\in S} r'_i$. Here we have $2^l = k$ different r_ss, and there's enough for us to apply the step above. But how do we compute b_is?

Pretend that we can magically compute $b'_i = \langle x, r'_i\rangle$, then we know b_S is simply $b_S = \bigoplus_{i\in S} b'_i$. So we generated rs by taking the subset sum of a very small number of rs, and to compute the bs for these rs we only need to compute the subset sum of these b's.

We reduced the task of computing for $k = 2^l$ different b_Ss to just computing for l different b'_is. The point is l is **logarithmic**, so we can just **guess** the b'_is, by iterating over all 2^l cases.

So, working backwards, the algorithm works as follows. We first guess r'_is and guess b'_is. We are right with some non-negligible probability. We then construct b_is and r,

and construct H' then H, and use algorithm in last class to recover x. If we iterated all possible b_i's, we can actually recover all xs.

Remark. Here the r_Ss and b_Ss are all correlated. We are not sampling r_Ss uniformly at random, but it doesn't matter. The analysis still works out just fine.

Question from student: In the part where we use majority and large enough k to achieve 31/32 probability for correctness, if r_S are correlated, how do we use the probability bounds (which requires independence)?

Mark: We have to check the detail, but the point is in every step of the analysis we don't need the variables to be completely independent, it turns out we just need them to be somewhat independent.

In fact, each r_S is on its own uniformly at random (or very close to it), and any two of them are randomly independent (in fact, any $log(k)$ of them).

Follow up question from student: If these r_Ss are completely independent, then the distribution will follow something like approximately normal distribution, and the probability of error should go down exponentially. So wouldn't it be enough to set k to be logarithmic, and no need to guess the "small" ones, but we can just guess the original r_is in the beginning?

Mark: Why doesn't that work? I think the problem is that, if you do the analysis, I think you really need something close to $1/varepsilon$ number of samples, if you compute what k you needed. This is polynomial and then you can't guess all the r_is.

In any case, for the purpose of this course, there's no need to go into that level of detail. We certainly went over some details like what exactly is k, but hopefully the algorithm itself makes sense.

3 Building signature scheme from OWF

3.1 Defining signature scheme

We show that one way permutation OWP implies pseudo-random generator PRG, which implies pseudo-random functions PRF, and implies encryption ENC.

We also know (beyond the content of this course) that one way functions OWF can make PRG.

Now we want to add a small piece to the puzzle: a one-way function OWF can make signatures, the digital equivalent of a wax seal outside of an envelope, with the same effect that no one has modified the content.

Definition. A *Signature scheme* is part of a public key scheme. We first make a pair

of keys by defining key-generator function[1]

$$(sk, pk) \leftarrow Gen(1^\lambda).$$

Then we can get signature of message M:

$$\sigma = Sig(sk, M).$$

Someone can then verify the signature via

$$0/1 \leftarrow Ver(pk, M, \sigma),$$

with output 0 means reject and 1 means accept.

The definition of correctness is trivial, the verification function will accept signatures correctly signed.

The intuition of the security definition is that the attacker can't change the content and forge a correct signature. The recipient will always verify the (potentially modified) signature, so the attacker must come up with a signature.

We first define **One-Time Security**:

- Challenger Ch first make key pairs $(sk, pk) \leftarrow Gen(1^\lambda)$. Give pk to attacker A.
- Attacker A chooses message M and gets signed $\sigma = Sign(sk, M)$.
- Attacker comes up with M^*, σ^*.
- Attacker wins if (i) $M^* \neq M$ and (ii) $Ver(pk, M, \sigma) = 1$.

Note that we're letting the attacker to be both the sender and receiver of the message, choosing M and getting its signature. The public key pk is broadcasted to everyone, so at some point the recipient can verify a received message's correctness. Here we are allowing the attacker to influence the message gets signed, similar to the encryption case where attacker can modify the plaintext being encrypted.

In many cases, you want **Many-Times Security**. The attacker can query for many correct signatures and do one forgery. The definition is similar, just allow multiple queries $\{M_i\}$ with $\sigma_i = Sign(sk, M_i)$ in between, with the same final acceptance of one forged message signature. Also, nontrivial attack means $M^* \notin \{M_i\}_i$, i.e., the attacker should come up with a new message, not repeating a message it queried.

The signature scheme is (one-time/many-time) secure if, $\forall$ PPT adversary A, $\Pr[A\ wins]$ is negligible.

[1]Note: unary notation $1^\lambda = (1111\ldots)$ is merely a syntactic notation for ease of defining stuff running in polynomial time (poly to input length λ). If we designate λ as input, its length would be logarithmic, so polynomial running time means poly-log which is too strict.

