CSCI 331: Introduction to Computer Security

Lecture 6: Password Cracking, part 1

Instructor: Dan Barowy

Williams

Topics

Lab 1

Crypto refresher

Password database attacks

Your to-dos

1. Reading response (Oechslin) **due Wed 9/29**.
   a. How to read this: understand the **big idea**.
2. Project part 1 **due Sunday 10/3**.
3. Lab 2 **due Sunday 10/10**.

<u>Office hours:</u>
Tuesday 1:10-2:35pm (**TBL 301**)
Thursday 4-6pm (**TBL 301**)

Lab 1

1. Labs should make you stretch, not be **hard**.
2. Calibration—**everybody** gets **1 extra late day**.
3. Extra office hours tonight: 6:30-7:30pm
   —> TCL 312 <—
4. Feeling confused? This is **not bad**. Get in touch.
5. Feeling discouraged? This is **not good**. Get in touch.
6. Nobody who hands in **something** will do poorly in this class, because…
7. **Resubmissions**.

Office hours:
Tuesday 1:10-2:35pm (**TBL 301**)
Thursday 4-6pm (**TBL 301**)
+
Tonight only 6:30-7:30pm (**TCL 312**)

Cryptography refresher

Encryption is the **process of encoding a message** so that it can be read only by the sender and the **intended recipient**.

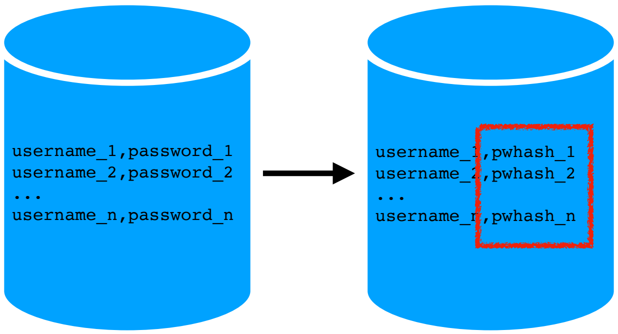
- A **plaintext** p is the original, unobfuscated data. This is information you want to protect.
- A **ciphertext** c is encoded, or encrypted, data.
- A **cipher** f is an algorithm that converts **plaintext** to **cipertext**. We sometimes call this function an **encryption function**.
  - More formally, a cipher is a function from plaintext to ciphertext, $f(p)=c$. The properties of this function determine what kind of encryption scheme is being used.
- A **sender** is the person (or entity) who enciphers or encrypts a message, i.e., the party that converts the plaintext into cipertext. $f(p)=c$
- A **receiver** is the person (or entity) who deciphers or decrypts a message, i.e., the party that converts the ciphertext back into plaintext. $f^{-1}(c)=p$

See the reading Why Stolen Password Databases are a Problem for a little more nuance.

Lab 2

Scenario

Entire password database **leaked** (bug; misconfiguration; theft by authorized personnel).



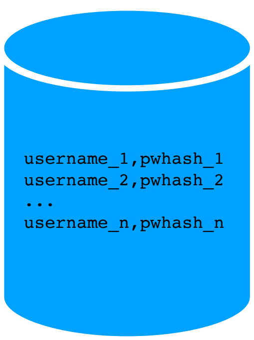
We keep passwords in **hashed** form.

Hashes are **not invertible**.

Password database attacks

- Random guessing attack
- Enumeration attack
- Dictionary attack
- Precomputed hash chain attack
- Rainbow table attack

Random guessing




```
for each entry in database:
  not_found = true
  // try until found
  while not_found:
    // random plaintext
    p = randPassword()
    // create ciphertext
    c = hash(p)
    // compare
    if c = entry.pwhash:
      print entry.pwhash, p
      not_found = false
```

Complexity?

Random guessing: complexity (one pw)

m = # of possible passwords

p = probability that random guess is correct

= 1/**m**

X = # guesses until success

$E[X] = (1-p)/p$ (geometric dist)

$= m - 1$

O(**m**) average **per pw** O(**mn**) average for **all pw**

Wait a sec…

Suppose password is: `password`

Hashes to `5f4dcc3b5aa765d61d8327deb882cf99`

Suppose we guess: `b1d78hdd`

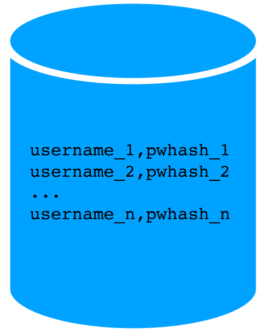
Hashes to `310fca1c70732a8191d0199bddba3a97`

Clearly that was not the passwrd.

