CS4236 Cryptography Theory and Practice Topic 8 - Hash applications and attacks

Hugh Anderson

National University of Singapore
School of Computing

October, 2022

Hash applications, rainbows and attacks...

Outline

1. **Applications of hash (continued)**
   - Fingerprinting, deduplication schemes
   - Commitment schemes

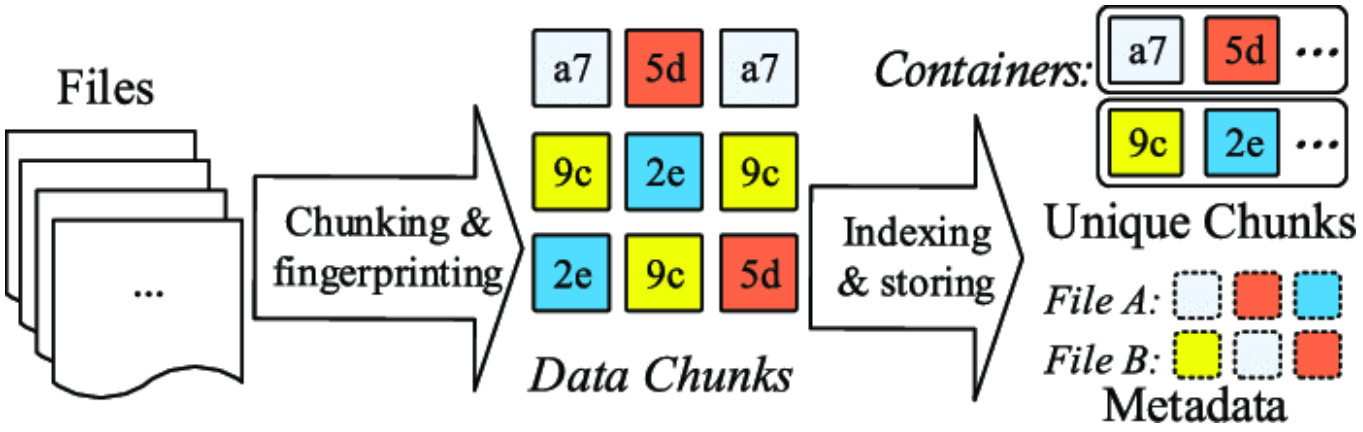
2. **Hash attacks**
   - Time memory tradeoff, to find preimages
   - Birthday attacks, to find collisions

The story so far ... where are we?

The last 7 weeks: weekly steps we have taken

1. We had a historical/contextual introduction in Session 1.
2. We progressed from the traditional view of perfect secrecy, through to a game/experiment view: *perfect* indistinguishability.
3. *Perfectly* indistinguishable was relaxed to give *computationally* indistinguishable. An EAV-Secure system was constructed with a PRG.
4. The notion of CPA-secure was developed, where the adversary had access to an encryption oracle. CPA-secure was achieved with a PRF.
5. Modes were described, and CCA-Security was outlined. The “padding oracle” example was described, motivating CCA.
6. We looked at the “right” tools for integrity: Cryptographic MACs and *authenticated* encryption (CCA-secure and unforgeable).
7. We began on hashes. Once again, (fixed-size) hash functions were defined, and a game-based experiment was used to define collision resistance. Some of the methods used to extend fixed-size to variable-size hash functions were shown. Some of the applications of hashes were outlined, including their use in password systems, proof-of-work in bitcoin/blockchain technology, and CGA.

The hash stands for the object



Hashes are everywhere (p182)

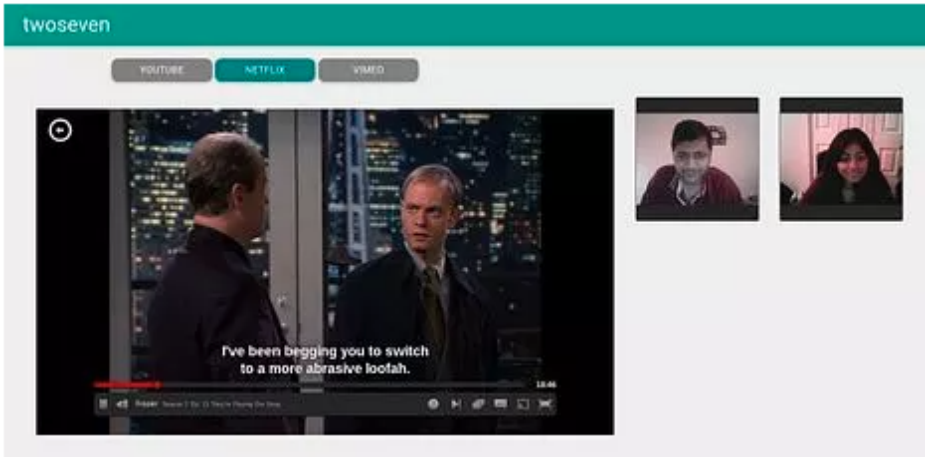
The hash of a file/object can serve as a unique identier for that file (otherwise there is a collision). Here are some examples of uses of this:

Fingerprinting: A virus scanner (for example) could have access to a database of hashes of previously identified malicious files. This is more efficient in speed of searches of the database.

Deduplication: To eliminate duplicate copies of data; perhaps a popular image or video stored by different facebook users. Deduplication allows only one copy to be stored, with links to it from the other users.

P2P shares: The hashes serve as unique identifiers for the files shared.

Motivating the commitment scheme



Movie night

Alice and Bob are talking over a secure channel. They want to decide which movie to watch. Alice prefers *movieA*, but Bob wants *movieB*.

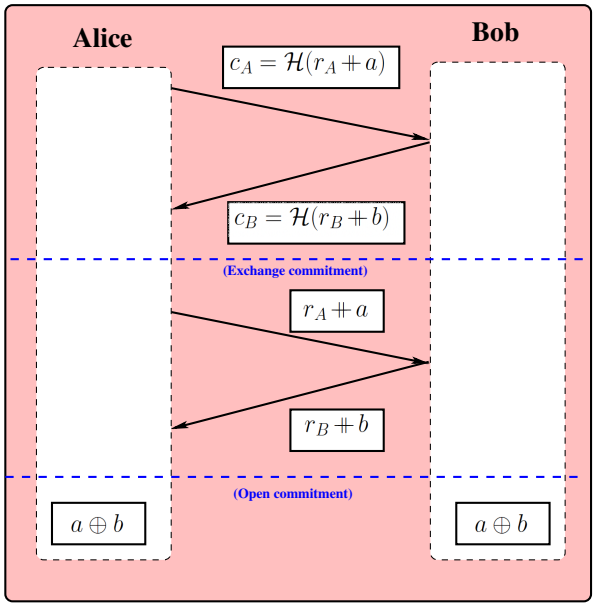
To resolve that, they derive this protocol: each of them is to think of a bit, say a and b. If $(a \oplus b) = 1$, then they will watch *movieA*, otherwise *movieB*.

