# CS 6160 Cryptology Lecture 13: Hash Functions

Maria Francis

October 20, 2020

# Cryptographic Hash Functions

- Basic idea: Map a long input string to a shorter output string called a digest.
- Primary requirement: avoid collisions, two inputs that map into the same digest.
- Uses: many, including HMACs for achieving domain extension for MACs.
- Hash functions are ubiquitous in crypto and are often used for properties stronger than collision resistance like for completely unpredictable outputs/ random oracles.
- They are lie in between the world of private and public key crypto – needs stronger than the existence of PRFs but a weaker assumption than the existence of public-key encryption.
- In practice, they are built using symmetric-key primitives.

# Classical Hash Functions

- Classical use of hash functions : data structures to enable $\mathcal{O}(1)$ lookup when storing a set of elements.
  - When the range of $H$ is $N$ then $x$ is stored in row $H(x)$ of a table of size $N$.
  - To retrieve $x$, compute $H(x)$ and then probe into that row of the table for the elements there.
  - A good hash functions will give few collisions where a collision is a pair $x, x^{'}$ s.t. $H(x) = H(x^{'})$.
- Collision-resistant hash functions are similar : here collision-resistance is a requirement.
- Also in data structures, elements are not chosen to make them collide. Here $\mathcal{A}$ is trying to select elements that will collide. Much harder to design collision resistant hash functions.

# Keyed functions with a difference

- A function $H$ is collision resistant if it is infeasible for any PPT algorithm to find a collision in $H$.
- Typically, domain $>>$ range. Collisions are bound to happen!
- Make it computationally hard to find them.
- Here we look at keyed hash functions. Two inputs : key, $s$ and string $x$, $H^s(x)$
- It must be hard to find a collision in $H^s$ for a randomly generated key $s$.
- Key differences:
  - Not all strings correspond to valid keys, $H^s$ may not be defined for certain $s$. Keys are generated by *Gen* and not chosen uniformly.
  - The key $s$ is not kept secret, even if $\mathcal{A}$ has $s$ collision resistance should be there.

# Definition of a Hash Function

- A hash function with output length $\ell$ is a pair of PPT $(Gen, H)$ where $Gen(1^n)$ outputs key $s$ and
- $H$ takes as input $s$ and a string $x \in \{0,1\}^*$ and outputs a string $H^s(x) \in \{0,1\}^{\ell(n)}$.
- If $H^s$ is defined only for inputs $x \in \{0,1\}^{\ell'(n)}$ and $\ell'(n) > \ell(n)$, then we say $(Gen, H)$ is a fixed-length hash function for inputs of length $\ell'$ or a compression function.

# Collision-finding experiment

$Hash - coll_{\mathcal{A},\Pi}(1^n)$:

- A key $s$ is generated by running $Gen(1^n)$.
- $\mathcal{A}$ is given $s$ and $\mathcal{A}$ outputs $x, x'$.
- Output is 1 iff $x \neq x'$ and $H^s(x) = H^s(x') \Rightarrow \mathcal{A}$ has found a collision.

A hash function $\Pi = (Gen, H)$ is collision resistant if for all PPT adversaries $\mathcal{A}$

$$Pr[Hash - coll_{\mathcal{A},\Pi}(1^n) = 1] \leq \mathrm{negl}(n).$$

# Unkeyed hash functions

- Cryptographic hash functions are usually unkeyed, i.e. just a fixed function $H : \{0,1\}^* \to \{0,1\}^\ell$.
- Then there is a problem : fixed function implies asymptotically you can always have a constant-time algo that outputs a collision: simply output $(x, x')$ hardcoded into the algo.
- We may not be able to write the explicit code for $\mathcal{A}$, since if $\ell$ is large $(x, x')$ is inaccessible, but it exists!
- For any hash function $H$ there exists some efficient adversary $\mathcal{A}$ that breaks the collision resistance of $H$.
- In case of keyed this wont happen since it is impossible to hardcode a colliding pair for every possible key using a reasonable amount of space.

