1 Lecture 6: Digital Signatures

What we are looking for in this lecture is some way for us to sign documents,confirming our identity and our consent/acknowledgment of something, without someone being able to forge our signature.

The first (obvious) thing to notice is that computers can copy bits by reading them in and spitting them out, so the signature has to change in some way. And, we don't want signatures to be able to moved from document to document by an adversary (think about the chaos this could cause!). So, we are going to solve both of these by having the signature depend on the document we are signing.

Formally, we define a "**signature scheme**" as a protocol consisting of 3 **PPT** algorithms:

1. **Keygen** takes in a security parameter k and some randomness. In return, it produces a *public key* PK, which is to be shared with everyone, and a *signature key* SK, or private key, which is to be shared with no one other than the signer.

2. **Sign** takes a document D, both keys PK and SK, and some randomness. In return, it produces a signature $\sigma(D)$, which corresponds directly to the document being signed.

3. **Verify** takes the publickey PK, a document D, and a signature $\sigma(D)$. In return, it returns a decision (accept/reject), accepting i.f.f. $\sigma(D)$ is the proper signature of the document.

Additionally, these 3 functions must together satisfy the following criteria:

1. **Correctness**: if Keygen and Sign are run properly, then Verify will accept all documents and their corresponding proper signatures.

2. **(Existential) Unforgeability**: No **PPT** adversary can win at the following game except with negligible probability:

This game is played between a challenger C who will sign documents, and an adversary A who will attempt to forge C's signature on some new document.

0. C runs Keygen and sends PK to A

1. For some polynomial number of trials: A sends a document D to be signed, and C sends back the true proper signature $\sigma(D)$

2. A wins if it can, after polynomially many trials, send a document D^* and a signature $\sigma(D^*)$ such that:

   (a) D^* was not sent in any of the trials

   (b) Verify(PK, D^*, $\sigma(D^*)$) accepts

We add two notes to this.

First, this game raises the concern that we should not be able to generate a valid signature of $\sigma(D_1||D_2)$ from valid signatures of D_1 and D_2, or else any adversary could simply ask for $D_1 = 0$ and $D_2 = 1$ and then properly sign any document.

Second, we add the note that a "**strong signature scheme**" is one for which an adversary cannot come up with a different legal signature for D, given D and some valid signature $\sigma(D)$. (Note that this is possible, since the signing scheme involves randomness at every step.)

1.1 Trapdoor Permutations

We have seen 1WP's, but now we consider a different type of permutation which is hard to invert: a "*trapdoor permutation*". The idea is that for an arbitrary **PPT** adversary, the trapdoor permutation should be as hard to invert as a 1WP. But if an adversary is given access to some trapdoor t, then it would be trivial for them to invert the permutation.

But as always, we are considering all adversaries, so we have to account for the adversary with the trapdoor hard-coded into it. In fact, since we are considering all polytime adversaries, we have to consider any adversary with polynomially many of these trapdoors for different permutations hard-coded into it.

The solution to this issue is to use an exponentially large family of trapdoor permutations, rather than a single one. Then, each time we need to do something with a trapdoor permutation, we randomly sample from the family. Then, the probability that any given adversary knows the trapdoor for any particular use of our family is negigible, as desired.

1.1.1 A (Faulty) Idea

Now, with a family of trapdoor permutations in hand, we might try the following signature scheme:

1. Keygen picks a trapdoor permutation f from the family, and assigns PK to be f and SK to be the trapdoor t
2. Sign simply uses the trapdoor to compute f^{-1} of the document.
3. Verify can compute f, given the signature, and checks that it matches the document

The problem with this is very subtle and lies in something that we did not explicitly state here, as we defined the "hardness" of inverting trapdoor permutations based on inverting 1WP's. The trouble is that an important aspect of the hardness of 1WP's, and thus trapdoor permutations, is that we generally use them on *random* inputs. Documents, on the other hand, are quite certainly *not random*.

In fact, we have a way for an adversary to beat our game, if we use this (faulty) signature scheme. Without even having to use any trials, the adversary can pick some random signature s and compute a corresponding document $D = f(s)$. This document may well be nonsense, and it might not actually be useful for an adversary to do this. But this document-signature pair will pass the verifier, so the adversary has beaten the game, and beats it with probability 1, which is definitely not negligible.

1.2 Lamport 1-Time Signatures

For now, we are going to put trapdoor permutations aside, and return to 1WP's.

We now examine the following signature scheme, which makes use of a 1WP f.

Keygen takes a positive integer m (the document length) and uses f to generate the following signature-key and public-key tables:

$$SK = \begin{array}{|c|c|c|c|c|} \hline x_1^0 & x_2^0 & x_3^0 & \cdots & x_m^0 \\ \hline x_1^1 & x_2^1 & x_3^1 & \cdots & x_m^1 \\ \hline \end{array}, \quad PK = \begin{array}{|c|c|c|c|c|} \hline f(x_1^0) & f(x_2^0) & f(x_3^0) & \cdots & f(x_m^0) \\ \hline f(x_1^1) & f(x_2^1) & f(x_3^1) & \cdots & f(x_m^1) \\ \hline \end{array}$$

The x's are random strings of some set length for which the 1WP is hard to invert. Note that this means that the table will end up being quite large, since 1WP's (and 1WF's) are really only difficult to invert on sizable inputs.

Then, we sign an m-bit document bit-by-bit using these tables. If our document is $D = b_1 b_2 \cdots b_m$, our signature is: $\sigma(D) = x_1^{b_1} x_2^{b_2} \cdots x_m^{b_m}$.