Does it make sense to guess: `b1d78hdd` again?

No! We should **remember** bad guesses.

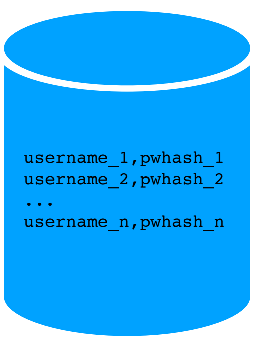
Enumeration: slightly better




```
for each entry in database:
  i = 0
  not_found = true
  // try until found or out of pt
  while not_found && i < NUM_PT:
    // gen ith possible plaintext
    p = genPassword(i)
    // create ciphertext
    c = hash(p)
    // compare
    if c = entry.pwhash:
      print entry.pwhash, p
      not_found = false
    i++
```

Complexity?

Enumeration: complexity



m = # of possible passwords

Average guesses to find **one pw**:

O(**m**/2)

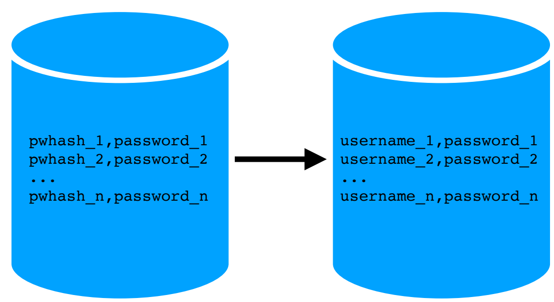
Average guesses to find **all pw**:

O(**n** x **m**/2)

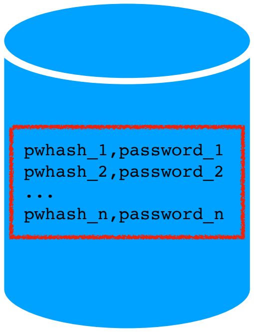
Dictionary attack

A **dictionary attack** is a form of **brute force attack** technique for **recovering passphrases** by systematically **trying all likely possibilities**, such as words in a dictionary.

Critically, a dictionary attack only tries each possibility once. It **trades space for time**.



Dictionary: much better



Ahead of time:

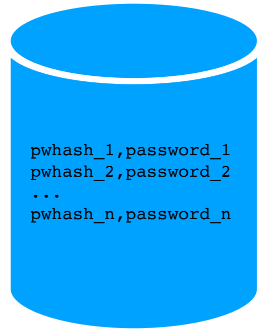
```
while i < NUM_PT:
    // gen ith possible plaintext
    p = genPassword(i)
    // create ciphertext
    c = hash(p)
    // save
    cracked_db[c] = p
```

Later:

```
for each entry in database:
  print cracked_db[entry.pwhash]
```

Complexity?

Dictionary attack: complexity



m = # of possible passwords

Time to compute dictionary:

O(**m**)

Time to lookup **one pw**:

O(**log m**)

Time to lookup **all pws**:

O(**n log m**)

Space needed:

O(**m**)

Activity: How much space?

It depends on the **number of possible passwords**.

Password scheme:

- Uppercase letters and numbers, except `O` and `I`.
- Up to 8 digits

How many passwords are there?

Activity: How much **space**?

m = # of passwords

$$= \sum_{k=1}^{8} 34^k = 1\,839\,908\,871\,710$$

≈ 1.8 trillion passwords

Suppose per-pw storage is **always 16 bytes**.
(8 bytes for cipertext, 8 bytes for plaintext)

16 x (1.8 x 10^{12}) bytes

≈ 29 terabytes

Is this a **feasible attack**?

https://www.amazon.com/Seagate-ST18000NM000J-Internal-Surveillance-Supported/dp/B08K98VFXT

Is this a feasible attack?

space: ≈ 29 terabytes

Time?

Suppose I can generate 1 million pw/sec

$(1.8 \times 10^{12}) / 10^{6} \approx (1.8 \times 10^{6})$ seconds

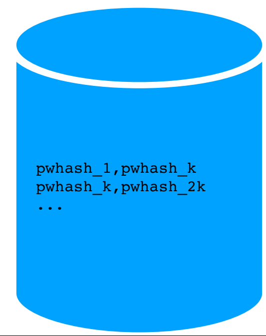
≈ **21 days** with **one computer**.