# Unkeyed hash functions

- In some other textbooks, they parametrize the hash function with a security parameter. An efficient $\mathcal{A}$ must be able to compute a collision as a function of the security parameter.
- These definitions are equivalent since the key $k$ is a function of $n$.
- Practically, we look for functions where the colliding pairs are unknown and finding them is computationally difficult.
- Proofs of security for keyed hash functions are meaningful for unkeyed ones too as long as the proof shows that any efficient $\mathcal{A}$ breaking the scheme *can be used to FIND a collision in* $H$.
- This keyed Vs unkeyed hash functions is a technical issue, a result of our need for a rigorous theoretical definition.

# What are cryptographic hash functions?

- One way functions (efficient to compute and infeasible to invert)
- Infeasible to find a collision.
- Should exhibit avalanche effect (a small change in $m$ should give a uncorrelated hash value) – making it kind of uniquely linked to the message with no pattern whatsoever.
- The idea of collision resistant hash functions was formally defined by Damgård.

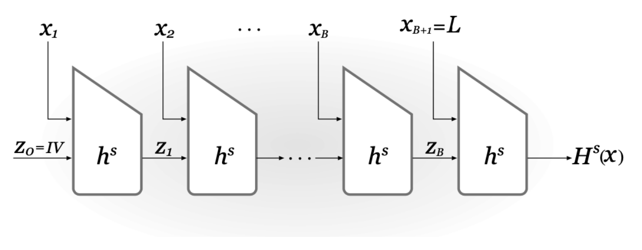
# Weaker Notions of Security

- Second-preimage resistance: If given $s$ and a uniform $x$ it is infeasible for a PPT $\mathcal{A}$ to find $x' \neq x$ s.t. $H^s(x') = H^s(x)$.
- Preimage resistance: If given $s$ and a uniform $y$ it is infeasible for a PPT $\mathcal{A}$ to find $x$ s.t. $H^s(x) = y$.
- Note this just means $H^s$ is one-way.
- Any hash function that is collision resistant is also second preimage resistant.
- Any hash function that is second preimage resistant is also preimage resistant. Well actually this one needs domain to be infinite! It needs that $H$ compresses and multiple inputs map to the same output with high probability.
- Collision resistance does not imply preimage resistance. In practice, a hash function that is only second pre-image resistant is considered insecure!

# Merkle-Damgård Transform

- We need domain extension to handle arbitrary length inputs.
- This transform extends the compression function while maintaining the collision-resistance property.
- Useful because now we need to restrict our attention only to the fixed-length case.
- Also this implies compressing by a single bit is as easy (or as hard) as compressing by an arbitrary amount.
- Introduced independently by Merkle and Damgård.

# Merkle-Damgård Transform




- $(Gen, H(x,s))$, $x \in \{0,1\}^*$ $|x| = L < 2^n$ from $(Gen, h)$ the hash function with fixed output length $n$.
- $B := \lceil \frac{L}{n} \rceil$ (Pad with 0s if needed), $x_{B+1} := L$
- $z_0 = 0^n$ IV, can be any constant
- For $i = 1, \ldots, B+1$, compute $z_i := h^s(z_{i-1} \circ x_i)$
- Output $z_{B+1}$

# Security Proof

If $(Gen, h)$ is collision resistant, then so is $(Gen, H)$.

- Idea: a collision in $H^s$ yields a collision in $h^s$.
- Let $x$ and $x'$ be two inputs $|x| = L$ and $|x'| = L'$ s.t. $H^s(x) = H^s(x')$.
- $x = x_1, \ldots, x_B$ and $x' = x'_1, \ldots, x'_{B'}$. Recall, $x_{B+1} = L$ and $X'_{B+1} = L'$.
- Case 1: $L \neq L'$
  - Last step: $z_{B+1} := h^s(z_B \circ L)$ and $z'_{B+1} := h^s(z'_B \circ L')$
  - $H^s(x) = H^s(x') \Rightarrow h^s(z_B \circ L) = h^s(z'_B \circ L')$.
  - Since $L \neq L'$ we have a collision for $h^s$.

