50.042 FCS Summer 2024 Lecture 5 – Applications for Hashing

Felix LOH
Singapore University of Technology and Design

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

With selected materials adapted from: *Understanding Cryptography: A Textbook for Students and Practitioners, by C. Paar and J. Pelzl*

Applications for cryptographic hash functions

Mathematically impossible to find inverse.

- Hash functions implement a one-way, *non-invertible*, function on data
  - Contrast these with encryption/decryption functions, which must be invertible (or bijective) functions
  - Ideally, the hash output (i.e. hash value) does not reveal any information about the input
- Hash functions can be used for:
  - Commitment schemes
  - Protection of message integrity and authentication
  - Error detection
  - Storage of secrets (with some caveats)

A commitment scheme

- Motivating problem: Suppose we have several individuals bidding on an item in a <u>sealed</u> bid scheme:
  - Each person can submit one sealed bid
  - Bids are compared after everybody has submitted their bids
  - The person with the highest bid gets the item
- How can we collect the bids and determine the highest bid securely (without using shared keys)?
- Alternative problem: Alice and Bob play a game of rock, paper, scissors over a remote connection
  - Need to seal each player’s choice securely, then only reveal the two choices simultaneously after both players have submitted their choice

A commitment scheme

- Both problems can be solved using a cryptographic hash scheme $z = h(x)$
- Use a two-phase protocol:
  1. Alice and Bob choose their respective actions e.g. x_a = “scissors” and x_b = “rock”, then they compute the corresponding hashes $z_a = h(x_a)$ and $z_b = h(x_b)$ and exchange their commitments z_a and z_b
  2. Alice and Bob then exchange their actual messages x_a and x_b, and they each verify that the message they received is authentic by recomputing the hash $h(x_a)$ or $h(x_b)$ of the received message and comparing it with the corresponding commitment hash z_a or z_b that they received in the 1st phase

An issue with the commitment scheme

- There is a problem with the commitment scheme as described in the previous slide:
  - During the 1st phase of the scheme (before the phase is completed), let's assume that Alice decides on some action and sends her commitment hash z_a over to Bob
  - Bob can then reverse-engineer Alice's commitment hash to identify the action that Alice chose
  - He then chooses the optimal action, then sends Alice his commitment hash z_b

An issue with the commitment scheme

- This reverse-engineering is possible because ‘rock’, ‘paper’ and ‘scissors’ always hash to the same values in the hash function

- There is only a small set of possible actions (i.e. three values), so it’s easy for Bob to just precompute all three possible hash values to determine Alice’s chosen action **before** she sends her message in the 2^{nd} phase

- We need to make the input less predictable (i.e. increase the input space)

Fixing the issue with the commitment scheme

- We can use *salting* to increase the input space of the hash function

- This basically entails adding some additional bits of data (i.e. a salt) to the original message *before* running the hash function

- The salt value can be different, even if the original message is the same

Fixing the issue with the commitment scheme

- Modify the two-phase protocol as follows:
    1. Alice and Bob choose their respective actions along with some random salt value s, then they compute the corresponding hashes $z_a = h(x_a, s_a)$ and $z_b = h(x_b, s_b)$ and exchange their commitment hash z_a and z_b
    2. Alice and Bob then exchange their message-salt pairs x_a, s_a and x_b, s_b and they each verify that the message they received is authentic by recomputing their respective hash $h(x_a, s_a)$ or $h(x_b, s_b)$ and comparing it with the respective commitment hash z_a or z_b that they received in the 1st phase

Padding for hash functions

- Recall that from the last lecture that we carry out padding for the last block of the message, before passing the message blocks to the hash function
- E.g. for SHA-1, the padding method is a '1' followed by a bunch of zeroes, followed by the 64-bit binary representation of the length of the message in bits before padding
- Other hash functions and ciphers use different padding methods
- Note that we **cannot** just simply pad the last block with random bits, because then it is no longer possible to verify the hash output (it is no longer deterministic)

Message authentication codes (MACs)

- Recall from last week’s lectures that encryption (or ciphers) **cannot** guarantee message integrity
  - Recall the “buy100” example
  - Eve can still modify the message despite not knowing the secret key

- Hash functions can guarantee integrity to a certain extent

Message authentication codes (MACs)

- But note that merely using a hash function, like SHA-1, cannot completely guarantee integrity
  - Eve/Oscar can still modify the message via the length extension attack