This is **definitely feasible**!

Precomputed hash chains

A **PCHC attack** is a form of **brute force attack** technique for **recovering passphrases** by systematically **trying all likely possibilities**, such as words in a dictionary.

Critically, a PCHC attack only tries each possibility once. It **trades space for time, but it compresses the database.**



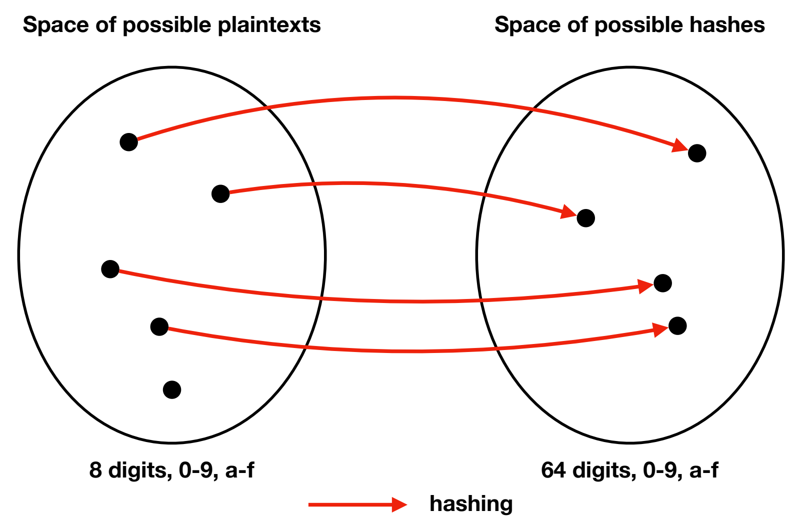
Hash function

Suppose we have:

$f(p)=c$, a **cipher** that maps **plaintexts** to **ciphertexts**; in this case, a **hash function**.

Because f is a hash function, there is **no inverse function** such that $f^{-1}(f(p))=p$.

Hash function



plaintext: “`9a55302d`” **ciphertext:** “`4651f1799e5e36c878f3d980c59e94ae`”

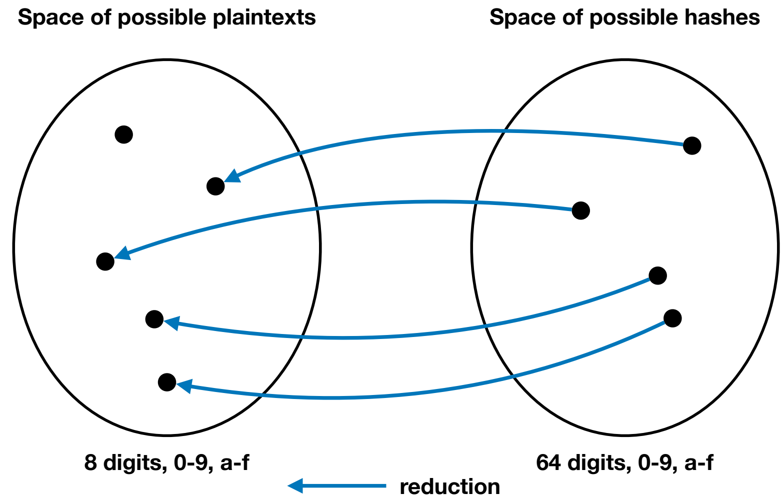
Reducer function

Suppose we have:

$r(c)=p$, that maps **cipertexts** to **plaintexts**, called a **reducer**.

A reducer is **not the inverse** of the hash!

Reducer function

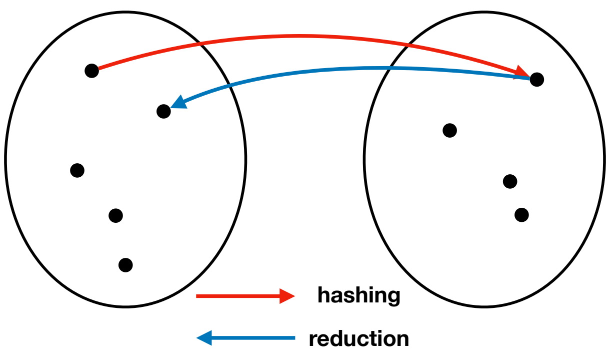


ciphertext: "`4651f1799e5e36c878f3d980c59e94ae`" **plaintext:** "`4651f179`"

Reducer function properties

A reducer $r(c)=p$ only needs to satisfy a couple properties.