# Security Proof

- Case 1: $L = L' \Rightarrow B = B'$.
- For $H^s(x)$ let the outputs be $z_0, \ldots, z_{B+1}$ and for $H^s(x')$ let the outputs be $z'_0, \ldots, z'_{B+1}$.
- Let $I_i = z_{i-1} \circ x_i$, $i$th input of $h^s$, set $I_{B+2} = z_{B+1}$ Similarly $I'$ for $x'$.
- Let $N$ be the largest index s.t. $I_n \neq I'_N$. (Clearly $N$ exists!)
- We have,

$$I_{B+2} = z_{B+1} = H^s(x) = H^s(x') = z'_{B+1} = I'_{B+2}, \Rightarrow N \leq B+1.$$

- We have $I'_{N+1} = I_{N+1}$ and $z_N = z'_N \Rightarrow$ a collision in $h^s$.

# Hash and MAC

- First hash-and-MAC and then HMAC.
- Simple mechanism: hash a message $m$ to $H^s(m)$ and then apply MAC to that, $MAC_k(H^s(m))$.
- $Verify_K(H^s(m), t)$ will return 1 if valid.
- This construction is secure if we have a secure MAC and $(Gen, H)$ is collision resistant. Since hash functions is collision resistant, then authenticating $H^s(m)$ is as good as authenticating $m$!

Formally,
If $\Pi$ is a secure MAC for messages of length $\ell$ and $\Pi_H$ is collision resistant, then the above method is a secure MAC for *arbitrary length* messages.

# Security Proof - Outline

- Say a sender uses the hash-and-MAC method to authenticate some set of messages $\mathcal{Q}$ and an $\mathcal{A}$ is able to forge a valid tag on a new message $m^* \notin \mathcal{Q}$.
- Then either of the two cases will happen:
  - there is a $m \in \mathcal{Q}$ s.t. $H^s(m^*) = H^s(m)$. But that means $\mathcal{A}$ has found a collision in $H^s$, a contradiction
  - for every $m \in \mathcal{Q}$ s.t. $H^s(m^*) \neq H^s(m)$. Then $\mathcal{A}$ has forged a valid tag on a new message w.r.t. the *fixed length* MAC $\Pi$, a contradiction to the secure MAC.
- We omit the formal proof.

# HMAC - Bellare et al.

- So far all MACs have used PRFs.
- Can we use hash functions? What about $MAC_k(m) = H(k \circ m)$?
- If $H$ is collision-resistant then $\mathcal{A}$ will find it difficult to predict the value of $H(k \circ m')$ given $H(k \circ m)$ for $m' \neq m$.
- But if $H$ is constructed using a Merkle-Damgård transform then it is completely insecure – Practice q (Q 5.10).
- We need two layers of hashing.

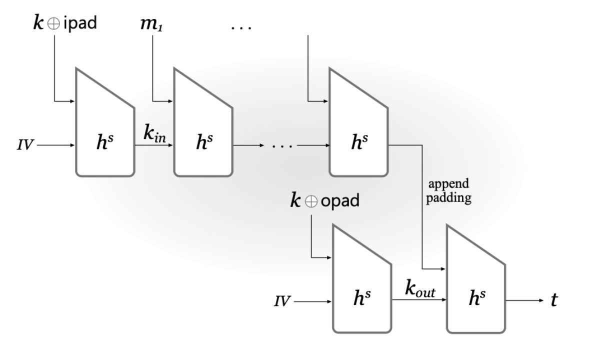
# HMAC

- $(Gen, H)$, a hash function obtained by Merkle-Damgård transform to $(Gen, h)$ which takes inputs of length $n + n'$.
- Let opad and ipad be fixed constant of length $n'$.
- Define a MAC as follows:
  - Gen$(1^n)$: Obtain a key $s \in \{0,1\}^n$ and also choose a uniform (secret) $k \in \{0,1\}^{n'}$. Output $\langle s, k \rangle$.
  - *MAC*: On input $m$,

$$t := H^s((k \oplus \mathrm{opad}) \circ H^s((k \oplus \mathrm{ipad}) \circ m)).$$

  - *Verify*: canonical verification.

