Fundamentals of Computer Security

Fall 2022

[Radu Sion]

Hash Functions

Thanks to [Ari Juels] for parts of this deck!

Hash functions

The workhorses of modern cryptography

Definition

Cryptographic hash h: $A \rightarrow B$:

1. For any $x \in A$, $h(x)$ is **easy to compute**
2. $h(x)$ is of fixed length for any x (**compression**)
3. For any $y \in B$, it is computationally infeasible to find $x \in A$ such that $h(x) = y$. (**pre-image resistance**)
4. It is computationally infeasible to find <u>any</u> two inputs x, $x' \in A$ such that $x \neq x'$ and $h(x) = h(x')$ (**collision resistance**)
5. Alternate form of 3 (stronger): Given any $x \in A$, it is computationally infeasible to find a different $x' \in A$ such that $h(x) = h(x')$. (**second pre-image resistance**)

Hash function

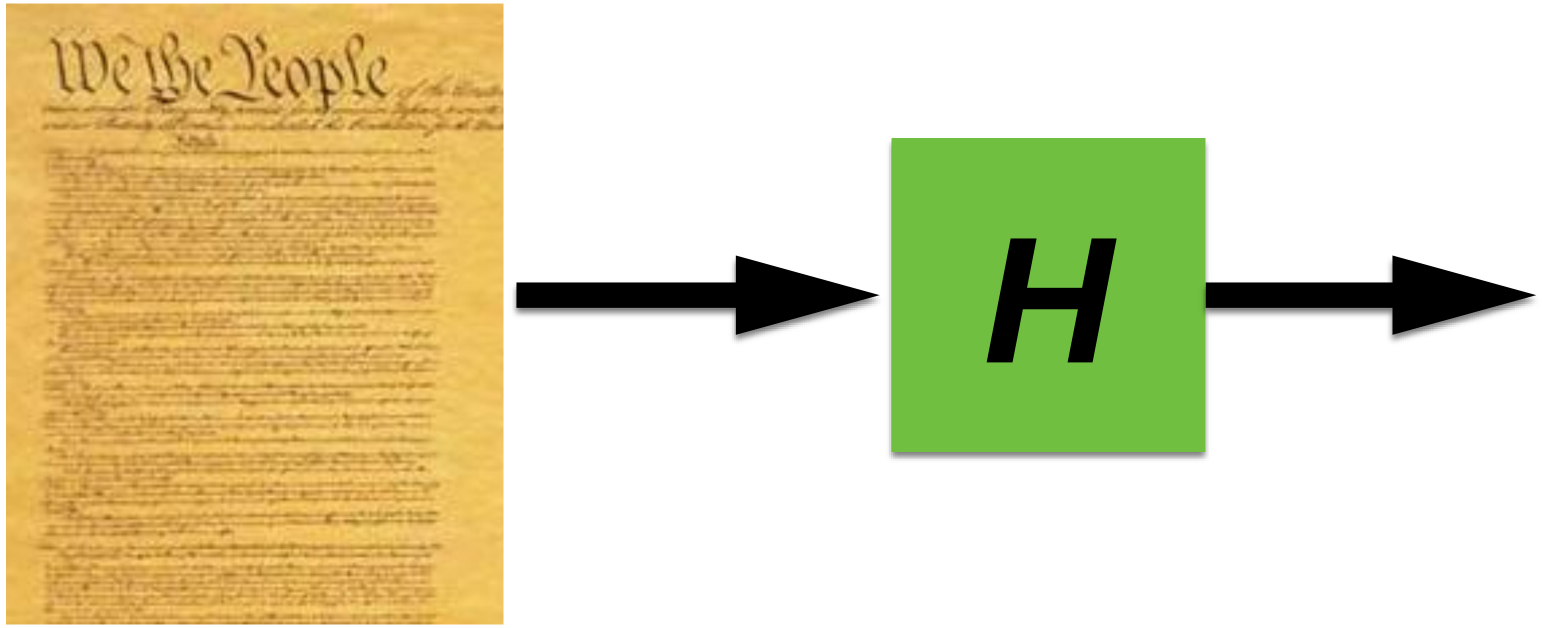
Maps ***message*** *x* of any length to short, fixed-length, random-looking *digest* *H*(*x*)