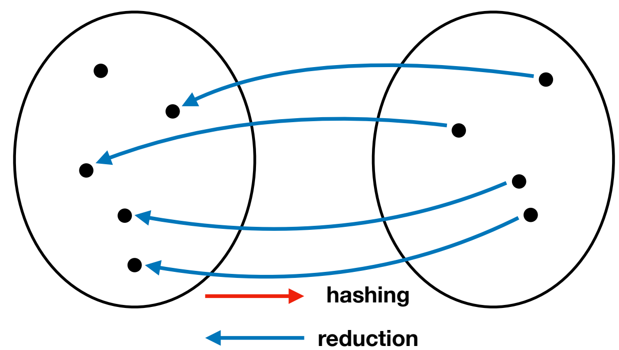
1. A reducer's output, p, should map to the same domain as the *input* of the hash function, $f(p)=c$ (i.e,. plaintexts)



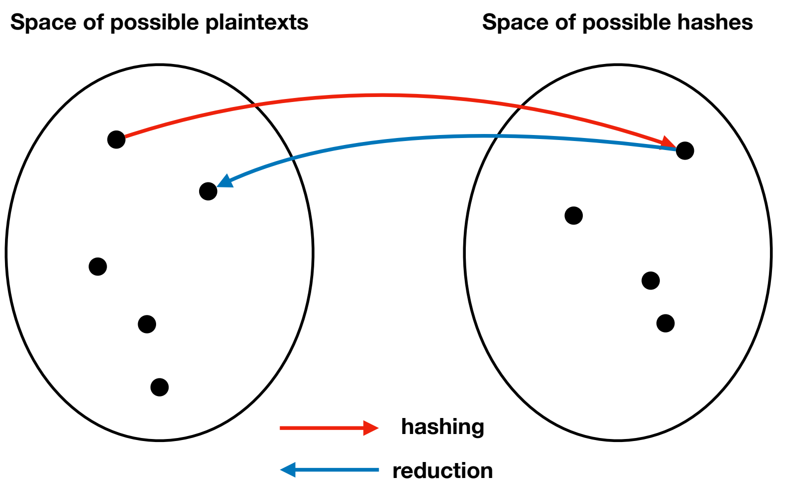
Reducer function properties

A reducer $r(c)=p$ only needs to satisfy a couple properties.

2. All plaintexts should be selected, given the space of ciphertexts, with equal probability.



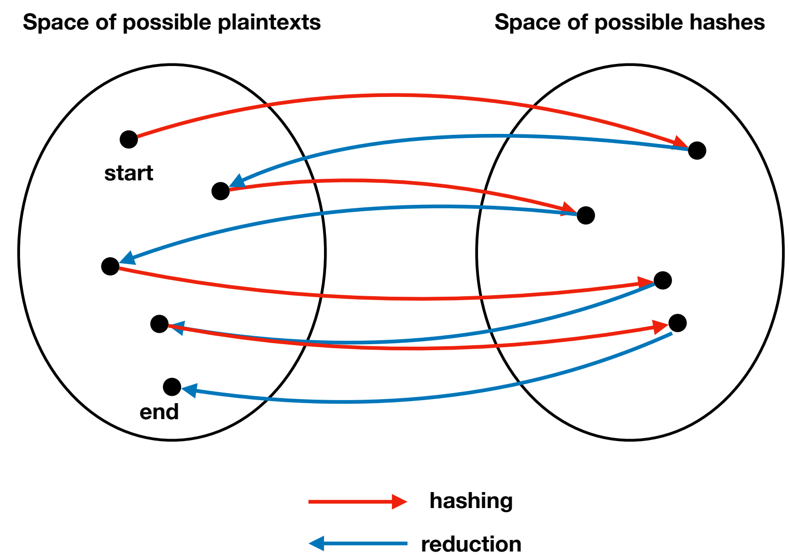
Hash-reduce "round trip"