3.2 From OWF to One-time signatures: Lamport signatures

We now show OWF implies one-time signatures, which then implies many-time signatures.

For the one-time case, the construction is elegant, and is called Lamport signatures.

First, for $Gen(1^\lambda)$: assume we have a OWF F. We choose $X_{i,b} \leftarrow \{0,1\}^\lambda$ for $i \in \{1,\ldots,n\}, b \in \{0,1\}$. Let $y_{i,b} = F(X_{i,b})$. Let $pk = y_{i,b_{i,b}}$ and $sk = X_{i,b_{i,b}}$.

Now, let's sign an n-bit message: let $Sign(sk, M \in \{0,1\}^n)$ output $\{X_{i,m_i}\}$. Imagine X as a grid with n columns and 2 rows, and we select one of the two grids for every column, based on the message's bits. We create $2n$ grids, and to sign a message we reveal n of them.

To verify, we check $F(\sigma_i)$ is indeed y_{i,M_i}; i.e., the signature is indeed the preimages of the y values.

Security proof: the idea is that $Sign(sk, M)$ reveals half of the key, but to compute $Sign(sk, M^* \neq M)$ for a different message will require a different key component that the attacker haven't seen. Let's assume adversary A first commits ($i \in \{1,\ldots,n\}, b \in \{0,1\}$) ahead of time, then gets pk. Then it plays the signing game by providing M, but satisfies $M_i \neq b$. Then it gets σ, and sends M^*, σ^* where $M^*_i = b$. Conceptually, the attacker commits which bit to attack beforehand (we will get rid of this later).

We reduce it to attacker B which gets $y = F(X)$ and its goal is to get the preimage X. A sets $y_{ib} = y$ and choose xi', b' at random for $(i', b') \neq (i, b)$ and $y_{i',b'} = F(X_{i',b'})$. So B knows the other components of the public key, because he constructed it himself. B then sends pk to A.

B knows all but one component of the secret key, but that's enough for answering all possible query from A. When A sends back $M*, \sigma^*$, B knows the preimage it wants to know.

To remove the i, b commitment, B simply guess them at random and abort if guessed incorrectly. B's guess has $1/2n$ chance being correct. So if A has non-negligible success rate ε, B has $\varepsilon/2n$ chance.

3.3 Towards many-time signatures: chaining

However, Lamport signature is only one-time secure and not even two-time secure. How to make it many-time secure? The solution is signature chaining.

Let us denote we already have a one-time secure signature scheme Gen_{OT} and $Sign_{OT}$. We now construct a many-time secure signature scheme Gen and $Sign$.

We first generate a pair of key $sk, pk \leftarrow Gen_{OT}(1^\lambda)$.

For the first signing request $Sign(sk, M_1)$: we get another pair of keys $sk_1, pk_1 \leftarrow$

$Gen_{OT}(1^\lambda)$ and let $\sigma_1 \leftarrow Sign_{OT}(sk, (M_1, pk_1))$, then output (pk_1, σ_1). Let the algorithm be stateful and remember $M_1, sk_1, pk_1, \sigma_1$.

For the second signing request $Sign(sk, M_2)$, we get another pair of keys $sk_2, pk_2 \leftarrow Gen_{OT}(1^\lambda)$, and $\sigma_2 \leftarrow Sign_{OT}(sk, (M_2, pk_2))$. Output $(pk1, \sigma_1, pk2, \sigma_2)$.

For verification, we need to verify the entire chain by running both $Ver_{OT}(pk, (m_1, pk_1), \sigma_1)$ and $Ver_{OT}(pk_1, (m_2, pk_2), \sigma_2)$.

Since pk_1 is stapled into the first signature, to forge the second signature the attacker must come up with pk_1, i.e., the signature of pk_1 signed by pk.

This scheme has some problem. First it's stateful. Also, the domain of the message space has to be bigger than the public key space. In the lamport signature case, the public key space is larger than the message space.

Lamport signature only does OWF evaluation, and is very fast (which is useful in practice when only one-time security is needed). In practice, we use other scheme for fast many-time secure signature scheme.

For performance problem and message space size, we usually just sign the hash of the message, as long as we have good properties of the hash. Hashing helps shrink the public key size.

So far we build a long chain of messages. Instead, we can build a tree. Each secret key can sign a pair of public keys, and this gives us a binary tree structure. To sign a message, pick a leaf node of the tree, and the signature chain is the path to root, with length is logarithmic. We can also generate the entire tree using PRF and just remember the single PRF key (as part of sk), so we don't need to keep any state.