---

message $x \in \{0,1\}^*$

digest $H(x) \in \{0,1\}^n$

e.g., $n = 256$




```
e49b69c1 efbe4786
0fc19dc6 240ca1cc
2de92c6f 4a7484aa
5cb0a9dc 76f988da
```

Hash function

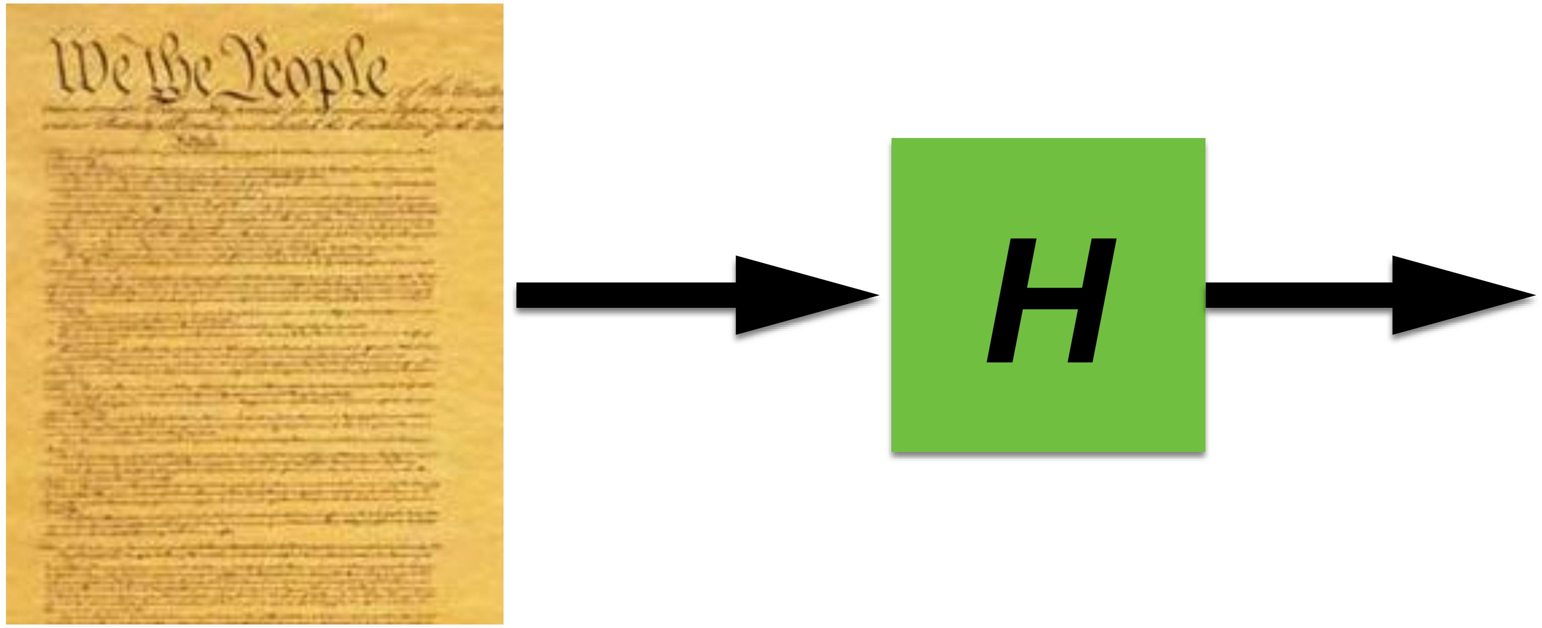
Think of it as both:

- A unique “fingerprint” of message *x*
- A very lossy compression of message *x*

---

message $x \in \{0,1\}^*$

digest $H(x) \in \{0,1\}^n$




```
e49b69c1 efbe4786
0fc19dc6 240ca1cc
2de92c6f 4a7484aa
5cb0a9dc 76f988da
```

