Hash Functions

Random Oracle Model (Andrés) and Some Applications (Kyle)

Recitation 4

Random Oracle Model (ROM)

Ideal Hash Function

- A *hash function* should satisfy main two properties: **one-wayness** and **collision resistance**.
- In many applications, we also want the hash function to “look random”.
- Basic properties of a hash function =!=> random function!
- What do we want from an “ideal” hash function?
    - We want it to behave like a random function. That is, a function where f(x) is a truly random string, for every x, independent of all other inputs.

“Random” function

- A random function maps every input to a new random string. If F is some random function, its table may look like:

Input	**Output (d bits)**
0	(A totally random d-bit string).
01	(Another totally random d-bit string).
00	(A third totally random d-bit string).
01	(Yet another totally random d-bit string).

- For every input, we sample a fresh random string of d bits .
- Important note: every random string is *independent* of all the other ones.
- Problem: **no** hash function (that's efficiently computable) can be a truly random function.

ROM

- Summary so far: we would like hash functions to behave like truly random functions, but no practical hash function will ever be a truly random function.
- Solution: we assume we have access to a *random oracle*: theoretical/abstract (public) “black-box” that implements a truly random function:
  - For every query x, check if x has been queried before. If yes, be consistent with prior answer. If no, sample a new d-bit random string.

ROM

x

H(x)

ROM

- The inner workings of the oracle H (the gnome) are unknown and magical. It just somehow implements this random function f.
- **ROM**: (theoretical) “world” where random oracles exists (i.e., a hypothetical world where perfect hash functions exist).
  - The ROM is a tool that we use in proofs.
  - We normally call the non-ROM world the *standard model*.

ROM

- First, we prove a protocol/scheme/etc is secure in the random oracle model.
- Then, when we implement this protocol in the real world, we replace the random oracle for a real hash function (e.g., SHA-256).
    - And we hope that this is good enough! I.e., that the behavior of a (good) hash function is indistinguishable from a truly random oracle.

Problems with ROM

- The random oracle model does not represent reality! A random oracle doesn’t (and will never) exist.
- What does it mean for a hash function to emulate a random oracle model? This is not even well defined…
  - Note: this is different from saying “we assume AES is a PRF”. We do have a definition of what it means to behave like a PRF.
- What does a proof in the ROM say about a proof in the real world? We don’t really know…
- A lot of active research into these questions.

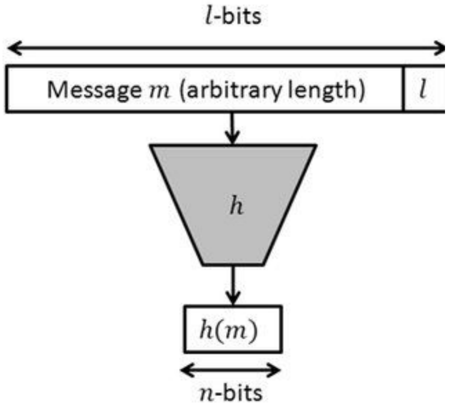
But…

- A ROM proof is still valuable: it shows the protocol has no “**intrinsic**” design flaws.
- There have been no attacks on implemented protocols that have been proven secure in the random oracle model!
    - However, there are some contrived examples of schemes that have been proven to be insecure for any instantiation of the random oracle!

Hash Functions in the Wild

6.857 Recitation 4

Commitments



Commitments

Cryptographic commitment:

- like a safe
  - Alice can put her message in the safe, lock it, and give the safe to Bob

Commitments

Cryptographic commitment:

- like a safe
  - Alice can put her message in the safe, lock it, and give the safe to Bob
  - Until Alice opens the safe, Bob learns nothing about Alice’s message

Commitments

Cryptographic commitment:

- like a safe
  - Alice can put her message in the safe, lock it, and give the safe to Bob
  - Until Alice opens the safe, Bob learns nothing about Alice’s message
  - Yet Alice cannot change her message after she places it in the safe

Commitments

Cryptographic commitment:

- like a safe
  - Alice can put her message in the safe, lock it, and give the safe to Bob
  - Until Alice opens the safe, Bob learns nothing about Alice’s message
  - Yet Alice cannot change her message after she places it in the safe

These properties are known as ***hiding*** and ***binding***

Commitments

Cryptographic commitment:

- like a safe
  - Alice can put her message in the safe, lock it, and give the safe to Bob
  - Until Alice opens the safe, Bob learns nothing about Alice’s message
  - Yet Alice cannot change her message after she places it in the safe

These properties are known as ***hiding*** and ***binding***

Hash functions are commonly used for commitments in pratice

One wayness provides hiding while collision resistance provides binding

Commitments using Hash functions

To commit to a message *m:*

1. Alice generates a random string *r* and computes $commit_m = H(r||m)$
2. She then sends $commit_m$ to Bob.

Commitments using Hash functions: Intermission

To commit to a message *m:*

1. Alice generates a random string *r* and computes $commit_m = H(r||m)$
2. She then sends $commit_m$ to Bob.

Why $r||m$ instead of just *m*?