# HMAC



# HMAC- a hash-and-MAC scheme

- We first hash an arbitrary-length message down to a short string $\overline{H}^s(m) := y = H^s((k \oplus \mathrm{ipad}) \circ m)$.
- Then compute the secretly keyed function $H^s((k \oplus \mathrm{opad}) \circ y)$.
- $\overline{H}^s(m)$ is collision-resistant for any value of $k \oplus \mathrm{ipad}$.
- Consider the outer computation: $H^s((k \oplus \mathrm{opad}) \circ y)$:
  - First step: $k_{out} := h^s(\underline{IV} \circ (k \oplus \mathrm{opad}))$, then $h^s(k_{out} \circ y)$.
  - This can be seen as $\overline{MAC}_k(y) := h^s(k \circ y)$. Actually the padded version of $y$.
  - I.e., if $\overline{MAC}$ is a secure fixed-length MAC, then

$$HMAC_{s,k}(m) = \overline{MAC}_{k=k_{out}}(\overline{H}^s(m)),$$

is a hash-and-MAC scheme.

# Why ipad and opad?

- This allows for HMAC to be built on a weaker assumption : $(Gen, H)$ is weakly collision resistant.
- Weakly collision resistant is the same as second preimage resistance.
  - You assume attacker has oracle access to $f(m) = H^s(k \circ m)$ ($k$ remains hidden)
  - Find distinct $m, m'$ s.t. $f(m) = f(m')$
- Weak collision resistance is also considered as a variant of strong collision resistance with just oracle access.
- Assume the $\overline{MAC}$ we defined above is a secure fixed-length MAC for messages of length $n$ and $(Gen, H)$ is a weakly collision resistant hash function. Then HMAC is a secure MAC.
- HMACs are widely used in practice.

# Attacks possible!

- Generic attacks for any hash function: Birthday attacks!
- Related to the birthday problem : if $q$ people are in a room, what is the probability that two people have the same birthday?
- About 23 peole give rise to a probability of same birthday greater than 1/2.
- Actual result is: for $y_1, \ldots, y_q$ chosen uniformly in $\{1, \ldots, N\}$, the probability of collision is roughly 1/2 when $q = \Theta(N^{1/2})$.
- In our setting if hash output is of length $\ell$, then taking $q = \Theta(2^{\ell/2})$ distinct inputs yields a collision with probability 1/2.
- Effective attacks possible :small-space birthday attack relying on Floyd's cycle-finding algo (check out wiki!)

# Motivation for Random-Oracle Model

- There are several examples of constructions based on cryptographic hash functions that cannot be proven secure based only on the assumption that the hash function is collision or preimage resistant.
- What do we do then?
  - Look for schemes that can be proven secure based on reasonable assumption about the hash function.
  - What do we do until such (efficient) schemes are found?
  - Just use schemes because no one has managed an attack on them.
  - That goes against this rigorous, modern approach to cryptography.
  - A middle ground approach – introduce an idealized model to prove security of schemes.
  - This may not be accurate reflection of reality, but you get some confidence in the design of the scheme.

# Random-Oracle Model - (Bellare and Rogaway)

- Assume a cryptographic hash function $H$ as a truly random function.
- It assumes the existence of a public, random function $H$ that can be evaluated only by querying an oracle/black box that returns $H(x)$ when given input $x$.
- Normal way of looking at things – standard model.
- ROs may not exist, although there have been suggestions that a RO can be implemented using a trusted setup.

# Random-Oracle Model

- It is a formal methodology to validate crypto schemes:
  - First prove a scheme is secure in RO model.
  - In real world, instantiate the RO by a crypto hash function $\overline{H}$.
- The hope is crypto hash functions are sufficiently good at emulating ROs. But no mathematical/heuristic proof!
- But there are (contrived) schemes that are proven secure in RO model but insecure in whatever way you instantiate the RO. (Canetti et al.)

# RO Model -Properties

- It is consistent. For the same input $x$ the output is always the same.
- If $x$ has not been queried to $H$, then the value of $H(x)$ is uniformly random.
- Proofs by reduction in the RO-model:
  - Extractibility : If $\mathcal{A}$ queries $x$ to $H$, the reduction can see this query and learn $x$.
  - Does not contradict that the queries to RO are private since $\mathcal{A}$ is a subroutine within the reduction which is simulating the RO for $\mathcal{A}$.
  - Programmability: The reduction can set the value of $H(x)$ as long as this value is uniform.
- The above two properties have no counterpart when you instantiate with a concrete function.