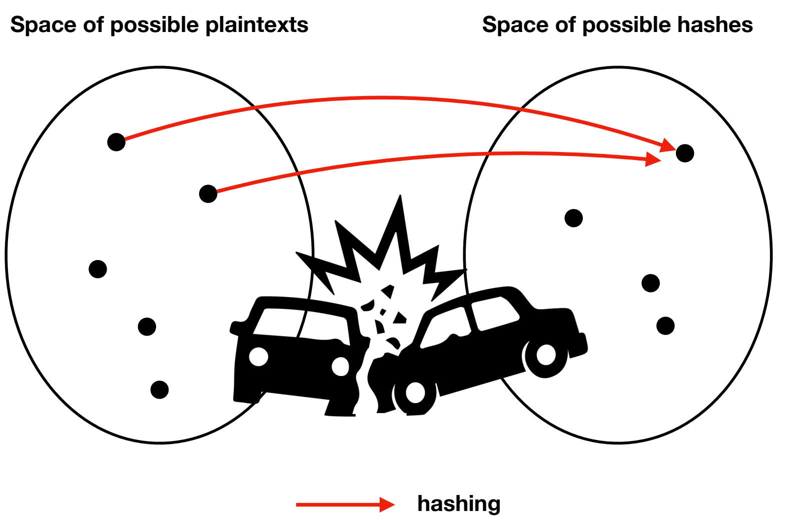
plaintext: "`9a55302d`" **ciphertext:** "`4651f1799e5e36c878f3d980c59e94ae`"

ciphertext: "`4651f1799e5e36c878f3d980c59e94ae`" **plaintext:** "`4651f179`"

Hash chain

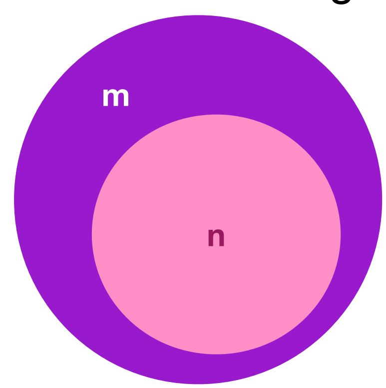


Hash functions are not usually perfect



Collisions!

Hashes are guaranteed to collide



m: # of passwords **n**: # of hashes

If **m > n**, we know that **at least (m-n)/m** must collide.

"pigeonhole principle"

Thought experiment

Let's **suspend disbelief** for a moment.

1. Our **hash** function is **perfect**, chooses ciphertext with probability 1/m.

2. Our **reducer** function is **perfect**, chooses plaintext with probability 1/m.

3. The **combination** of hash function and reducer function is also **perfect**.

Real cryptographic hash functions are designed to approximate #1.

Real reducers can actually be perfect.

Thought experiment

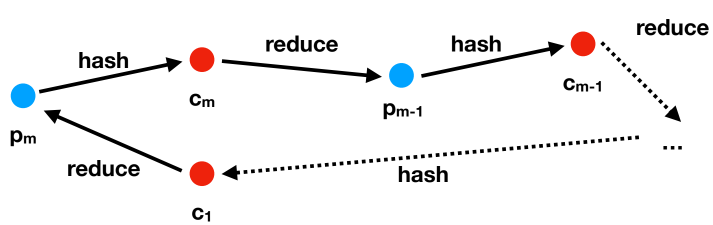
What can we **do** with this information?

Thought experiment

plaintexts
ciphertexts

Suppose f(p_i) = c_i

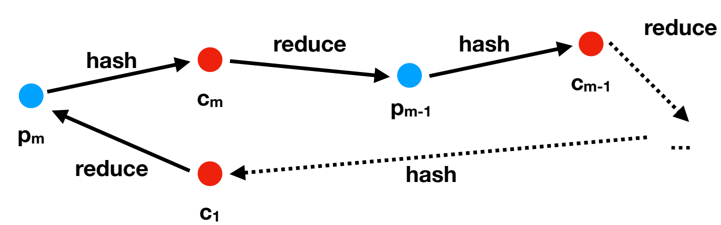
Suppose $r(c_i) = p_{i-1}$ if $i>1$ otherwise p_m



Thought experiment

plaintexts
ciphertexts

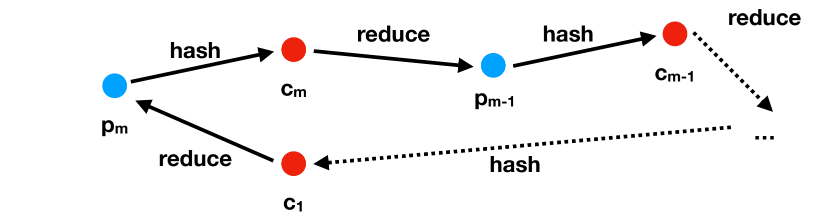
Such a scheme (a *hash chain*) lets us generate all plaintexts (and hashes) from a seed plaintext.