Now, how to exchange these two bits a and b so that they are unable to cheat? If Alice sends Bob $a = 1$ first, Bob can force Alice to watch *MovieB*.

Tossing a coin?

Alice and Bob are not in the same room. It is problematic.

A commitment scheme



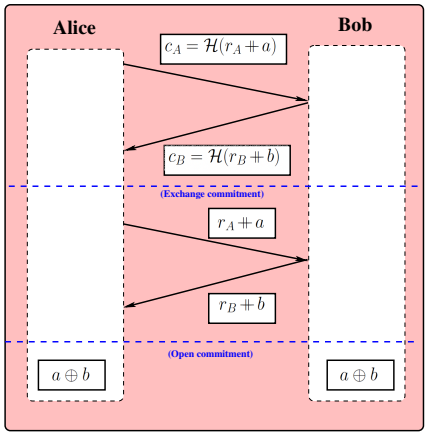
Using a collision-resistant hash

Alice and Bob randomly pick r_A, r_B, two 511-bit sequences, and *secret* bit choices a and b. They then perform the following steps:

1. Alice computes $c_A = \text{commit}(r_A, a) = \mathcal{H}(r_A \# a)$, and Bob computes $c_B = \text{commit}(r_B, b) = \mathcal{H}(r_B \# b)$
2. Alice and Bob exchange their commitment values c_A, c_B.
3. Once they have done that, they then exchange the $r_A \# a$ and $r_B \# b$ sequences. This is called "*opening the commitment*".

Alice and Bob verify c_B, c_A respectively, and the final random bit is $a \oplus b$.

A commitment scheme



But is it secure?

Suppose initially, Bob chooses $b = 0$. After Bob receives a from Alice, can he change his mind? If he can do so, this means that he can find another r such that $c_B = \mathcal{H}(r + 1)$.

That is, can he find an r such that $\mathcal{H}(r_B + 0) = \mathcal{H}(r + 1)$, a collision of $\mathcal{H}$. This contradicts the assumption that $\mathcal{H}$ is collision resistant.

For a formal proof, assume the collision-resistant game. From Definition 5.2 the probability of Bob finding a collision is $\Pr[\text{Hash-coll}_{\mathcal{A},\Pi}(n) = 1] \leq \text{negl}(n)$, and only roughly half of any such collisions have 0 as the least significant bit. So Bob is unlikely to be able to revise his choice.

Formalisation of the commitment scheme

A commitment scheme $\mathrm{Com} = (\mathrm{Setup}, \mathrm{Commit}, \mathrm{Open})$

With 3 entities: Alice, Bob and a trusted party T.

$\mathrm{Setup}(1^n)$: During the Setup phase, T helps Alice and Bob to establish the common parameters, perhaps parameters to a crypto primitive like the value of n or public keys.

$\mathrm{Commit}(a)$: During this phase, Alice commits her choice. This is like putting a into a safe, which only she can open.

$\mathrm{Open}(c_a)$: Alice opens the commitment to convince Bob that she does not change her mind.

Definition 5.13: Secure commitment scheme

Com is secure if for all PPT adversary $\mathcal{A}$, there exists a negl, s.t.

$$\begin{aligned} \Pr[\mathrm{Hiding}_{\mathcal{A},\mathrm{Com}}(n) = 1] &\leq \tfrac{1}{2} + \mathrm{negl}(n) \\ &\text{and} \\ \Pr[\mathrm{Binding}_{\mathcal{A},\mathrm{Com}}(n) = 1] &\leq \mathrm{negl}(n) \end{aligned}$$

The Binding experiment indicates the value of a can't be modified after commitment. Hiding indicates that no information of the value a is leaked.

Use of commitment schemes

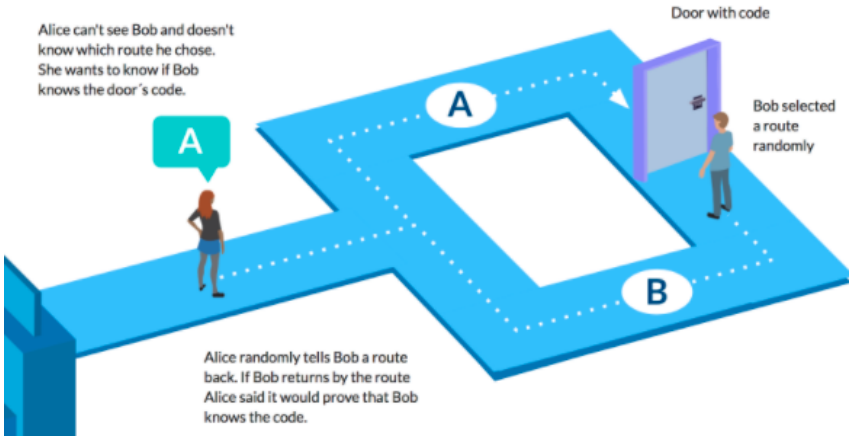
What can be done with commitment schemes?

A not-at-all exhaustive list of examples:

- Secure multi party computation. An example might be where Alice and Bob each have secrets a, b and a shared function $f(x, y)$. After secure multi-party computation, Alice knows $a, f(a, b)$ but not b. Bob knows $b, f(a, b)$ but not a.
- Playing games over the Internet. For example - consider an online Casino, how does the operator convince the players that the operator doesn't cheat? Perhaps a commitment scheme where the Casino opens their commitment only after all the bets are in.
- Online bidding/selling.

There are many such uses.

Proof of properties in committed values



Proof without revealing information - ZKP

In some applications, the verifier wants to see some “proof” that the committed values meet some properties. For example - at least one of the 2 committed values has the value “0”, the sum is less than a certain threshold, or the committed string is a valid NRIC number.
It is important that this “proof” does not leak more than what is required.

It is possible to achieve that efficiently for some types of properties. In general, it is also possible for any property (of course, in the first place, we must have a PPT algorithm that verifies the property from the plaintext) using “non-interactive zero-knowledge proof”.

An aside: tossing a coin another way

Alice and Bob want to toss a coin, so do this:

1. Alice finds two primes p, q, calculates $N = pq$, and then sends N to Bob. If Bob can factorize this number, then he wins. $N = 35 = 5 * 7$
2. Bob selects random $r = 31$, and sends $y = r^2 \bmod N$ to Alice. $y = 31^2 \bmod 35 = 16$
3. Alice knows the prime factors of N, calculates the square roots of 16:

$$\begin{aligned} 4^2 \bmod 35 &= 16 & \qquad 31^2 \bmod 35 &= 16 \\ 24^2 \bmod 35 &= 16 & \qquad 11^2 \bmod 35 &= 16 \end{aligned}$$

and then sends one (say 24) back to Bob.