The verify function simply plugs in each of the $x's$ from the signature to f and checks that the resulting string is in fact, equal to $f(x_1^{b_1} f(x_2^{b_2}) \cdots f(x_m^{b_m})$, as specified by the public key.

1.2.1 Security of Lamport 1-Time Signatures

We will show that this signature scheme is as secure as the 1WP it uses, against an adversary who may request the signature of one document before attempting to commit a forgery (hence the "1-Time" in the name of the signature scheme). As usual, we do this by contrapositive.

Suppose we have some adversary $A \in \mathbf{PPT}$ that, with non-negligible probability ε, can win the game described above, by using only 1 trial before creating a forgery.

Then, we create a new function A' which does the following:

1. Choose some y which is the same length as any of the entries in the signing and public keys.
2. Place y randomly into the public key table, leaving the corresponding signing key entry empty.
3. Allow A to request the signature of some document D. If A asks for the signature of the bit corresponding to y, start over (since we do not know this signature, $f^{-1}(y)$). This happens with probability $\frac{1}{2}$.
4. When (3.) is successful, get a forgery from A. The forged document must differ from D in at least one bit, so we have a probability $\frac{1}{m}$ that the forged document differs from D in the column where y is, meaning that A gives us $f^{-1}(y)$ in the signature of the forged document.

Note that since the signing key is completely randomly generated and f is a 1WP, no adversary of this type will ever be able to tell apart a real public key from a public key where we have replaced a random entry with some random y. And, A' inverts f on any such random y with probability $\frac{1}{2n} \cdot \varepsilon$, meaning that we have broken the 1WP.

Thus, the scheme is as secure as the 1WP we choose.

1.3 Extending Lamport's Scheme

The main problem(s) with the scheme outlined above are that we have to generate a ton of new random strings for every new document that we want to sign, and it can only be used for one document at a time; if we want to sign a new document, we have to go

We deal with the first problem by using "**Collision-Resistant Hash Functions**" so that our keys don't have to be so large, and then we deal with the second problem with a key-refreshing/generating scheme proposed by Naor and Yung.

1.3.1 Collision-Resistant Hash Functions

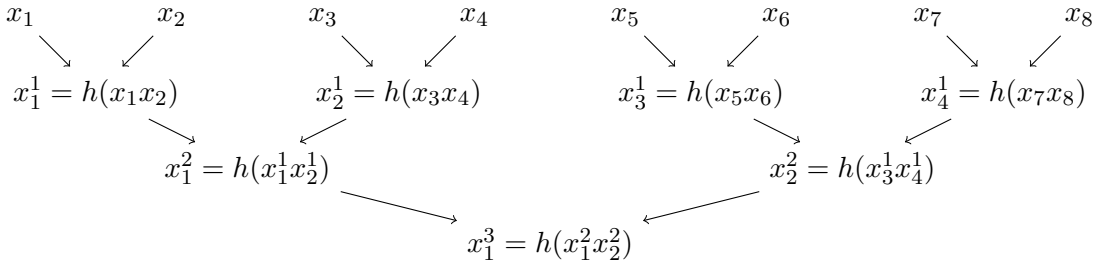
As we ran into with PRF's and Trapdoor Permutations, we find here that in order to have something that really fits our needs, we are going to have to look at *families* of hash functions, rather than a single hash function.

Then, we define a family of hash functions to be collision-resistant if any **PPT** adversary cannot win the following game between challenger C and adversary A, except with negligible probability:

0. We define a family of hash functions, H.
1. C randomly chooses a hash h from the family and sends it to A.
2. A wins if it can send a, b such that $h(a) = h(b)$

We use this mechanism to shorten our keys by hashing our documents down to a manageable length, and then signing the reuslting hash rather than the entire document.

And, even though hash functions take inputs of a given length n and hash these inputs down to a shorter length m, we may hash documents of arbitrary length down to length m by using a Merkle hash tree. For a document with $4n < m \leq 8n$ bits, and a hash function h that takes $2n$ bits to n bits, we could pad the document to $8n$ bits to get a Merkle tree that looks like the following:



Note that hashing with this kind of a tree is as strong as only applying h once, as we can see that a collision for the whole tree implies a collision somewhere within the tree.

This is because two different inputs to the tree must necessarily result in different trees. But if they cause a collision, this means that there is some first level $0 < k \leq m$ (going from top= 0 to bottom= m) where the two trees are exactly the same. This means that between levels k and $k-1$, there was a collision somewhere. So anything that can produce a collision in the larger tree must necessarily produce a collision for the individual hash function.

1.3.2 "Refreshing" the keys

The idea propsed by Naor and Yung is a simple one, and it essentially involves chaining multiple keys together using a linked-list mechanism.

With hashing in mind, we create a first public key of length $2n$, which is twice the length that we would need due to our hashing family. Then, we may sign a document using the first half of this key.

The second half of the key will be reserved for signing a new key, which will be hashed down and then signed. We can continue this indefinitely, but note that by this mechanism, we aren't doing a whole lot better than before, since each key is only generating one new key, so we have linear growth.

If, instead, we used each key to sign two new keys (i.e., used both the first and second halves to sign different keys), then we would have exponential growth in the number of keys we have verified for our own use, which puts us in a much better spot. This is the mechanism/scheme that Naor and Yung proposed.

Note that with all this in mind, we may use a PRF to generate all the keys in our tree, and then we may sign and validate them all using this scheme. This works because as we have previously shown, PRF's are just about as good as true randomness.