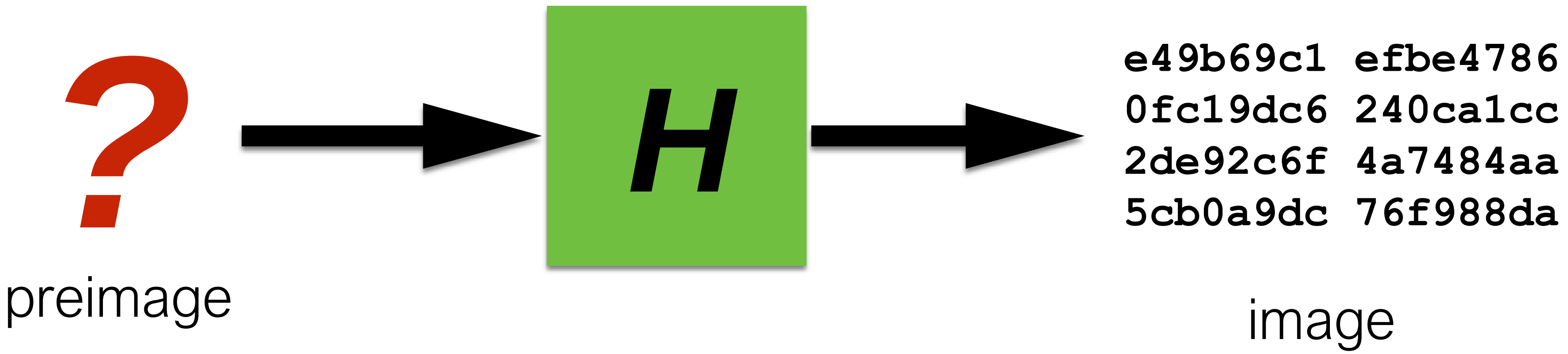
Cryptographic hash function

- Common examples: MD5, SHA-1, SHA-256 (n = 256)
- $H(x)$ should be easy to compute
- Two key security properties for a crypto hash function H:

1. **pre-image resistance:**
   - *Image* is any n-bit value y
   - Given image y, a ***preimage*** is any x s.t. $H(x)$ = y
   - Preimage resistance: given ***random*** y (uniform over $\{0,1\}^n$), it's *infeasible* to find image x, i.e., x such that