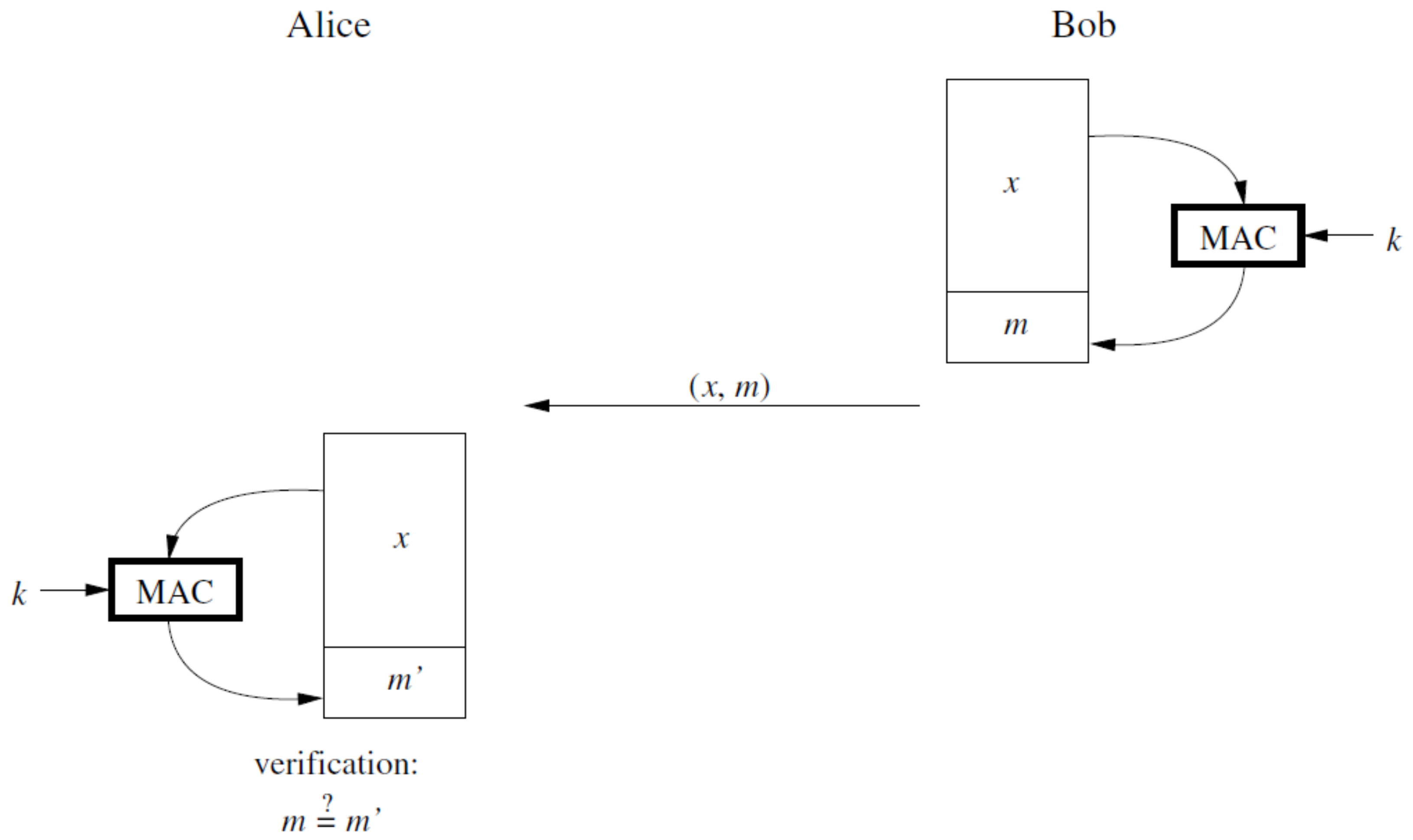
- An even worse situation: Eve/Oscar can perform a *man-in-the-middle* attack, by intercepting Alice's transmission of her plaintext-hash pair to Bob, then substituting that plaintext-hash pair with any arbitrary plaintext-hash pair of Eve/Oscar's choice – Bob has no way of knowing that the plaintext-hash pair did not come from Alice

Message authentication codes (MACs)