If Bob receives $\pm r$ he learns nothing.

But when Bob receives another value (in this case 24), he does this:

$$\begin{aligned} \gcd(24 + 31, 35) &= \gcd(55, 35) \\ &= 5 \end{aligned}$$

Bob calculates a factor of N, and Alice is unable to tell she has divulged it.

Reversing a hash of a password - by brute force?

To try checking the hashes for all the 9 character passwords, you might have to try (say) $78^9 = 106,868,920,913,284,608$ hashes.

That would take a long time. At 1uS for trying each hash, this would take over 3000 years to calculate.

Hugh will look like this when he reverses your password:

Password	(MD5) Hash
aaaaaaaaa	`3dbe00a1676..`
aaaaaaaab	`2125ea8b81b..`
aaaaaaaac	`ea67f32d4e6..`
aaaaaaaad	`746a8ab05d6..`
aaaaaaaae	`c554d695eb0..`
aaaaaaaaf	`09eb61fd25b..`
aaaaaaaag	`68b5af18408..`
...	...

Reversing a hash of a password - by a dictionary?

A precomputed table for 9 character passwords, might have (say) $78^9 = 106,868,920,913,284,608$ entries, each containing a 16 byte value.

Thats a big disk (about 1,500,000 TB)! My 2TB disk cost me about $100/TB, so I would need $150,000,000 to buy this disk. Who has this sort of money? (... well ... actually ...)

Indexing by hash is even worse. We do not really have names for disks that big.

Password	(MD5) Hash
aaaaaaaaa	`3dbe00a1676..`
aaaaaaaab	`2125ea8b81b..`
aaaaaaaac	`ea67f32d4e6..`
aaaaaaaad	`746a8ab05d6..`
aaaaaaaae	`c554d695eb0..`
aaaaaaaaf	`09eb61fd25b..`
aaaaaaaag	`68b5af18408..`
...	...

More formally...

The same ideas expressed in the language of cs4236...

Let consider a hash $\mathcal{H}()$ which is preimage resistant (i.e. one-way). Given a digest y, it is difficult to find an x s.t. $\mathcal{H}(x) = y$.

Suppose we know that the message x is chosen from a relatively small set in dictionary D. For illustration, let's assume x is a randomly and uniformly chosen 48-bit message.

Even if $\mathcal{H}()$ is one-way, given the digest y, it is still feasible to find an x in D s.t. $\mathcal{H}(x) = y$. This can be done by exhaustively searching the 2^{48} messages in D. Although feasible, this would take a long time. As the attacker, we want to speedup the process to support a *realtime* attack.

We are allowed to perform pre-processing

Build a dataset with 2^{48} pairs $\langle \mathcal{H}(x), x \rangle$, and store these elements in a data structure that supports fast lookup such as a hash table.

Now, given a digest y, we can query the data structure and readily find the associated x. However, such a table is huge! If each entry was (say) $16 + 6 = 22$ bytes, then it would be $2^{48} \times 22 = 6192\text{TB}$.

Time-memory tradeoff: chains

Idea #1: define a reduce function $R()$

The reduce function $R(y)$ maps a digest y to a word w in the dictionary D.

For illustration, if D consists of all 48-bit messages, and each digest is (say) 320-bits, then a possible reduce function simply keeps the first 48-bits of the digest. This is then used to generate the next word in D. Let's look at this chain, starting from a randomly chosen word w_0 in D:

$$w_0 \xrightarrow{\mathcal{H}} y_0 \xrightarrow{R} w_1 \xrightarrow{\mathcal{H}} y_1 \xrightarrow{R} w_2 \xrightarrow{\mathcal{H}} y_2$$

w_0 is the starting-point, and y_2 the ending-point. The pair $\langle w_0, y_2 \rangle$ is stored in the data-structure T, with other chains from different starting points.

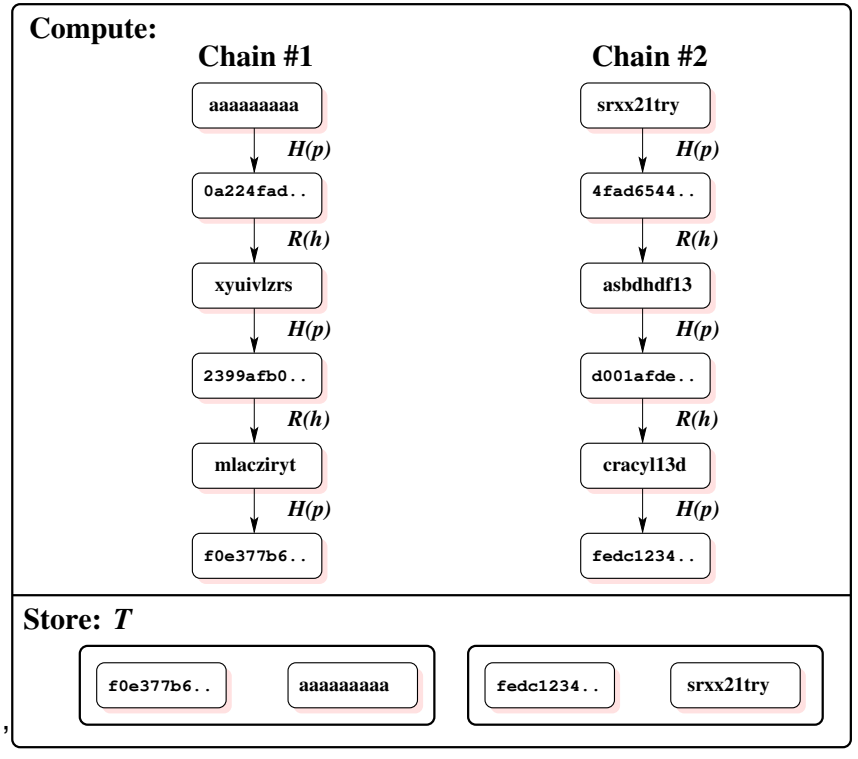
---

1. If y is in T (an ending-point; in the above $y = y_2$), then the corresponding starting point w_0 is traced to find the preimage of y_2. We discover w_2.
2. Otherwise, from y we recalculate a chain until we reach an endpoint. In the above if $y = y_1$, then the chain is re-computed until y_2, and then the corresponding starting point w_0 is traced to find the preimage of y_1.

Using these tables for password reversing...