$$\boldsymbol{H(x) = y}$$

Pre-image resistance

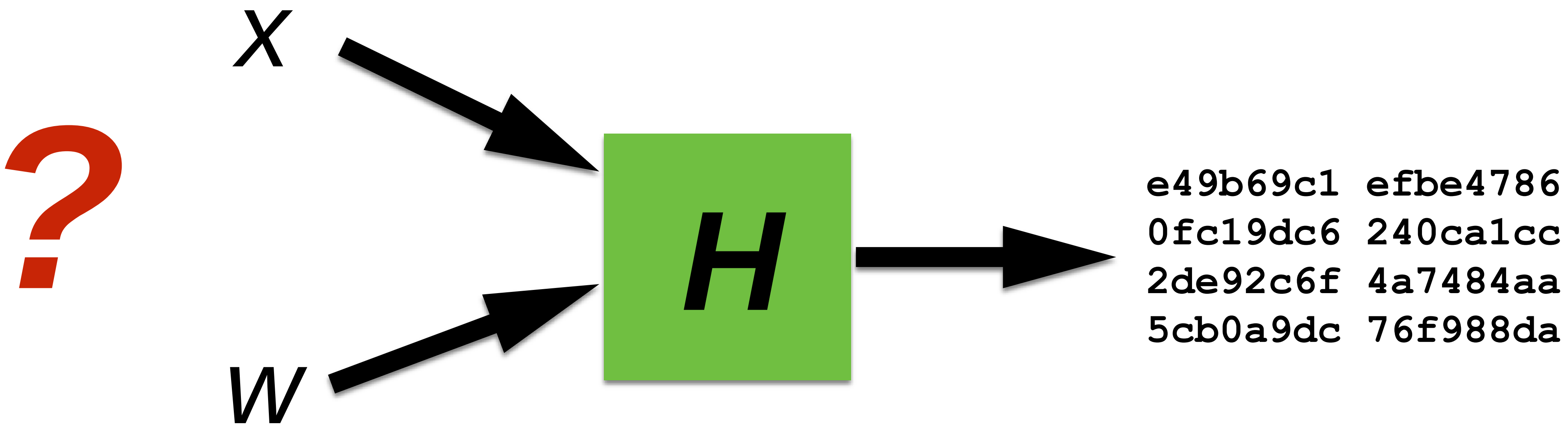


Note, though, that for one image, there are infinitely many preimages ! (Why?)

Collision-resistance

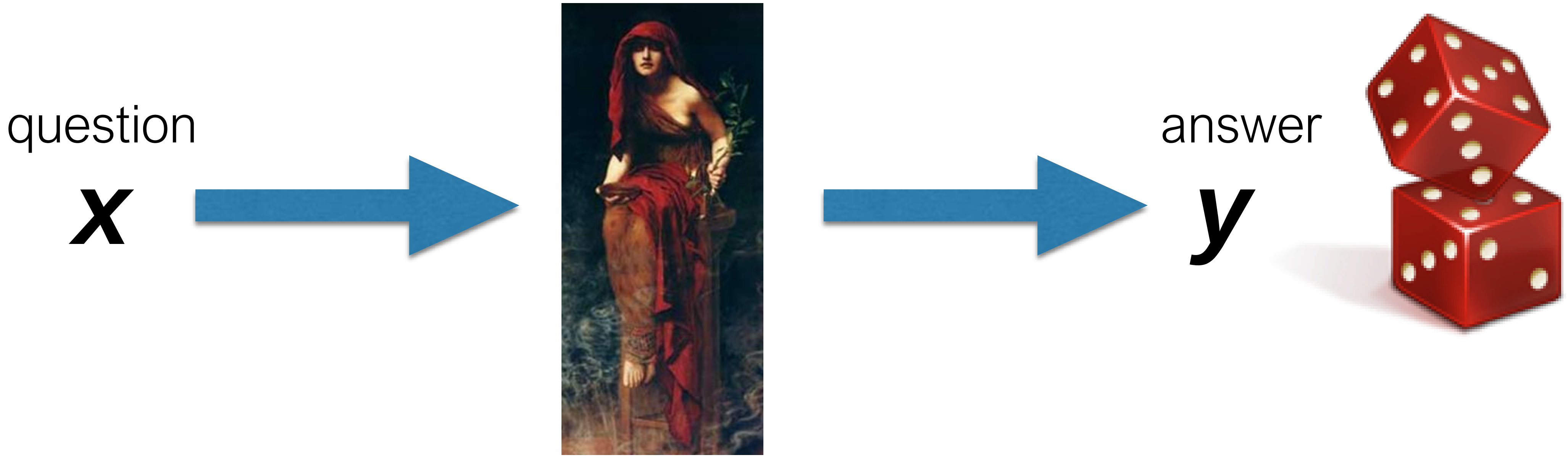
2. **Collision-resistance:** It is hard to find any pair of inputs (w,x) such that