- Using a *shared secret key*, combined with a hash function, can provide a better guarantee of message integrity
  - This is the basic idea of a MAC
  - MACs can also be derived from block ciphers
  - MACs offer better protection against man-in-the-middle attacks, since only Alice and Bob possess the key, and thus are the only two parties capable of computing the MAC

Message authentication codes (MACs)

- Basic principle of MACs:



Message authentication codes (MACs)

- Motivation for using MACs: Alice and Bob need assurance that any modifications of a plaintext message during transit are detected
- Suppose Bob wants to send Alice a plaintext message *x*. They both share a secret key *k*
- He computes the authentication tag *m* as a function of the message *x* and the shared secret key *k*:

  $m = MAC_k(x)$

- He then sends both the message *x* and the authentication tag *m*
- When Alice receives *x* and *m*, she recomputes the authentication tag using the received message *x* and her key *k* (call this tag *m'*)

Message authentication codes (MACs)

- She can then check that $m' = m$. If this is the case, she can be confident that the message x was unaltered during transit, since her MAC computation m' would yield a different result if x was altered

- *Message integrity* is provided as a security service by MACs, just like hash functions

Message authentication codes (MACs)

- MACs also provide *message authentication*, in addition to message integrity
  - Alice can be assured that Bob is the source of a message and vice versa
  - This is possible because only Alice and Bob possess the secret key *k* and therefore are the only ones capable of computing the tag
  - Eve and Oscar are unable to compute the tag as they do not have the key

- However, MACs **cannot** provide *non-repudiation*
  - This is because the secret key *k* is shared by Alice and Bob, and there is no way to prove to a neutral third party that a message and its authentication tag (i.e. MAC) originated from either Alice or Bob

MACs from hash functions (HMAC)

- As mentioned earlier, we can use cryptographic hash functions (like SHA-1), together with a secret key, to realize MACs
- One possible construction, HMAC, is popular and commonly used
- HMAC is used in the TLS protocol
- HMAC is secure, provided that certain assumptions are made

- Two basic ways to construct HMAC:

  $m = MAC_k(x) = h(k \,||\, x)$ → secret prefix MAC

  $m = MAC_k(x) = h(x \,||\, k)$ → secret suffix MAC
- "||" is the concatenation operator

Attacks against HMACs

- Intuitively, both ways should result in strong cryptographic MACs, since modern cryptographic hash functions have the preimage resistance (one-wayness) property and good scrambling operations
- However, both approaches have weaknesses; let's discuss possible attacks against the secret prefix MAC and secret suffix MAC
- For these attacks, assume that the hash function is based on an MD construction, i.e. the intermediate hash value (output) of the *i*th iteration is $h_i = f(h_{i-1}, x_i)$, where f is some compression function

Attacks against secret prefix MACs

- $m = h(k \mid\mid x)$
- Suppose Bob wishes to send a message $x = (x_1, \ldots, x_L)$ to Alice
- He computes an authentication tag m using his secret key k, with

  $m = MAC_k(x) = h(k \mid\mid x_1, \ldots, x_L) = h_L = f(h_{L-1}, x_L)$
- He then sends x and m over to Alice

length ext attack for secret prefix MACs.

$m = MAC_k(x) = h(x \mid\mid k)$

Assume $h_i = f(h_{i-1}, x_2)$ ← MD construction.

Bob: $x = (x_1, x_2 \ldots x_L)$

$m = MAC(k \mid\mid x_1, x_2, \ldots x_L)$

Oscar: intercept Bob's message x & m.

⇒ note: oscar <u>cannot</u> send any arbitrary x' & m'

but he can <u>construct</u> $x_{oscar} = (x_1, x_2 \ldots x_L, x_{L+1})$ ← extra block for length ext attack.

construct $m_{oscar} = h(k \mid\mid x_1, x_2, \ldots x_L, x_{L+1})$

$= f(h_L, x_{L+1})$

$= f(m, x_{L+1})$ ← no 'k' involved here as m was provided by Bob.

stolen from Bob.

then sends x_{oscar}, m_{oscar} to Alice

Alice: recomputes m_{oscar}' using x_{oscar}

⇒ valid tag as $m_{oscar}' == m'$

⇒ thinks x_{oscar} is authentic

Attacks against secret prefix MACs

- However, Oscar intercepts the transmission containing x and m. He constructs an altered message $x_{Os} = (x_1, \ldots, x_L, x_{L+1})$ by adding an additional block x_{L+1} to the original message
- He then computes an authentication tag m_{Os}, with