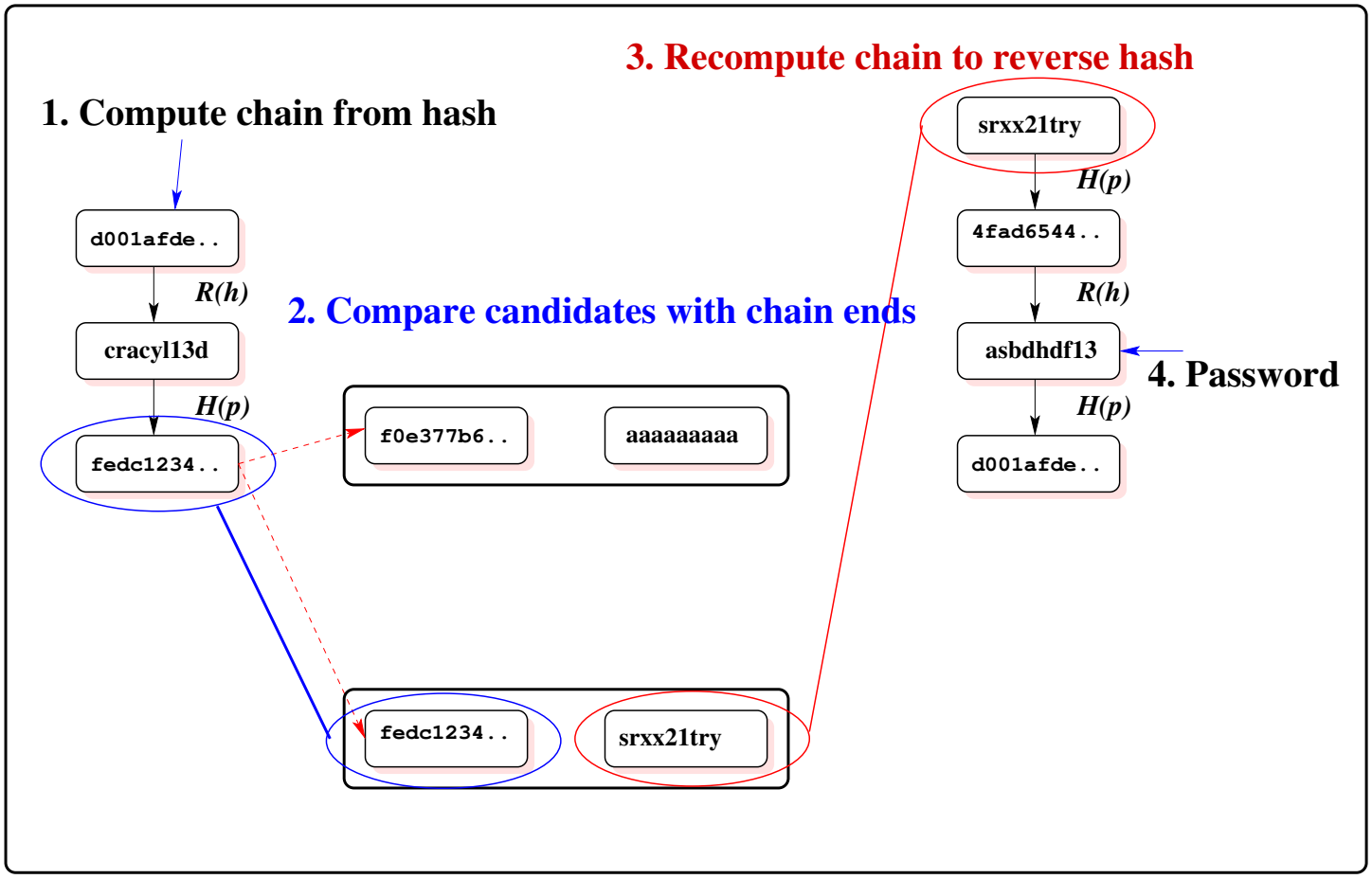
Precompute chains of values starting from a password guess, and using alternately, a hash function $\mathcal{H}(p)$, and a reducing function $R(h)$, which generates a predictable plausible guess from the hash.

Only store the first and last entries from the chain. It is space efficient, and you can recompute the intermediate values (a space-time tradeoff).



Reversing the hash

Compute, compare with "last" values, recompute...



Analysis of time and space required

(Compared with the full table)

Space: If a chain had n pairs in it, then we only need to store only 1 pair, the beginning and the end pair. if (for example) we had n such chains, we would reduce the space from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$.

Time/query: The number of hash and reduce operations now needed is now also $\mathcal{O}(n)$.

Accuracy: The chains contain repetitions. (why?)

In general, we can choose the length of the chain so that the reduction of space is a factor of k, with the penalty of increase the search time by a factor of k, where k is a parameter. In our 48-bits example, if we choose $k = 2^{16}$, and the number of entries in the table to 2^{32}, then the total number of entries covered in the virtual table is 2^{48} (as we shall see later, these 2^{48} entries are not unique).

Collisions due to the reduce function

$$w_0 \xrightarrow{\mathcal{H}} y_0 \xrightarrow{R} w_1 \xrightarrow{\mathcal{H}} y_1 \xrightarrow{R} w_2 \xrightarrow{\mathcal{H}} y_2 \xrightarrow{R} w_3 \xrightarrow{\mathcal{H}} y_3$$

$$w_9 \xrightarrow{\mathcal{H}} y_9 \xrightarrow{R} w_2 \xrightarrow{\mathcal{H}} y_2 \xrightarrow{R} w_3 \xrightarrow{\mathcal{H}} y_3 \xrightarrow{R} w_4 \xrightarrow{\mathcal{H}} y_4$$

Collisions of the reduce function may happen frequently

The pairs $\langle w_0, y_3 \rangle$ and $\langle w_9, y_4 \rangle$ will be stored. This causes two issues:

1. Efficiency in storage: Part of the chain is duplicated. The words w_2 and w_3 appear in the table twice.
2. Efficiency in search: Lead to searches in the wrong chains, before hitting the right chain. For the query y_9, the lookup process would transverse both chains.

Solution to the problem: the "Rainbow" table

Suppose there are k reduce operations and k hash operations in the chain, instead of using the same function $R()$ in all the reduce operations, the Rainbow table uses k different functions. The first reduce is $R_1()$, followed by $R_2()$ and so on.