$$H(w) = H(x) = y$$



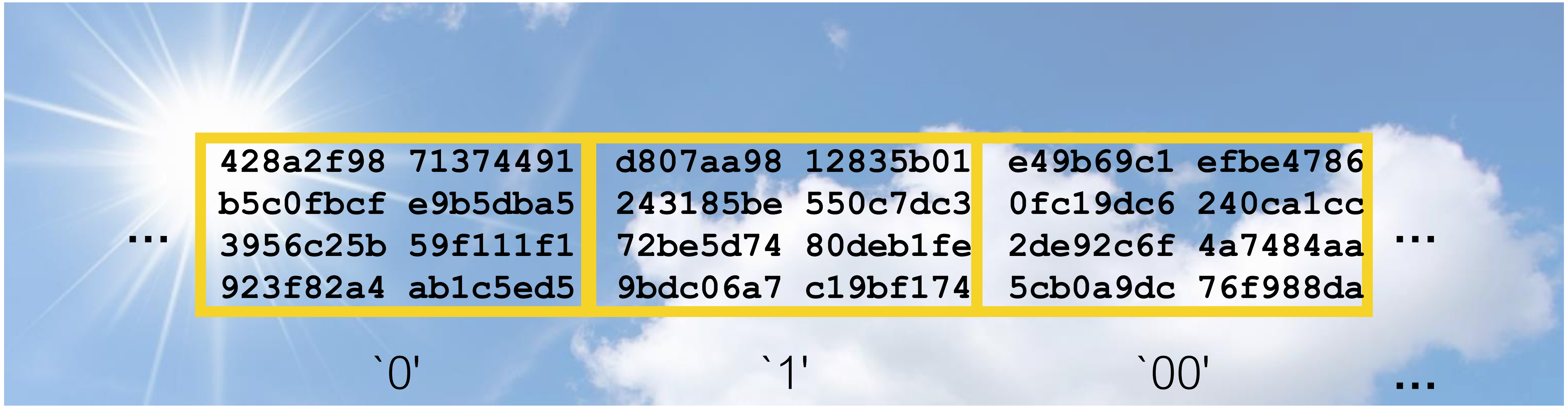
Random Oracle Model (ROM)

- Simple concept
- *Captures other, even stronger security properties than preimage- and collision-resistance*
- In this class, we'll use this *ideal* model of hash functions.