  $m_{Os} = h(k \,||\, x_1, \ldots, x_L, x_{L+1}) = h_{L+1} = f(h_L, x_{L+1}) = f(m, x_{L+1})$

- In other words, $m_{Os} = f(m, x_{L+1}) = h(m \,||\, x_{L+1})$, which implies that Oscar does **not** need to know the secret key k to compute m_{Os}

Attacks against secret prefix MACs

- Oscar sends x_{Os} and m_{Os} over to Alice, who computes $m' = MAC_k(x_{Os}) = h(k \,||\, x_{Os}) = h(k \,||\, x_1, \ldots, x_L, x_{L+1}) = m_{Os}$
- Alice accepts the message x_{Os} as authentic, since $m' = m_{Os}$
  - Note that $m \neq m_{Os}$, but this does not matter as m_{Os} itself is a **valid** tag

Attacks against secret suffix MACs

- $m = h(x \,||\, k)$
- Again, suppose Bob wishes to send a message x to Alice
- He computes an authentication tag m using his secret key k, with

  $m = MAC_k(x) = h(x \,||\, k)$
- He then sends x and m over to Alice

Bob: x

$m = MAC_k(x) = h(x \,||\, k)$

Oscar: intercepts x, m

harder than finding collision.

needs to find a <u>2nd preimage</u> x_{oscar}

such that $h(x_{oscar}) = h(x)$

$\Rightarrow$ challenging, but if he is successful,

he can send x_{oscar}, m to Alice

Attacks against secret suffix MACs

- Again, Oscar intercepts the transmission containing x and m
- This time, Oscar tries to create a second message x_{Os} such that $h(x_{Os}) = h(x)$, with $x_{Os} \neq x$
- This is basically finding a second preimage of the hash function
- If he can find such a message x_{Os}, then $m = h(x || k) = h(x_{Os} || k)$, because of the iterative nature of the hash function
- So m would also be a valid tag for x_{Os}
- Then Oscar can send x_{Os} and m over to Alice

Attacks against HMACs: comments

- Attacks against secret prefix MACs are basically length extension attacks
- Attacks against secret suffix MACs require finding a second preimage of the hash function, which is much harder to achieve, as cryptographic hash functions are designed to have second preimage resistance
- So secret suffix MACs are better than secret prefix MACs

Storage of secrets

- We can also use hashes for storage of secret data
- Storage of usernames and the corresponding passwords on a computer:
  - It’s not a good idea to store the passwords in plaintext
  - To protect the passwords, the passwords are hashed using a cryptographic hash function
  - E.g. The username-passward pair “**`alice p@ssw0rd`**” may be stored as “**`alice 0x234a123d456efeed`**” on the computer, after the password is passed through a cryptographic hash function
  - When the user Alice inputs her password on the computer, the entered password is hashed and the resulting hash is then compared against the stored hash value

Storage of secrets

- On Linux distributions, the passwords are stored as SHA-512 hashes in the directory `/etc/shadow`
- Hashing the passwords before storage helps to maintain the secrecy of the passwords
- But there are ways to attack this storage scheme
  - The strategies of these attacks mainly revolve around finding the preimages of these hashes

Finding the preimages of hashes

- Suppose Oscar manages to steal a list of hashed password values
- Is there a way for him to derive the original plaintext passwords?

- Yes, Oscar can create a list of commonly used passwords in plaintext, then compute the corresponding hash for each password
- He can then compare each computed hash with the hashes in the stolen list; matching hashes indicate the corresponding plaintext password
- To save time, he can use a precomputed dictionary of password hashes and their corresponding plaintext passwords, or rainbow tables
  - E.g. password cracking programs like Hashcat and John the Ripper can utilize precomputed dictionaries and/or rainbow tables

Finding the preimages of hashes: mitigation

- One way in which we can mitigate such attacks on password hashes is to salt the plaintext password before hashing it
- Add some additional bits of data to the plaintext password
  - These bits of data are usually derived from the corresponding username of that password
  - Alternatively, the bits may be derived from a pseudo-random number generator
- Makes dictionaries and rainbow tables significantly less effective, because the same password for different usernames will map to different hash values
- Also increases the input space of the hash function

Other examples of attacks on hashes

- Yuval's square root attack
- MD5 Collisions Inc.