COM S 6830 – Cryptography Nov, 17, 2011

Lecture 20: Authentication - Part 2

Instructor: Rafael Pass *Scribe: Shentong Wang*

Review: One Time Digital Signature

During the last class, public key signatures were defined A construction was also provided for a One Time Public Key Signature, which restricted the attacker by only allowing the attacker to query the signature of only one other message. This attacker is then tasked with fabricating the signature of any message which is different from the one he/she already queried for. The construction look as follows:

1. $Gen(1^n) : (sk \leftarrow (x_1^0, x_2^0 ... x_n^0, x_1^1, x_2^1 ... x_n^2), vk \leftarrow (f(x_1^0), f(x_2^0) ... f(x_n^0)), f(x_1^1), f(x_2^1) ... f(x_n^1)))$
2. $Sign(s_k, m) : (x_1^{m_1}, x_2^{m_2} ... x_n^{m_n})||m$
3. $Ver(v_k, (y_1, y_2 ... y_n)||m) : (f(y_1) = f(x_1^{m_1}) \wedge f(y_2) = f(x_2^{m_2}) ... f(y_n) = f(x_n^{m_n}))$

This main problem with this construction is that the public key and signed message is much longer than the message itself, because for every bit of the message, an entry twice the size of the input to the one-way function is needed. Thus, it would be desirable to shrink the message beforehand in order to allow shorter verification keys and signatures.

Hash Function

A Hash Function will be used as the means to shrinking the message enough in order to allow for short verification keys and signatures without any loss of secrecy. Currently, there are no known constructions of good hash functions based on One Way Functions. Even though there are valid constructions of public key signatures based solely on One Way Functions without using hash functions, these constructions are tedious and beyond the scope of the course. For now, more assumptions will be made beyond the existence of One Way Functions which will allow for the construction of hash functions. A family of hash functions, $\{h_i : D_i \rightarrow R_i\}$, has the following properties:

1. Input shrinking: $|D_i| > |R_i|$ (For the needs of public key signatures, only shrinking by 1 bit is sufficient, because this hash function can be iterated over and over again to allow for arbitrary shrinking)
2. Hard to fabricate collisions: For all non-uniform machines, $\mathcal{A}$, the probability of finding two collisions for a randomly sampled hash function from $\{h_i\}$ is negligible.

Birthday Attack

The requirements for breaking a hash function requires that an attacker only is able to find any two inputs linked with the same output. This is a much easier than the requirements for an attacker in the One Way Function case. In the previous case, in order for an attacker to invert a Strong One Way Function, he/she would have to use brute force on all the inputs, which would result in a $O(2^n)$ computation time. In the case of a hash function, even if an attacker is pitted against a good hash function which satisfies the previous properties, he/she can break it in much faster time at the cost of meomry usage. (Though this attacker will still use exponential time) The Birthday Attack for any hash function, $\{h_i : D_i \to R_i\}$, where $|D_i| = |R_i| + 1 = n + 1$, goes as follows:

1. Sample $2^{\frac{n+1}{2}}$ inputs from D_i.
2. Compare whether any of the two sampled inputs collide. (This can be done in time linear to the number of sampled inputs by using a hash table)

It's not intuitive, but it can be shown using the linearity of expectations that an attacker only has to sample an amount of inputs equal to the square root of the total number of inputs in order to find an expected collision. The probability of any two inputs resulting in a collision is at least $\frac{1}{2^n} - \frac{1}{2^{n+1}}$. (The probability of any two inputs collide is equal to $\sum_{y_i \in R_i} (Pr(x \leftarrow D_i : f(x) = y_i))^2 \geq \frac{1}{2^n}$. The probability the two selected inputs were the same is $\frac{1}{2^{n+1}}$.) Thus, by linearity of expectations, the expected number of collisions results in being $\binom{2^{\frac{n+1}{2}}}{2} \frac{1}{2^{n+1}} \approx (2^{n+1})^2 \frac{1}{2^{n+1}} = 1$. This not only provides us with a way to break a hash functions (abeit still inefficiently), but also a way to break any assumptions which can be used to create a hash function.

Hash Function from Discrete Log Assumption

Given the Discrete Log Assumption, a hash function can be constructed as follows:

1. $Gen(1^n) : (g, p, y)$, where p is a n-bit prime, $y \in Z_p^*$, and g is a generator for Z_p^*.
2. $h_{g,p,y}(x||b) = y^b g^x \bmod p$, where $b \in \{0, 1\}$, and $x \in Z_p^*$.