If collisions happen

$$w_0 \xrightarrow{\mathcal{H}} y_0 \xrightarrow{R_1} w_1 \xrightarrow{\mathcal{H}} y_1 \xrightarrow{R_2} w_2 \xrightarrow{\mathcal{H}} y_2 \xrightarrow{R_3} w_3 \xrightarrow{\mathcal{H}} y_3$$

$$w_9 \xrightarrow{\mathcal{H}} y_9 \xrightarrow{R_1} w_2 \xrightarrow{\mathcal{H}} y_2 \xrightarrow{R_2} w_5 \xrightarrow{\mathcal{H}} y_5 \xrightarrow{R_3} w_6 \xrightarrow{\mathcal{H}} y_6$$

If w_2 appears twice as above

Fortunately, both $\langle w_0, y_3 \rangle$ and $\langle w_9, y_6 \rangle$ will be stored in the data-structure. Now, for the query y_9, its preimage w_9 can be found.

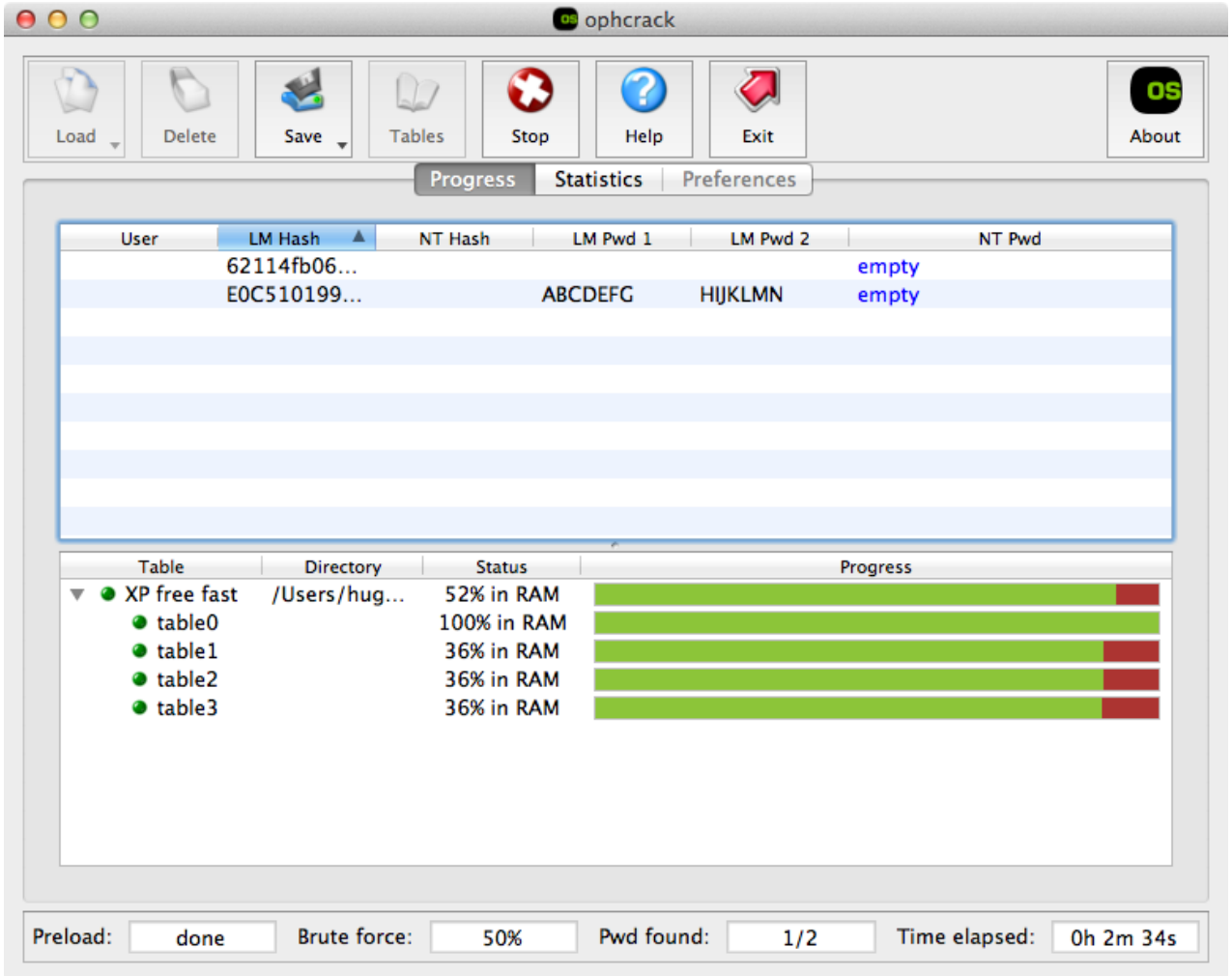
$$w_0 \xrightarrow{\mathcal{H}} y_0 \xrightarrow{R_1} w_1 \xrightarrow{\mathcal{H}} y_1 \xrightarrow{R_2} w_2 \xrightarrow{\mathcal{H}} y_2 \xrightarrow{R_3} w_3 \xrightarrow{\mathcal{H}} y_3$$

$$w_9 \xrightarrow{\mathcal{H}} y_9 \xrightarrow{R_1} w_1 \xrightarrow{\mathcal{H}} y_1 \xrightarrow{R_2} w_2 \xrightarrow{\mathcal{H}} y_2 \xrightarrow{R_3} w_3 \xrightarrow{\mathcal{H}} y_3$$

Or if collision was on the same R_x? There will be two y_3!

Both chains may have to be traversed. Rainbow tables lower the number of duplications, and increase the number of words that can be found.

Reversing an old Windows password hash



Not only hashes!

This technique can be applied to a known plaintext attack on an encryption scheme where the keyspace is small. (how?)

The Birthday "Paradox" explained

It is NOT the likelihood that someone in a room full of people shares a specific person's birthday...

It is the likelihood that amongst every pair of candidates in a room there will be (at least) one matching pair. It turns out that it is easiest to calculate the likelihood that the people will not share birthdays[a]:

- If $N = 2$, then the likelihood the two do not share a birthday is $\frac{364}{365}$, because the first person can have any of the 365 days, leaving 364 days available for the second person.
- If $N = 3$, then the likelihood all three do not share is $\frac{364}{365} \times \frac{363}{365}$.
- For N, the likelihood is $\frac{364}{365} \times \frac{363}{365} \times \ldots \times \frac{365-(N-1)}{365}$.

For $N = 23$, this likelihood is about 0.5.

