Today’s Agenda

- Wrap Up: Collision-resistant hash functions
- Definition
- Constructions

1 Definition

**Definition 1** (CRHF Family). \((\text{ParamGen}, \text{Hash})\) is a family of collision-resistant hash functions (CRHF) if

1. **Collision-resistance:** \(\forall \text{ PPT } A \exists \text{ negl. } \nu() \text{ such that}

\[
\Pr[\text{params} \in \text{ParamGen}(1^k) ; x, y \leftarrow A(1^k, \text{params}) : x \neq y \text{ and } \text{Hash}_{\text{params}}(x) = \text{Hash}_{\text{params}}(y)] = \nu(k)
\]

2. **Length-reducing:** For all \(\text{params} \in \text{ParamGen}(1^k),

\[
\text{Hash}(\text{params}, \cdot) : \{0,1\}^* \mapsto \{0,1\}^k
\]

The second property of hash functions can also be stated as length-halving meaning that for all \(\text{params} \in \text{ParamGen}(1^k),

\[
\text{Hash}(\text{params}, \cdot) : \{0,1\}^{2k} \mapsto \{0,1\}^k
\]

In the last class, we saw a generic construction of a hash function from a hash function that satisfies the property described above.

2 Discrete Log-based construction

Let us look at the high-level idea for DL-based construction. As our parameters, we have group \(G\) of prime order \(q\) with generators \(g, h\).

\[
\text{Hash}_{g,h,q} = g^{x_1}h^{x_2} \text{ for } x_1, x_2 \in \mathbb{Z}_q
\]

We claim that if the DL assumption holds in \(G\) then \(\text{Hash}\) described above is a CRHF. Recall that the DL assumption states that on input \((G, g, h, q)\), it is hard to find \(\alpha\) such that \(h = g^\alpha\). Hard to find also means that for all PPT \(A \exists \text{ negl. } \nu() \text{ such that } \Pr[A(G, g, h, q) = \alpha : h = g^\alpha] = \nu(k)\)

So we want to say that for this hash function, \(A\) cannot find a collision \((x_1, x_2) \neq (x'_1, x'_2)\) such that \(g^{x_1}h^{x_2} = g^{x'_1}h^{x'_2}\). Let \(x_1 \neq x'_1\) then \(g^{x_1} = \frac{g^{x'_1}h^{x'_2}}{h^{x_2}}\). Hence \(g^{x_1 - x'_1} = h^{x'_2 - x_2}\). Finding a collision would be equivalent to breaking the DL assumption.

Note that in a prime-order group, every element except the identity is a generator.
2.1 Construction

DL assumption does not hold in $\mathbb{Z}_q$. So pick a random $k$-bit prime $q$ and try till $p = 2q + 1$ is also prime. Output our group $\mathbb{G} = QR_p$. Output $g, h$ as random elements of $QR_p$. (Just pick $a, b \leftarrow \mathbb{Z}_p^*$, let $g = a^2$ and $h = b^2$). We know that $g, h$ generated this way are generators because group $QR_p$ is of prime order $q$ and any element except the identity is a generator in that case.

$\text{Hash}_{QR_p, g, h, q} : \mathbb{Z}_q \times \mathbb{Z}_q \mapsto QR_p$ with $\text{Hash}_{QR_p, g, h, q}(x_1, x_2) = g^{x_1} h^{x_2}$. So we map a string of length $2k - 2$ to length $k + 1$. So the hash function is length-reducing as required. We also have collision-resistance due to the DL assumption.

3 Merkle Hash Tree

Our goal is that on input of length $2^l \cdot k$, output a $k$-bit string: Let $x_1, \ldots, x_{2^l}$ be the input such that each $|x_i| = k$. $x_1, \ldots, x_{2^l}$ form the leaves of the hash tree, denoted by $h_{1^l}, \ldots, h_{2^l}$. In general, we have $h_{1^j} = x_i$ and for $0 \leq j \leq l - 1$ and $1 \leq i \leq 2^l$,

$$h_{i^j} = \text{Hash}(h_{2i^j+1}, h_{2i^j})$$

Output $h^0$ which is the root value of the merkle tree.

Claim: Let Merkle-hash be the algorithm that on input $x_1, \ldots, x_{2^l}$ for $x_i \in \{0, 1\}^k$ outputs $h^0$ the root of the Merkle tree. Then Merkle−Hash$_{params} : \{0, 1\}^{2^l \cdot k} \mapsto \{0, 1\}^k$ is collision-resistant.

(The proof of this claim will be on the homework.)

Applications of merkle-tree: Suppose you as a client want to store some big chunk of data on a server and retrieve parts of it at later points of time. What if the server is dishonest and sends you random chunks of data when you query without storing your actual data. You need to store something as a proof, suppose you store the merkle hash of your data. One solution is that the server as a proof sends you the whole data when you make a query, then you can hash the values and check that you get the root value that you have stored.

But this requires you to store huge amounts of data. There is a more efficient way of doing the same thing with the Merkle-tree. It gives a way for the client to verify the data: Client stores the
root hash, client want to retrieve \( x_{01} \) let’s say. The server gives the authenticating path which is \( u_{00}, u_1 \), in general the sibling value for each node on the path from \( x_{01} \) to the root. The client can now easily verify whether root hash = \( H(H(u_{00}, H(x_{01})), u_1) \)

4 Digital Signatures

Digital signatures are a widely used primitive in cryptography. The scenario is that Alice has a signing key \( sk \) which is her private key which lets her sign documents. Correspondingly, there is a public verification key \( vk \) which lets anyone, in particular Bob verify signatures signed by Alice. Here, we want to protect our digital signature scheme from an adversary who is trying to forge signatures. So an adversary \( A \) can have the following abilities:

1. Knows verification key \( vk \)
2. Observes message-signature \((m, \sigma)\) pairs from Alice
3. Get signatures on messages of his/her choice.

Correspondingly, the adversary might have one of the following goals:

1. Forge Alice’s signature on \( m \) of his/her choice. (Selective forgery)
2. Forge a signature \( \sigma \) for any message \( m \) that Alice did not sign. (Existential forgery)

We want that the most powerful \( A \), the adversary with the most abilities cannot even achieve the most modest goal of existential forgery. That would be the strongest notion of security. We will see constructions of digital signatures after the spring break.