Commitments using Hash functions: Intermission

To commit to a message *m:*

1. Alice generates a random string r and computes $commit_m = H(r||m)$
2. She then sends $commit_m$ to Bob.

Why $r||m$ instead of just m?

Hash functions are deterministic!

m could be chosen from a distribution that makes it easy for Bob to guess and check.

Commitments using Hash functions: Intermission

To commit to a message *m:*

1. Alice generates a random string *r* and computes $commit_m = H(r||m)$
2. She then sends $commit_m$ to Bob.

Why $r||m$ instead of just *m*?

Hash functions are deterministic!

m could be chosen from a distribution that makes it easy for Bob to guess and check.

$m \leftarrow \{\text{heads, tails}\}$

Bob only needs to try two strings! So include *r* to ensure the message space is uniformly random in the length of *r*

Commitments using Hash functions

To commit to a message *m:*

1. Alice generates a random string r and computes $commit_m = H(r||m)$
2. She then sends $commit_m$ to Bob.

Opening $commit_m$:

1. Alice sends m' and r to Bob
2. Bob computes $commit_{m'} = H(r||m')$
3. Bob checks that $commit_{m'} == commit_m$

Commitments using Hash functions

To commit to a message *m:*

1. Alice generates a random string r and computes $commit_m = H(r||m)$
2. She then sends $commit_m$ to Bob.

Opening $commit_m$:

1. Alice sends m' and r to Bob
2. Bob computes $commit_{m'} = H(r||m')$
3. Bob checks that $commit_{m'} == commit_m$

Binding:

Hiding:

Commitments using Hash functions

To commit to a message *m:*

1. Alice generates a random string r and computes $commit_m = H(r||m)$
2. She then sends $commit_m$ to Bob.

Opening $commit_m$:

1. Alice sends m' and r to Bob
2. Bob computes $commit_{m'} = H(r||m')$
3. Bob checks that $commit_{m'} == commit_m$

Binding: m != m' requires Alice to find a collision for H()!

Hiding: If Bob can learn m from H(r||m) then Bob can invert H()

Not covered:

Proofs for hiding and binding in ROM

Use in practice:

Zero knowledge!

Verifiable secret sharing!

both cool areas for a project :)

Passwords

Password based login: a password is a ‘something you know’ based login mechanism

In order to login to an account, Alice must prove that she knows the password associated with that account

Passwords

Password based login: a password is a ‘something you know’ based login mechanism

In order to login to an account, Alice must prove that she knows the password associated with that account

However, we **really really** do not want the server to store passwords!

Passwords

Password based login: a password is a ‘something you know’ based login mechanism

In order to login to an account, Alice must prove that she knows the password associated with that account

However, we **really really** do not want the server to store passwords!

➔ If it gets hacked, all the passwords are revealed :(

Passwords: defending against breaches

Instead of storing *pwd* directly, the server stores H(pwd)

Are we done?

Passwords: defending against breaches

Passwords are *definitely* not chosen from a random distribution

Hackers can compute H(pwd) for a bunch of popular passwords offline then easily compare against the server's database of hashed passwords!

Top 20 most common passwords according to NordPass[3]

Rank	2021
1	123456
2	123456789
3	12345
4	qwerty
5	password
6	12345678
7	111111
8	123123
9	1234567890
10	1234567
11	qwerty123
12	000000
13	1q2w3e
14	aa12345678
15	abc123
16	password1
17	1234
18	qwertyuiop
19	123321
20	password123

Passwords: defending against breaches

Passwords are *definitely* not chosen from a random distribution

Hackers can compute H(pwd) for a bunch of popular passwords offline then easily compare against the server's database of hashed passwords!