# RO Model - Sound?

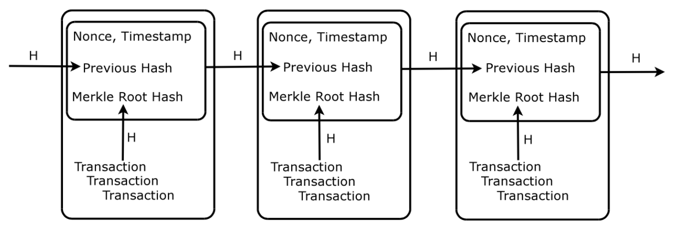
- For more details read the textbook.
- There are many objections for the RO-model. Does it help in the real-world scenario?
- But then a proof of security in the RO model is significantly better than no proof at all.
- A proof of security in the RO-model indicates a sound design. Many may disagree!

# Practical Applications of Hash Functions

- There are many! It is a short identifier for a file/message.
- It can speed up searching, used to store passwords (password hashes instead of plaintext passwords).
- Signature schemes – we will see this later.
- Blockchains - we have student presentations on this!
- Main idea: Identifier of a sequence $x_1, \ldots, x_t$ of messages.
- Can we do $H(x_1 \circ x_2 \circ \ldots \circ x_t)$? Not a very efficient *Verify*!

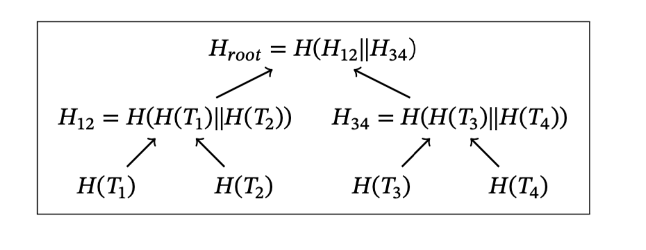
# Blockchains

- Merkle trees are more efficient at handling a large number of blocks.
- Blockchain is a sequence of linked blocks, a distributed ledger, which records transactions in an efficient and verifiable way.
- Each block contains the hash value of the previous block.
- Transactions in a block cannot be modified without changing all subsequent hash values.



# Merkle Trees

- Hashes of $T_1, T_2, T_3, \ldots$ form the leaves of the Merkle Tree, a binary tree.
- The nodes further up are hashes of two children nodes and the root of the Merkle is the top hash value.




- The root forms an identifier for *all transactions on a block.* Individual transactions can be verified by their hash path from leaf to root.

# An example

- We want to prove transaction $T_3'$ is included in the blockchain.
- Then we only need to provide the hashes $H_4 = H(T_4)$ and $H_{12}$ along with $T_3'$.
- Verfier checks the hash path by computing:
  - $H(T_3')$,
  - $H_{34}' = H(H(T_3') \circ H_4)$
  - $H_{root}' = H(H_{12} \circ H_{34}')$.
  - Verify $H_{root}' = H_{root}$
- Merkle trees are very efficient even when there are thousands of leaves and have applications beyond blockchains.

# Blockchains

- Used in cryptocurrencies and whenever you need a decentralized ledger.
- In cryptocurrencies, blockchain records the transactions of previously unspent cybercoins from one or more input addresses to one or more output addresses.
- Each new block contains a proof-of-work: by adapting the nonce value, a miner has to find a hash value of the new block that is smaller than the network's difficulty target.
- This may require a lot of hashing operations and consumes significant energy! The miner in turn is rewarded with new coins.
- Proof-of-work prevents the blockchains from manipulations and forks.

# Practical Constructions of Hash Functions

- MD5:
  - 128 bit output, designed in 1991.
  - Completely broken in 2004 in less than a minute on a desktop PC.
- Secure Hash Algorithm (SHA) Family
  - NIST standardized, SHA-1 and SHA-2.
  - First a fixed length compression function from a block cipher (Special block ciphers are used!)
  - Then, the Merkle-Damgård transformation is applied.
  - SHA-2 family contains SHA-224,-256,-384,-512.
- SHA-3 (Keccak)
  - Winner of the NIST competition.
  - Very different construction.
  - Stage I: unkeyed permutation of block length 1600 bits.
  - Stage II: Sponge construction. Nevers reveal the full state and prevents length extension attacks.