Note that for any two inputs, where the last two bits are identical (the b portion), then their first n bits (the x portion) must also be identical, because g is a generator and $(g^{x_1} = g^{x_2}) \Rightarrow (x_1 = x_2)$. Assume for contradiction that there exists a machine $\mathcal{A}$, which breaks the hash function. We can create another machine, $\mathcal{A}$' which breaks the discrete log assumption as follows:

1. Given (g, p, y), $\mathcal{A}$' gives $\mathcal{A}$ input (g, p, y) which represents the hash function $h_{g,p,y}$.
2. $\mathcal{A}$' receives $x_0||0$ and $x_1||1$ as output from $\mathcal{A}$. (Their last bit must differ as shown above)
3. Since $g^{x_0} = g^{x_1}y$, $y = \frac{g^{x_0}}{g^{x_1}}$. Therefore, $g^{x_0 - x_1} = y$. Thus, output $x_0 - x_1$.

Since a hash function can be constructed based on the Discrete Log Assumption, there also exists a birthday attack on the Discrete Log Problem by converting it into a hash function and running the Birthday Attack on it. This kind of attack can be extended to other thought to be hard problems.

Hash and Sign

Thus, given a secure public key signature scheme, $(Gen, Sign, Ver)$, it can be transformed into a shorter signature scheme assuming the existence of hash functions. Let $\{h_i : \{0,1\}^* \to \{0,1\}^n\}_{i \in I}$ be a collision resistant hash function, define a public key signature scheme, (Gen', Sig', Ver'), as follows:

1. $Gen'(1^n) : (sk, vk \leftarrow Gen(1^n); i \leftarrow Gen_h(1^n) : sk' \leftarrow (sk, i), vk' \to (vk, i))$
2. $Sign'(sk', m) : Sign(sk, h_i(m))||m$
3. $Ver'(vk', \sigma||m) : Ver(vk, h_i(m))$

Since h_i reduces the input from an arbitrary size to length n, this scheme produces a verification key and signature that is order constant size with respect to the size of the message. A detailed proof will not be provided for this construction. The general gist goes as follows:

1. Assume for contradiction there exists a machine which breaks this scheme.
2. The message that this machine fabricates must either hash to the same value as the signed message provided as its one time request or to a different value.
3. If the fabricated message hashes to the same value as the given message, then the hash function h_i can be broken using this machine.
4. If the fabricated message hashes to a different value than the given message, then the provided encryption scheme $(Gen, Sign, Ver)$ can be broken using this machine.

Constructing a Stateful Public Key Signature Scheme

Using the One Time Secure Scheme, a stateful public key secure scheme can be constructed by generating a secret key and public key, and including the public key in the message signed by the sender. This new secret key and public key will be used for the next message. Thus, the Sign subroutine will be expanded by including a call to Gen to setup for the next communication. This scheme looks similar to the following diagram:

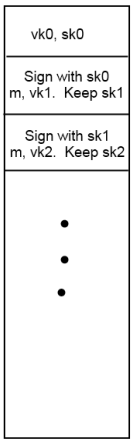

Figure 1: **Stateful Signature Scheme**

This construction has some noticeable problems:

1. This construction requires state, which is undesirable to coordinate in practice.
2. In order for the message receiver to not have to keep state, this construction requires that the entire chain of signatures leading up to the original verification key be sent in each message. (If the receiver does not keep state of which signatures have already been received, it will have to reference the chain to figure out that the signature is valid because of the public key sent in the previous signature which is valid because of the public key sent in the signature before the previous...and so on.) This requires each signature to be linear in the length of the previous signatures sent, which is also undesirable.

A Smarter Construction

Similar to the pseudo-random function construction, a linear approach is almost never optimal in these cases. Thus, we will resort to a tree approach for these construction.