Worse, many people use the same passwords – adversary only needs to learn which pwd results in H(pwd) one time

Top 20 most common passwords according to NordPass[3]

Rank	2021
1	123456
2	123456789
3	12345
4	qwerty
5	password
6	12345678
7	111111
8	123123
9	1234567890
10	1234567
11	qwerty123
12	000000
13	1q2w3e
14	aa12345678
15	abc123
16	password1
17	1234
18	qwertyuiop
19	123321
20	password123

Passwords: defending against breaches

Passwords are *definitely* not chosen from a random distribution

Hackers can compute H(pwd) for a bunch of popular passwords offline then easily compare against the server's database of hashed passwords!

Worse, many people use the same passwords – adversary only needs to learn which pwd results in H(pwd) one time

Aside: it is slightly more complicated than this, but the technique (rainbow tables) is obsolete so we won't cover it

Top 20 most common passwords according to NordPass[3]

Rank	2021
1	123456
2	123456789
3	12345
4	qwerty
5	password
6	12345678
7	111111
8	123123
9	1234567890
10	1234567
11	qwerty123
12	000000
13	1q2w3e
14	aa12345678
15	abc123
16	password1
17	1234
18	qwertyuiop
19	123321
20	password123

Passwords: salt your passwords

We had this very same problem with commitments and solved it with *r*

For passwords, this is called a *salt* and the server stores [H(pwd||salt), salt]

Passwords: salt your passwords - Intermission

We had this very same problem with commitments and solved it with *r*

For passwords, this is called a *salt* and the server stores [H(pwd||salt), salt]

Why store *salt* on the server?

Passwords: salt your passwords - Intermission

We had this very same problem with commitments and solved it with *r*

For passwords, this is called a *salt* and the server stores [H(pwd||salt), salt]

Why store *salt* on the server?

The server is the one computing H(pwd||salt) and clients will forget the salt!

Passwords: salt your passwords - Intermission

We had this very same problem with commitments and solved it with *r*

For passwords, this is called a *salt* and the server stores [H(pwd||salt), salt]

Why store *salt* on the server?

The server is the one computing H(pwd||salt)

Why?

Passwords: salt your passwords - Intermission

We had this very same problem with commitments and solved it with *r*

For passwords, this is called a *salt* and the server stores [H(pwd||salt), salt]

Why store *salt* on the server?

The server is the one computing H(pwd||salt)

Why?

Clients must login with pwd **NOT** H(pwd)!!! Very important. If the client sends H(pwd) to login, then H(pwd) effectively *is* the password. Ruins all the effort to store hashes instead of passwords in case of breach :(

Passwords: salt your passwords

We had this very same problem with commitments and solved it with *r*

For passwords, this is called a *salt* and the server stores [H(pwd||salt), salt]

This prevents an adversary from precomputing hashes of popular passwords

1. Users with the same pwd will now have different salts
2. Adversary may have ideas of popular passwords, but salts are uniformly random so it can’t guess them in advance!

Passwords: H(H(H(H(H(pwd)))))

In practice the server actually hashes the password many times, not just once

Passwords: H(H(H(H(H(pwd)))))

In practice the server actually hashes the password many times, not just once

This is to make brute force attacks harder!

Passwords: H(H(H(H(H(pwd)))))

In practice the server actually hashes the password many times, not just once

This is to make brute force attacks harder!

Computing a bunch of hashes is time consuming and expensive (see: Bitcoin)

Passwords: H(H(H(H(H(pwd)))))

In practice the server actually hashes the password many times, not just once

This is to make brute force attacks harder!

Computing a bunch of hashes is time consuming and expensive (see: Bitcoin)

Say Eve breaches a password database and learns that account `aliceincryptoland` has password `[H(pwd||salt), salt]`

If the password is hashed ***once***, Eve only has to compute ***one hash*** to check each password she wants to guess

Passwords: H(H(H(H(H(pwd)))))

In practice the server actually hashes the password many times, not just once

This is to make brute force attacks harder!

Computing a bunch of hashes is time consuming and expensive (see: Bitcoin)

Say Eve breaches a password database and learns that account `aliceincryptoland` has password `[H(pwd||salt), salt]`

If the password is hashed ***n*** times, Eve has to compute ***n hashes*** to check each password she wants to guess

Passwords: H(H(H(H(H(pwd)))))

In practice the server actually hashes the password many times, not just once

This is to make brute force attacks harder!

Computing a bunch of hashes is time consuming and expensive (see: Bitcoin)

Say Eve breaches a password database and learns that account `aliceincryptoland` has password `[H(pwd||salt), salt]`

If the password is hashed ***n*** times, Eve has to compute ***n*** hashes to check each password she wants to guess

This doesn't make things noticeably slower for Alice, but will really ruin Eve's day on a DB of millions of passwords

Questions?