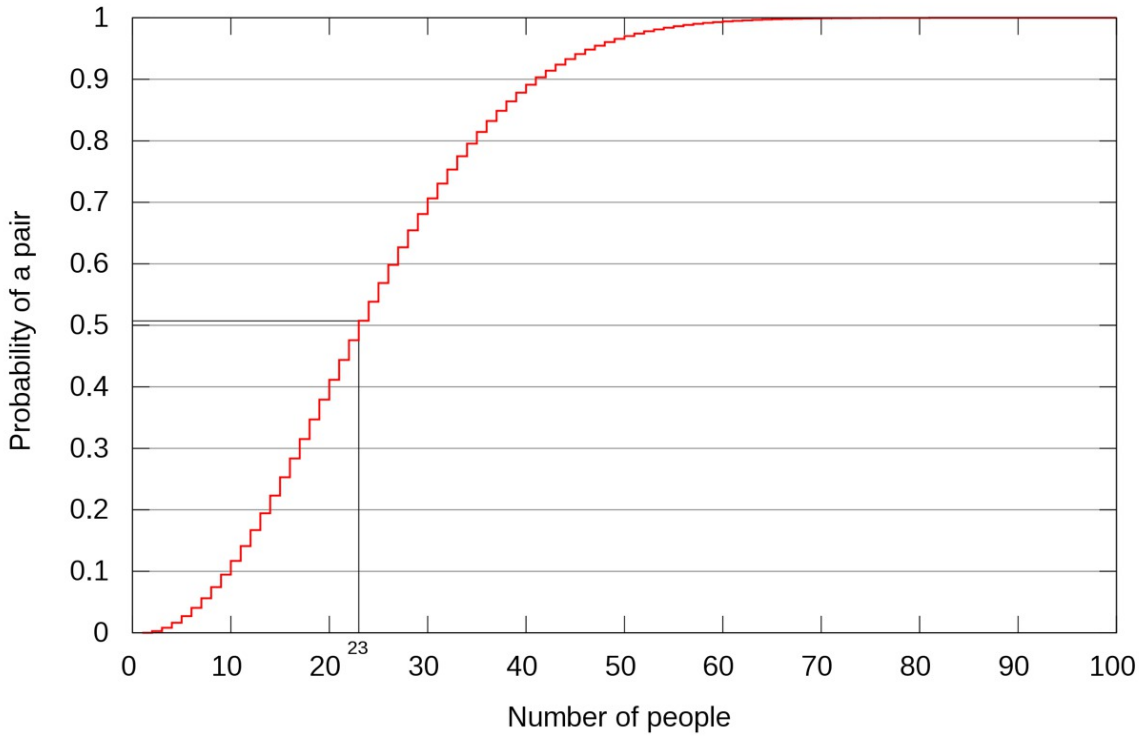
We can attack some *digital signatures* using an attack based on the birthday paradox.

---

[a] Note that the likelihood that the people *will* share birthdays is 1 (one) minus the likelihood that the people will *not* share birthdays.

The Birthday "Paradox"

Likelihood versus number of random choices



In general, in a birthday attack over n items, to achieve a probability of p, we should choose about $\sqrt{2n \ln \frac{1}{1-p}}$ items. In the above, for $n = 365, p = 0.5$ we have about $1.177\sqrt{365} \approx 22.5$. This value (1.177) gives rise to the common approximation we use: *take the square root*[a].

[a] For n-bit attack, only need $2^{\frac{n}{2}}$ values to get 50% chance of success.

The birthday attack - straightforward method

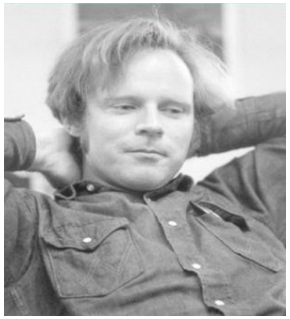
Find x, x' s.t. $\mathcal{H}(x) = \mathcal{H}(x')$; time $\mathcal{O}(2^n)$, and memory $\mathcal{O}(2^n)$..

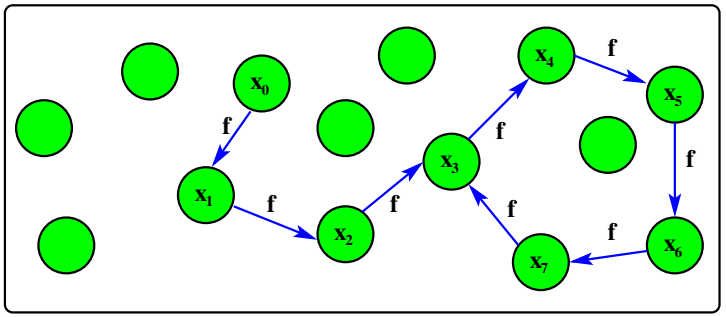
The birthday attack can be used to speedup the search for a hash collision with a "square-root" reduction.

Suppose a digest is n bits, and we have a random set of around $2^{\frac{n}{2}}$ messages (by the birthday attack, the chance of having a collision is around 0.5 probability). A straightforward implementation of a birthday attack stores this set in the memory and then searches for a collision. However, in practice, the memory requirement is more difficult to meet compared to the running time requirement. Imagine if $n = 80$ bits, then there are 2^{40} pairs of messages and digests. Assuming each pair take up 32 bytes, then the total storage is 2^{45} bytes, i.e. 32TB. Nevertheless, 2^{40} hash operations can be done in reasonable time.

The following *improved* birthday attack takes about twice the time, compared to the straightforward implementation, but uses constant memory.

Floyd - for chains like $x_0 \to f(x_0) \to f(f(x_0)) \to \ldots$





Find loop with $f(x) \stackrel{\text{def}}{=} \mathcal{H}(x)$; time $\mathcal{O}(2^n)$, and memory $\mathcal{O}(1)$

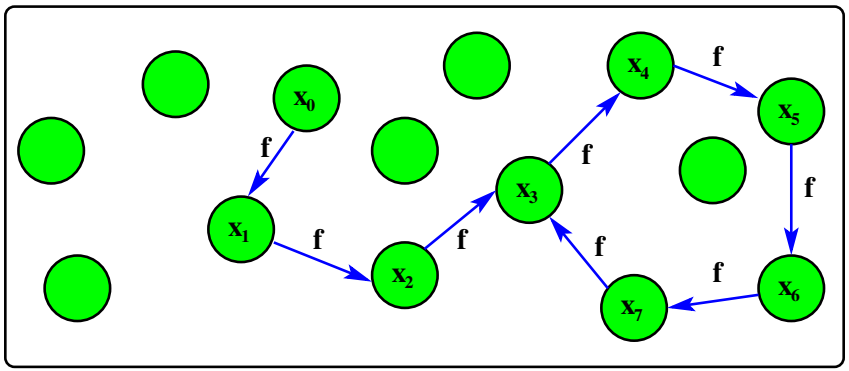
(Robert W.) Floyd's cycle-detection algorithm: the tortoise *t* and the hare *h*. Variables hop through a graph at different rates, stopping when they collide.