Instead of only sending 1 more public key in every signed message, the sender can send 2 more public keys. These public keys represent how the progression of public keys will branch in the linear case. This construction will form a construction similar to the following diagram:

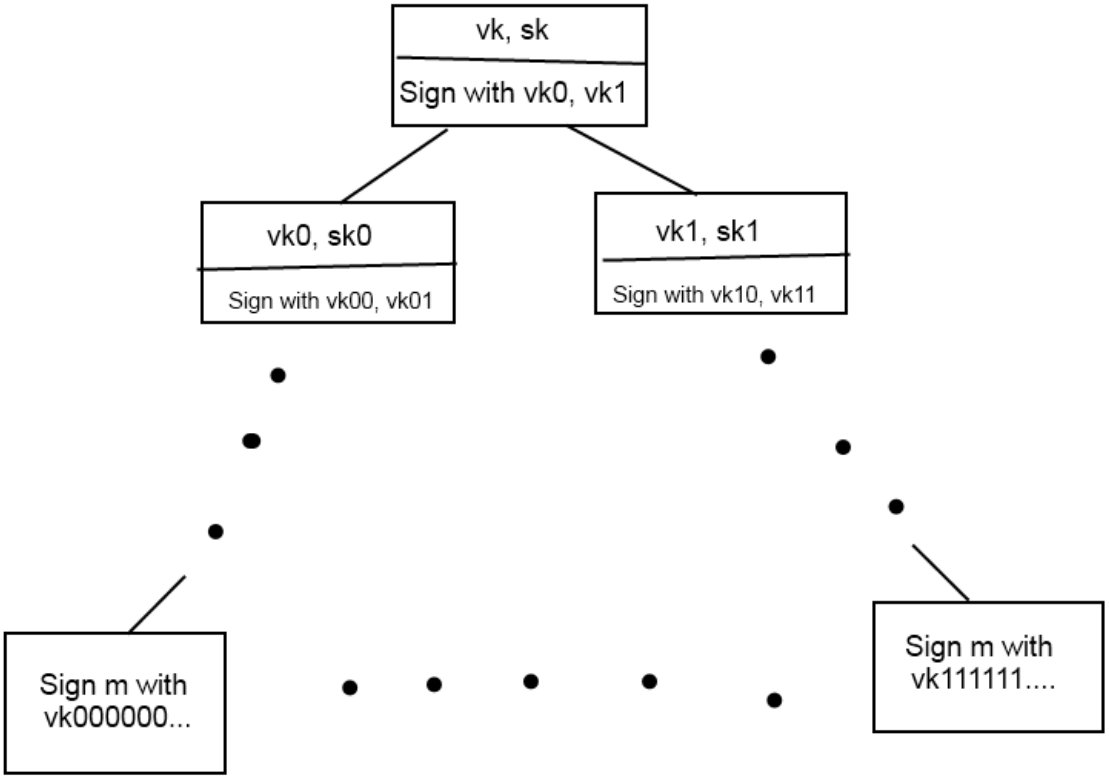

Figure 2: **Tree Stateful Signature Scheme**

This construction requires the signer to send chain of $log(n)$ signatures which is the depth of the tree. (For example: In order to send the 0100011^{th} message, the sender will have

to send the signature chain including $[vk_0, vk_{01}, vk_{010}, vk_{0100}, vk_{01000}, vk_{010001}, vk_{0100011}]$ in order to prove the validity of public key $vk_{0100011}$.) Thus, the total required state which needs to be kept by the sign are:

1. The number of messages, which determines which leaf of the tree he/she will use as the next verification key.
2. The previous keys, which allows him/her to retrace down the branch of the tree when necessary to get the next leaf.
3. The previous signatures, which he/she will have to resend for messages which share common parents. (Note that he/she cannot just regenerate the signatures, or the one time secure scheme used at each level might be broken)

A Stateless Construction

With a little clever tweaking, the signer will not have to keep track of the state in the signature scheme.

1. In order to forgo the number of messages, the signer can instead, index on the message instead of the message number. (This will require a chain as long as the message, but that can be shortened using a hash function)
2. In order to forgo the previous keys and previous signatures, the signer can use a PRF for each level of the tree to generate the randomness which will be used in Gen and Sign to recreate the verification key, secret key, and the signature. Thus, the signer will only have to keep track of two seeds, s_0 and s_1. Whenever a secret key and verification key is needed for a node in the tree at spot m, he/she computes $Gen_{f_{s_0}(m)}(1^{|m|})$. Whenever a signature is needed for a node in the tree at spot m, he/she computes $Sign_{f_{s_1}(m)}(sk_m, vk_{m||0}||vk_{m||1})$. By using a PRF to generate the randomness for Gen and Sign, each entry of the tree can be acquired when needed deterministically each time. (But to an attacker, the values all look random from the PRF)