EE 374: Blockchain Foundations **Stanford, Spring 2022**

Lecture 3: Lecture 3: Hashses, Signatures, Ledgers

April 6, 2022

Lecturer: Dr. Dionysis Zindros Scribe: Kenan Hasanaliyev, Kaylee George, Koren Gilbai

1 Hash Functions

We use hash functions to represent objects in the network. A hash function is a mathematical function that takes in an arbitrary string input and outputs a unique identifier of that string: $\mathcal{H} : \{0,1\}^* \to \{0,1\}^\kappa$. In implementation, we use the SHA-256 hash function ($\kappa = 256$), meaning that the outputted address is of length 256.

Gossiping strategy:

1. Upon learning of an object, advertise that you have it by its objectid, which is the hash of the object (meaning that we send it to all peers).
2. Upon seeing an objectid, request it if you don't have it.

Note: It is important that each object has a *unique* identifier (one-to-one), since in the Gossip protocol, instead of sending an entire object, we signal ownership of it by sending its hash.

However, since $|\kappa|$ may be $\leq *$, by the Pigeonhole Principle, $\exists x_1, x_2 : H(x_1) = H(x_2)$. In other words, collisions exist because the input space is larger than the output space.

However, while collisions exist, they are hard to find which we will show.

To find a collision without polynomial time restriction, we can use a brute-force method:

Algorithm 1 Brute force collision finding

```
function BRUTE FORCE COLLISION FINDING($1^\kappa$)
    for $i \in \{0, \dots, 2^\kappa\}$ do
        for $j \in \{i+1, \dots, 2^\kappa\}$ do
            if $H(i) = H(j)$ then
                return $(i, j)$
            end if
        end for
    end for
end function
```

The three properties we want to achieve are as follows:

Algorithm 2 Collision Game

```
1: function $\mathsf{COLL}_{\mathcal{A}}(\kappa)$:
2:     $x, x' \leftarrow \mathcal{A}(1^\kappa)$                ▷ Invoke adversary for both inputs
3:     return $x! = x' \wedge H(x) = H(x')$
4: end function
```

In Alg 2, we invoke the adversary for both inputs and return a collision. The adversary is successful if she returns $x \neq x'$ such that $H(x) = H(x')$.

Algorithm 3 2nd-preimage Game

```
1: function $\mathsf{2PRE}_{\mathcal{A}}(\kappa)$:                ▷
2:     $x_1 \leftarrow R\{0,1\}^{2\kappa+1}$                ▷ Given $x_1$, adversary finds second input
3:     $x_2 \leftarrow \mathcal{A}(x_1)$
4:     return $x_1! = x_2 \wedge H(x_1) = H(x_2)$
5: end function
```

In Alg 3, the adversary is given a randomly sampled $2\kappa + 1$ bit input x_1 by the challenger. The adversary is successful if she can come up with an $x_2 \neq x_1$ that hashes to the same value as x_1, i.e. $H(x_1) = H(x_2)$.

Algorithm 4 Preimage Game

```
1: function $\mathsf{PRE}_{\mathcal{A}}(\kappa)$:                ▷
2:     $x \leftarrow R\{0,1\}^{2\kappa+1}$
3:     $y = H(x)$
4:     $x' \leftarrow \mathcal{A}(y)$                ▷ Give $\mathcal{A}$ output $y$
5:     return $H(x') = y$
6: end function
```

In Alg 4, the adversary is given the hash of a randomly sampled $2\kappa+1$ bit input and is successful if she can find an x that hashes to the given hash.

A hash function is easy to compute but hard to invert, meaning that $x \rightarrow y$ is easy (polynomially computable) while the inversion of the hash function, $y \rightarrow x$, is hard.

Theorem 1.1. *If H is collision resistant, then H is 2nd-preimage resistant. More formally, $\forall$ PPT (probabilistic polynomial time) $\mathcal{A}$: $Pr[\mathsf{2PRE}_{\mathcal{A}}(\kappa) = 1] \leq negl(\kappa)$.*

Proof. For the sake of contradiction, suppose that H is not 2nd-preimage resistant. Thus, there exists some adversary $\mathcal{A}$ such that $Pr[\mathsf{2PRE}_{\mathcal{A}}(\kappa)] = p$, where p is non-negligible. We construct an adversary $\mathcal{A}'$ against Collision game (Alg. 5).

$Pr[\mathsf{COLL}_{\mathcal{A}'}(\kappa) = 1] = Pr[\mathsf{2PRE}_{\mathcal{A}}(\kappa) = 1] =$ non-negligible $\implies Pr[\mathsf{COLL}'_{\mathcal{A}}(\kappa) = 1]$ is non-negligible. This is a contradiction, hence, H is 2-nd-preimage resistant.

Algorithm 5 $\mathcal{A}'$ adversary

```
1: function $\mathcal{A}'(1^\kappa)$:
2:     $x_1 \leftarrow R\{0,1\}^{2\kappa+1}$
3:     $x_2 \leftarrow \mathcal{A}(x_1)$
4:     return $x_1, x_2$
5: end function
```

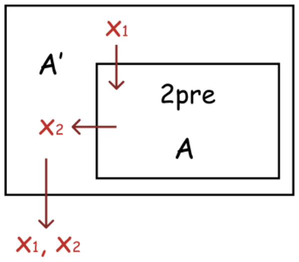

Figure 1: 2nd-preimage game visualization: Follows 5 construction, in which $\mathcal{A}'$ produces x_1 and gives x_1 to $\mathcal{A}$. Then, $\mathcal{A}$ provides x_2 and $\mathcal{A}'$ returns x_1, x_2.