Algorithm 5.9: part 1 finds the loop in a (digest/hash) chain

```
def H(x):
    return md5(x.encode('utf-8')).hexdigest()
def f(x):
    return H(x)[:10]

def floyd(x0):
    t = f(x0)
    h = f(f(x0))
    print('Find the loop')
    while (t != h):
        t = f(t)
        h = f(f(h))
```

Floyd - for chains like $x_0 \to f(x_0) \to f(f(x_0)) \to \ldots$

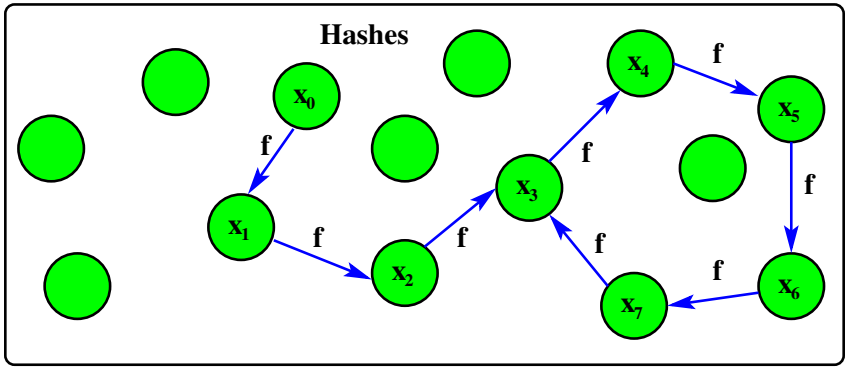


Algorithm 5.9: part 2 finds the collision

```
t = x0
print('Find the collision')
while (t != h):
    if (f(t) == f(h)):
        break
    t = f(t)
    h = f(h)
    if (t != h):
        temp_t = t
        temp_h = h
        pass
return temp_t,temp_h
```

This reruns the *tortoise* and the *hare*, but both just doing one step at a time. The *tortoise* restarts at the beginning, and the *hare* at the result of part 1. They meet at the beginning of the loop (where the collision is).

Finding a collision with floyd.py

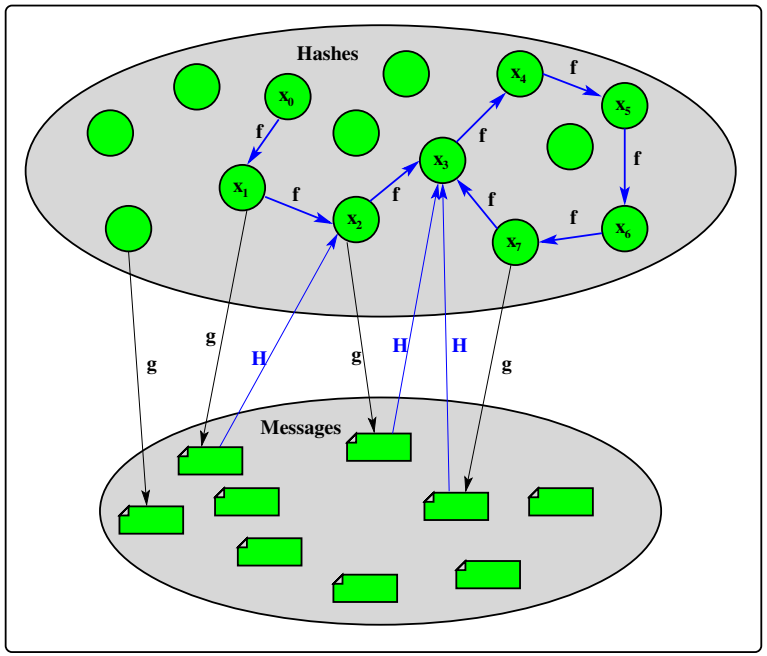


Run it with a starting point...

```
hugh@comp % python3 floyd.py 42
   Find the loop
   Find the collision
   Tortoise f375526901 > cfd1d92d5b1d0fcae405b61165d261ed
   Hare     0eecccc477 > cfd1d92d5bf0b71d1655bc12981bd608
hugh@comp % md5 -s f375526901
   MD5 ("f375526901") = cfd1d92d5b1d0fcae405b61165d261ed
hugh@comp % md5 -s 0eecccc477
   MD5 ("0eecccc477") = cfd1d92d5bf0b71d1655bc12981bd608
hugh@comp %
```

Note that we are only matching the first 10 hex digits of the digest.

Practical use in a message/digest collision attack

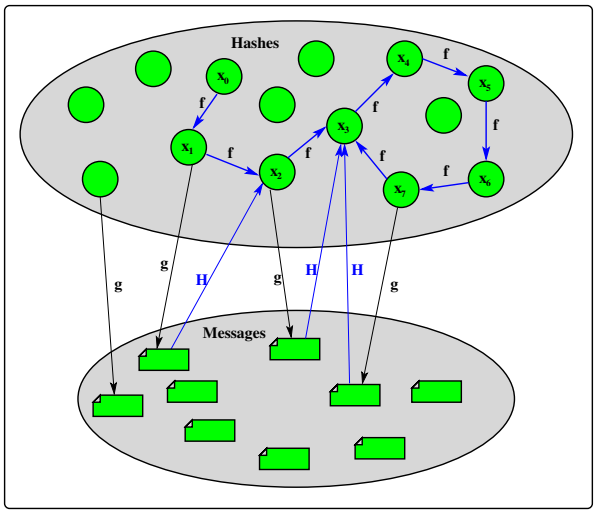


Apply Floyd by using $f(x) \stackrel{\text{def}}{=} \mathcal{H}(g(x))$

Consider messages $m \in M$, digests $h \in H$, the functions $g : H \to M$, and $f : H \to H$ defined as $f(x) \stackrel{\text{def}}{=} \mathcal{H}(g(x))$. The elements in the *chain/trail* of digests are mapped to the messages. In the textbook, p168, there is an example with *good*, and *bad* messages about Bob:

$$\begin{aligned} g(0000) &= \text{Bob is a good and honest worker.} \\ g(0001) &= \text{Bob is a difficult and....} \end{aligned}$$

Code changes for the collision attack



Algorithm 5.9 for 2^{16} messages in G. Few code changes...