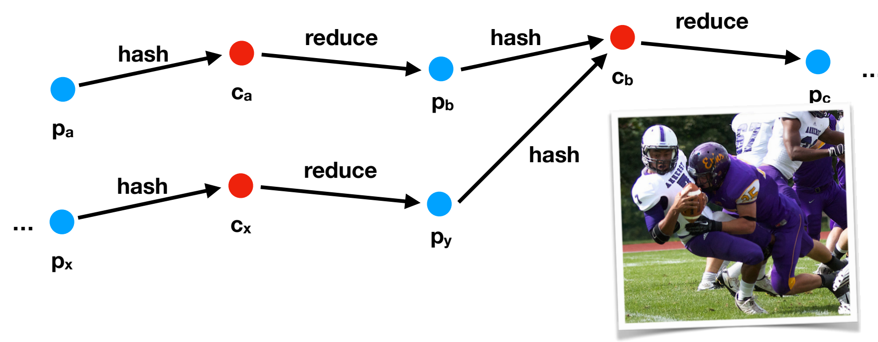
Only need to save the seed. **Drawbacks?**

Thought experiment: **drawbacks**

- Saving just the first password **buys us nothing**. On average, we have to compute **O(m/2)** hash-reductions to find a password.
- **Hash functions collide!**



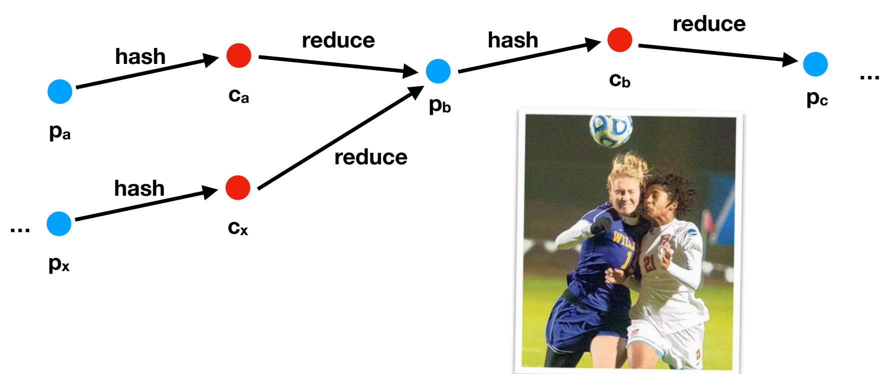
Collisions in a hash chain



After the **collision**, the chain "**loops**."

Collisions prevent us from enumerating the **entire space**!

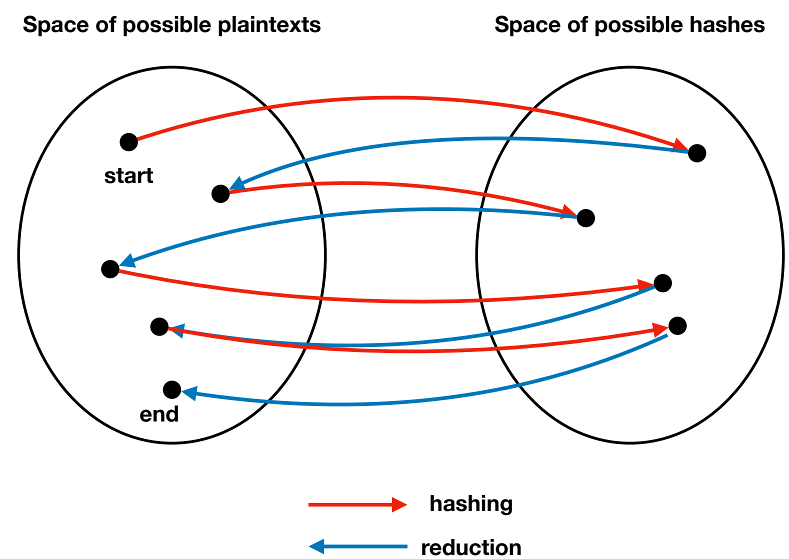
Collisions in a hash chain



Reducers can produce collisions too!