□

Theorem 1.2. *If H is 2nd-preimage resistant, then H is preimage resistant.*

Proof. For the sake of contradiction, suppose that H is not preimage resistant. Then, there exists an adversary against preimage. We construct $\mathcal{A}'$ against 2nd-preimage resistance 6.

Algorithm 6 $\mathcal{A}'$ Construction

```
1: function $\mathcal{A}'(1^\kappa)$:
2:     $x \leftarrow R\{0,1\}^{2\kappa+1}$
3:     $y \leftarrow H(x)$
4:     $x' \leftarrow \mathcal{A}(y)$
5:     return $x' \neq x_1 \wedge H(x') = H(x)$
6: end function
```

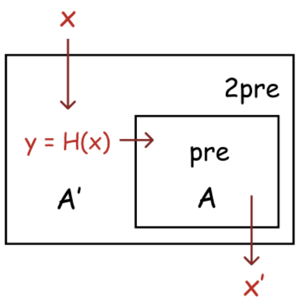

Figure 2: Preimage game visualization: Follows 6 construction, in which $\mathcal{A}'$ produces $y = H(x)$ given x. Then, give y to $\mathcal{A}$ to produce x'.

We hope $x' \neq x$. Partition the input space into "boxes" (y: output value) containing dots (x: input value) that all hash to the same value. Sort boxes by their sizes (number of inputs) in ascending order, in which the larger boxes are on the left. The purpose of this is that we want to argue that it's harder to guess a dot in a larger box.

Next, we split into two halves of equal number of inputs ($2^{2\kappa}$ dots each half). It is impossible to have $< 2^{\kappa}$ dots in a "box" on the left half because then each box on the right half has $< 2^{\kappa}$. There are $< 2^{\kappa}$ boxes on the right half, each with $< 2^{\kappa}$ dots, so the total dots in the right half is $< 2^{\kappa} \cdot 2^{\kappa} = 2^{2\kappa} \implies$ there are $< 2^{2\kappa}$ dots. So, all the left-half boxes are "large". The probability that we sampled an input on the left half $= \frac{1}{2}$ (sampling uniformly). The y value of the box of our input will have $\geq 2^{\kappa}$. The adversary has no better strategy than guessing uniformly at random from the box.
Thus, there holds: $Pr[\mathsf{2PRE}'_{\mathcal{A}}(\kappa)] \geq Pr[\mathsf{PRE}_{\mathcal{A}}(\kappa) = 1](\frac{1}{2})(1 - 2^{-\kappa}) =$ non-negligible * constant * value close to 1 = non-negligible. So $Pr[\mathsf{2PRE}'_{\mathcal{A}}(\kappa)]$ is non-negligible. This implies that 2PRE security is broken, in contradiction. □

2 Transactions and Signatures

Verifiability of money: If money is created correctly, money has verifiability property such as ownership validation. Here, we discuss a transfer of ownership.

In blockchain networks, a transaction is an entry in local state. Each node keeps a ledger, which is a recorded sequence of transactions ("tx"s). Ledgers are used to verify and update the system state, a record of the balances of all parties in the system, such that consensus across nodes is maintained.

Assume we have parties A, B, C, etc. Each party can have some sort of money and each party will share all financial information. An example of a transaction could be party A saying "Give 5 units to party B." The following is an outline of ledger/transaction procedures:

1. Initial system state: some distribution of money.
2. Issue a tx by broadcasting
3. Upon receiving a tx:
   a) Check validity (source has enough money)

   b) Append to local ledger

4. Read ledger to determine balances (system state)

However, nobody should be able to issue transactions spending money from someone's else balance. Therefore, we need to authenticate transactions. One way to achieve it is by using digital signatures.

Digital signature scheme, Alg. 7, consists of three algorithms: key generation, signing and verifying algorithms.

Algorithm 7 Private and Public Keys: Signature Scheme

```
1: $(pk, sk) \leftarrow Gen(1^\kappa)$
2: $\sigma \leftarrow Sig(sk, m)$
3: $Ver(pk, m, \sigma) \leftarrow \{\text{True/False}\}$
```

An important property of digital signature scheme is **correctness**: $\forall m : (pk, sk) \leftarrow Gen(1^\kappa) : Ver(pk, m, Sig(sk, m)) = \text{True}$. In other words, for any message, we should be able to verify using a public key and m such that $Ver() = True$. Transactions are happening between public keys and messages are signed with private key by the author, verified by other users via the author's public key.

For a signature scheme to be secure, it should be impossible for adversary to produce a valid signature without knowing a private key. More formally, a security game for digital signatures is defined as follows in Alg 8:

Algorithm 8 Signatures are unforgable

```
1: function $\mathsf{Forgery}_{\mathcal{A}}(\kappa)$
2:     $M \leftarrow \emptyset$
3:     $(pk, sk) \leftarrow Gen(1^\kappa)$
4:     $\sigma, m \leftarrow \mathcal{A}^O(pk)$
5:     return $Ver(pk, m, \sigma) \wedge m \notin M$
6: end function
```

We want to allow the adversary to have access to an oracle such that the adversary can choose an adversarial message. We use M to track the set of messages requested by the adversary. We don't want to allow the adversary to simply invoke the oracle 9 function and return a forged messaged outputted by M.

Oracle, that can be called by the adversary is defined in Alg. 9.

Algorithm 9 Signatures are unforgable

```
1: function O($m'$):
2:     $\sigma' \leftarrow Sig(sk, m')$
3:     $M \leftarrow M \cup \{m'\}$
4:     return $\sigma'$
5: end function
```

References

[1] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. `https://bitcoin.org/bitcoin.pdf`.