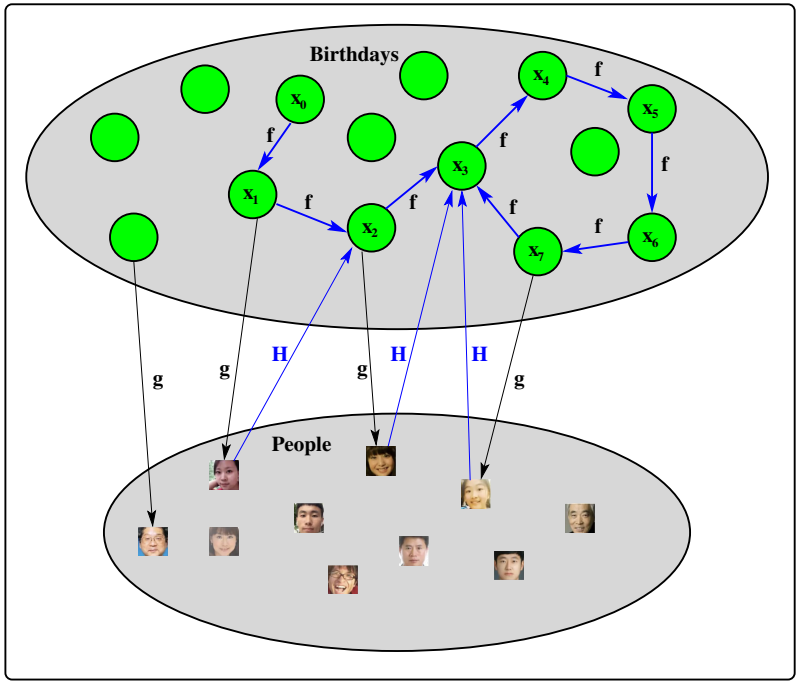
```
G = [m1,m2,m3...]
def g(x):
    return G[int(x[:4],base=16)]
def f(x):
    return H(g(x))[:4]
```

This code only matches the first 4 hex digits of the digest:

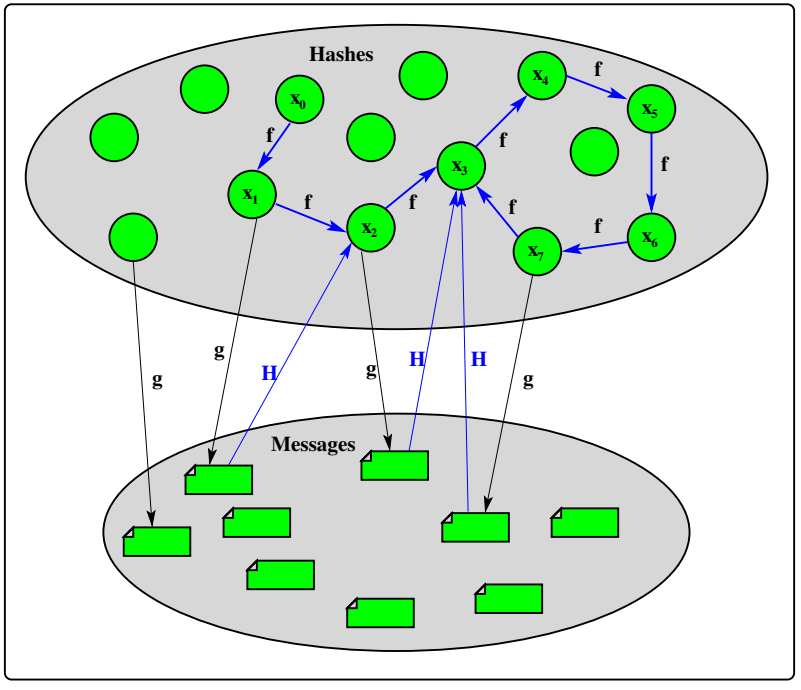
```
hugh@comp % python3 floydfile4.py 423485
    Tortoise Lord Baltinglas...  > e09c3f2d63f7841d33fdbe0bfb4fd444
    Hare     Tell me more ab...  > e09c59046d3b1201931ff649d8b287e8
```

Birthdays and hashes? All are $f(x) \stackrel{\text{def}}{=} \mathcal{H}(g(x))$

People sharing birthdays



Message digests colliding



B is the set of birthdays, P the set of people. and we have:

- $g : B \to P$ (random map)
- $H : P \to B$ (birthday)
- $f : B \to B$

H is the set of digests/hashes, M the set of message, and we have:

- $g : H \to M$ (random map)
- $H : M \to H$ (here MD5)
- $f : H \to H$

References

Commitment

- Textbook Section 5.6.5 (pg 187)
- Commitment schemes and Zero-Knowledge Protocols, https://www.daimi.au.dk/~ivan/ComZK07.pdf
- Wiki at https://en.wikipedia.org/wiki/Commitment_scheme
- Damgard and E. Fujisaki, An Integer Commitment Scheme based on Groups with Hidden Order. (supports ZK proofs of some arithmetic properties of the committed value.).

Rainbow tables

- Textbook: Section 5.4 (pg 164)
- Reference: https://en.wikipedia.org/wiki/Rainbow_table https://kestas.kuliukas.com/RainbowTables/
- The original research paper: P. Oechslin. Making a Faster Cryptanalytical Time-Memory Trade-Off, CRYPTO 2003. https://iacr.org/archive/crypto2003/27290615/27290615.pdf

Symmetric encryption schemes

$$\mathrm{ENC}_k = (\mathrm{Gen}(1^n), \mathrm{Enc}_k(m), \mathrm{Dec}_k(c)) \qquad \textbf{(ch1)}$$

Properties	Defs	Constructions	Proofs
Perfect secrecy	2.3	(One time pad)	Thm 2.9
↕			Lemma 2.6
Perfect indistinguishability	2.5		
↓			
EAV-secure	3.8	3.17 (PRG)	Thm 3.18
↑			Proof given
EAV-Multi-encryption	3.19		
↑			
CPA-Multi-encryption	3.22		
↕			Thm 3.24
CPA-secure	3.23	3.30 (PRF)	Thm 3.31
↑			
CCA-secure	3.33		
+			
Unforgeable	4.16		
↕			Thm 4.19
Authenticated	4.17	4.18 (Enc-then-Auth)	

MAC and HASH schemes

$$\mathrm{MAC}_k = (\mathrm{Gen}(1^n), \mathrm{Mac}_k(m), \mathrm{Vrfy}_k(m,t)) \quad \textbf{(4.1)}$$

Properties	Defs	Constructions	Proofs
Unforgeable	4.2		
↑			Discussed
Strong	4.3	4.5 (Using PRF)	Thm 4.5
		4.7 (Variable length)	Thm 4.8
		4.11 (CBC-MAC)	Thm 4.12
		4.11(a) (Variable CBC-MAC)	

$$\mathrm{HASH} = (\mathrm{Gen}(1^n), \mathcal{H}^s(x)) \quad \textbf{(5.1)}$$

Properties	Defs	Constructions	Proofs
CollisionResistant	5.2	5.3 (Long message)	

Commitment scheme

$$\mathrm{COM} = (\mathrm{Setup}(1^n), \mathrm{Commit}(a), \mathrm{Open}(c_a)) \quad \textbf{(ch5)}$$

Properties	Defs	Constructions	Proofs
Binding			
Hiding			
Secure (Binding+Hiding)	5.13	(Hash based system)	