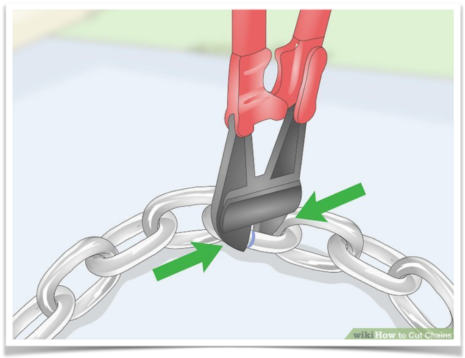
This is what we mean by an **imperfect reducer**.

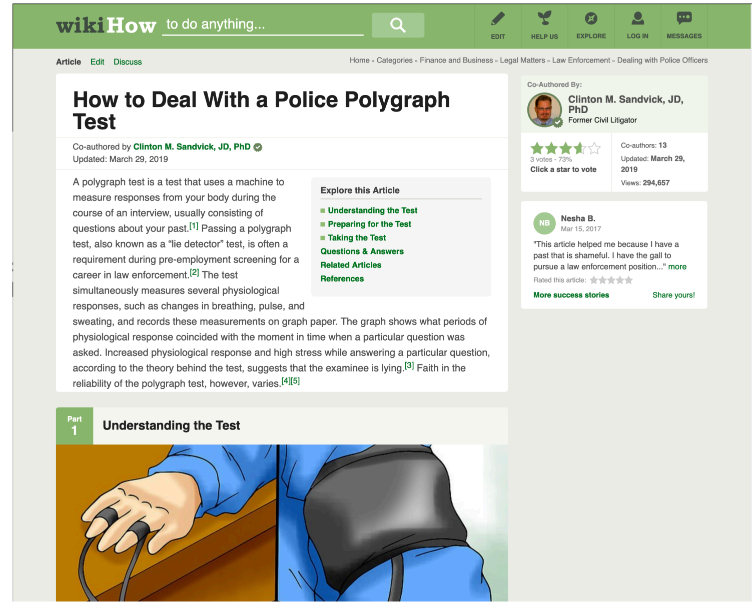
Hash chain



Hash chain of length k

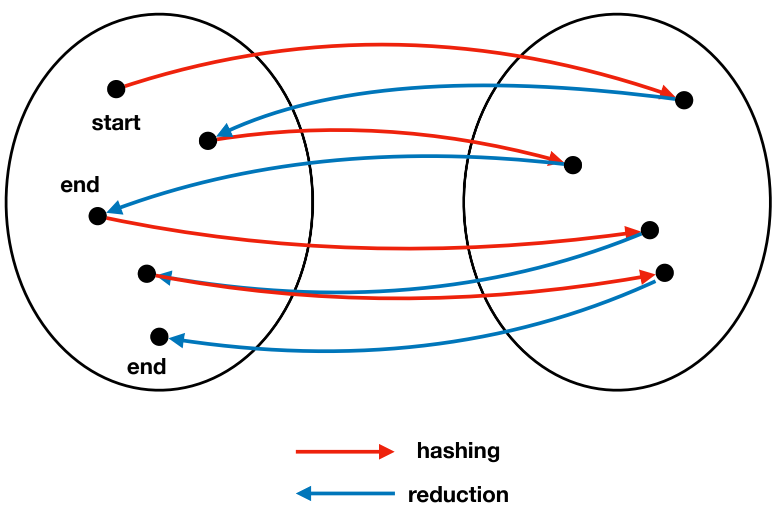
We are going to chop up our long chain into **smaller chains** of length **k**.





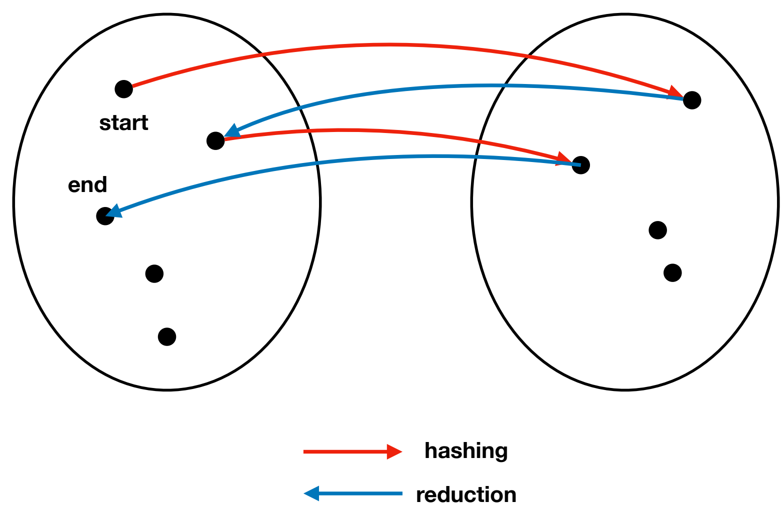
Hash chain

Space of possible plaintexts **Space of possible hashes**

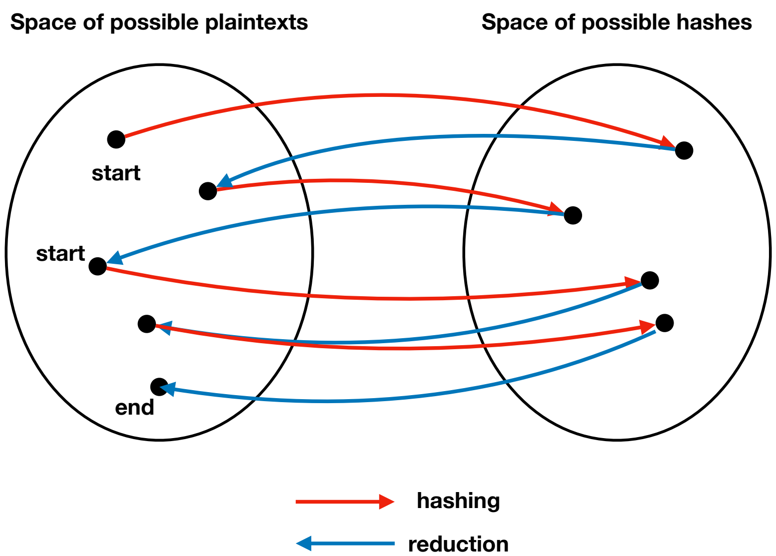


Hash chain of length 2

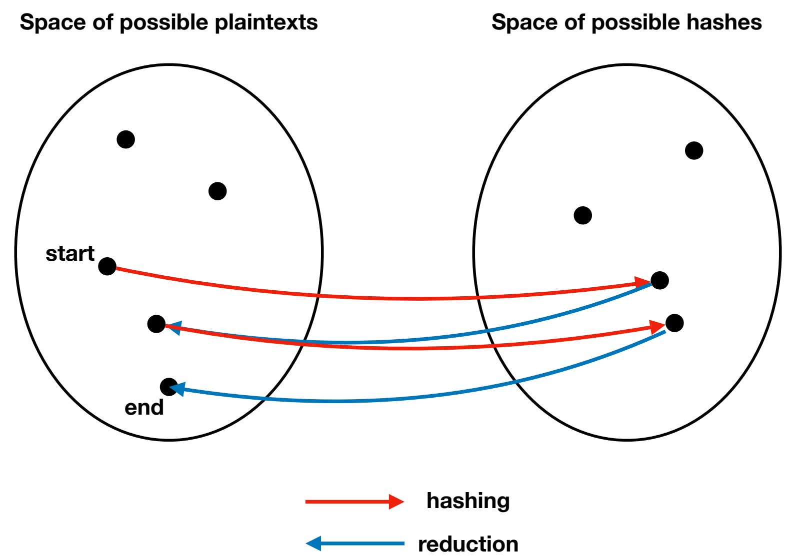
Space of possible plaintexts **Space of possible hashes**



Hash chain



Hash chain of length 2



Store only **start** and **end**

```
start, end
pm   , pm-3
…
p5   , p3
p3   , p1
```

Store it **backward**

```
end, start
pm-3   , pm
…
p3   , p5
p1   , p3
```

Suppose you are given c_4

```
end, start
pm-3   , pm
…
p3   , p5
p1   , p3
```

$c_4 \xrightarrow{\text{reduce}} p_3$

Is p_3 an **end point**? **yes**

Hash and **reduce** from **start point**.

password! original ciphertext

$p_5 \xrightarrow{\text{hash}} c_5 \xrightarrow{\text{reduce}} p_4 \xrightarrow{\text{hash}} c_4$

Recap & Next Class

Today we learned:

Password attacks

Password attack complexity

Trading space for time

Next class:

PCHC lookup algorithm