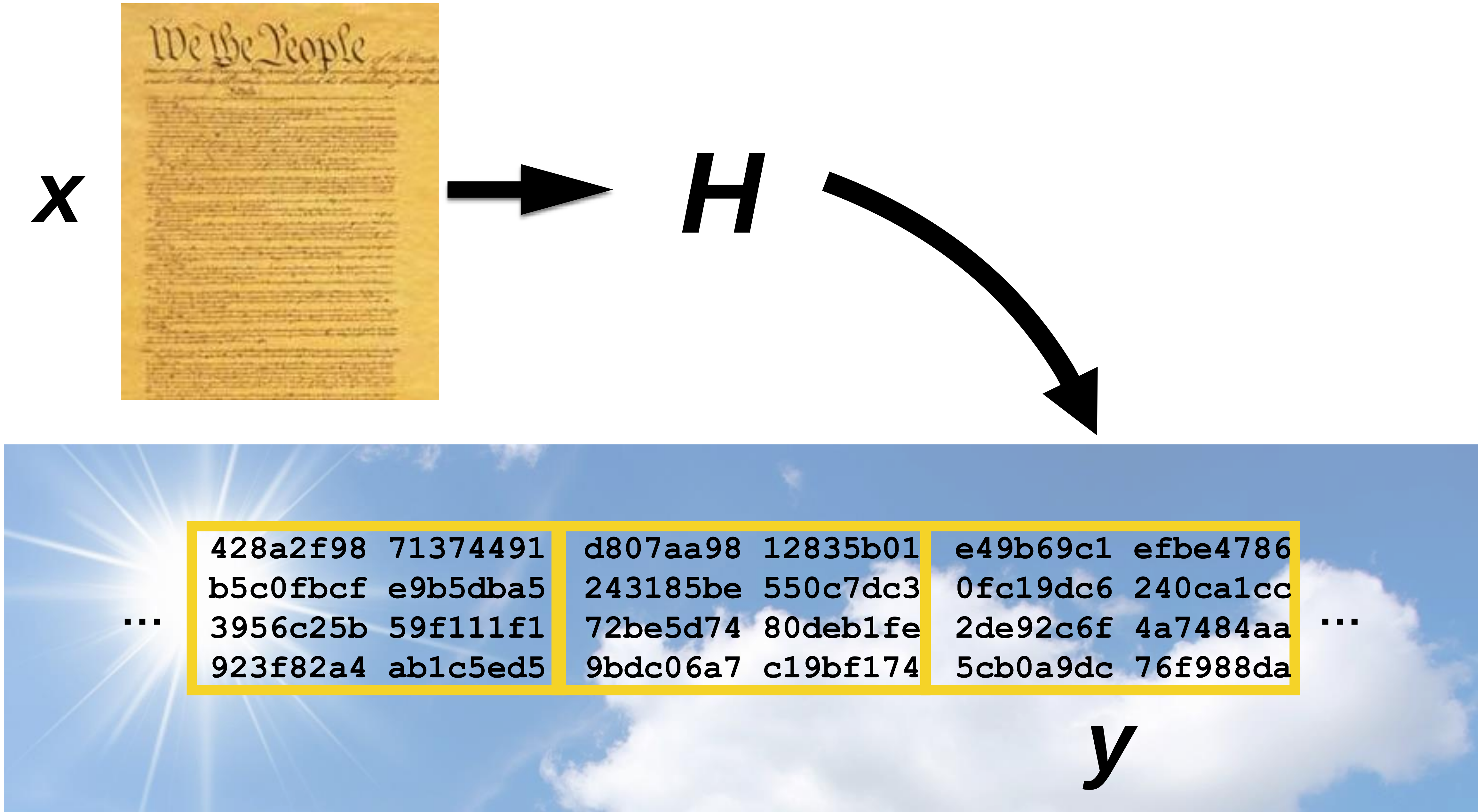


Random Oracle Model (ROM)

- Someone (NIST, NSA, Ron Rivest, God) wrote *infinitely* long tape of cells in the sky
- Each cell contains uniformly random *n*-bit (e.g., 256-bit) value
- Each bitstring *x* (arbitrary length) corresponds to unique cell
    - I.e., *x* = ‘0’ mapped to first cell, *x* = `1’ to second, *x* = `00’ to third, *x* = `01’ to fourth, etc.
- *H*(*x*) outputs value in cell for *x*

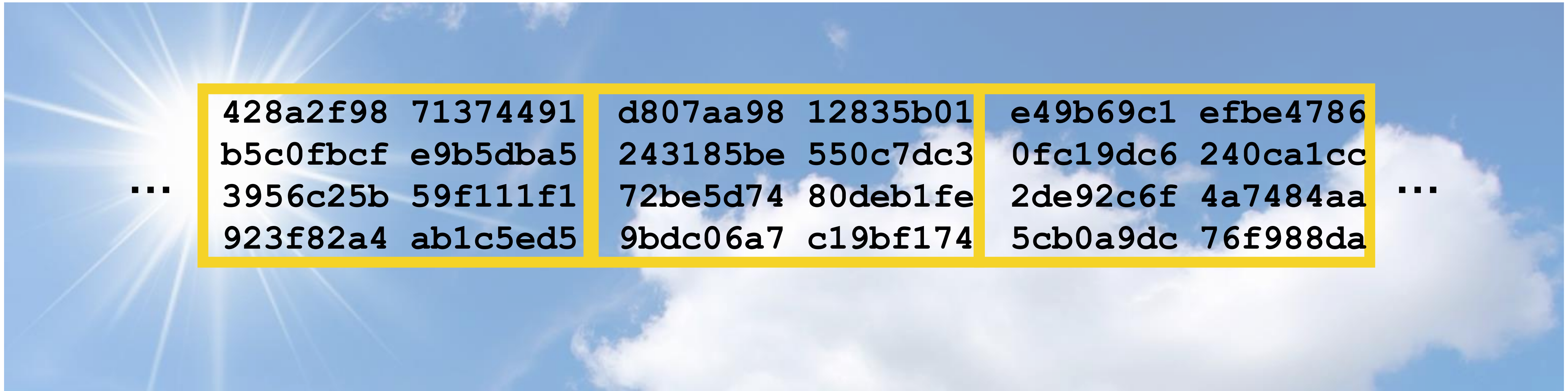


Random Oracle Model (ROM)



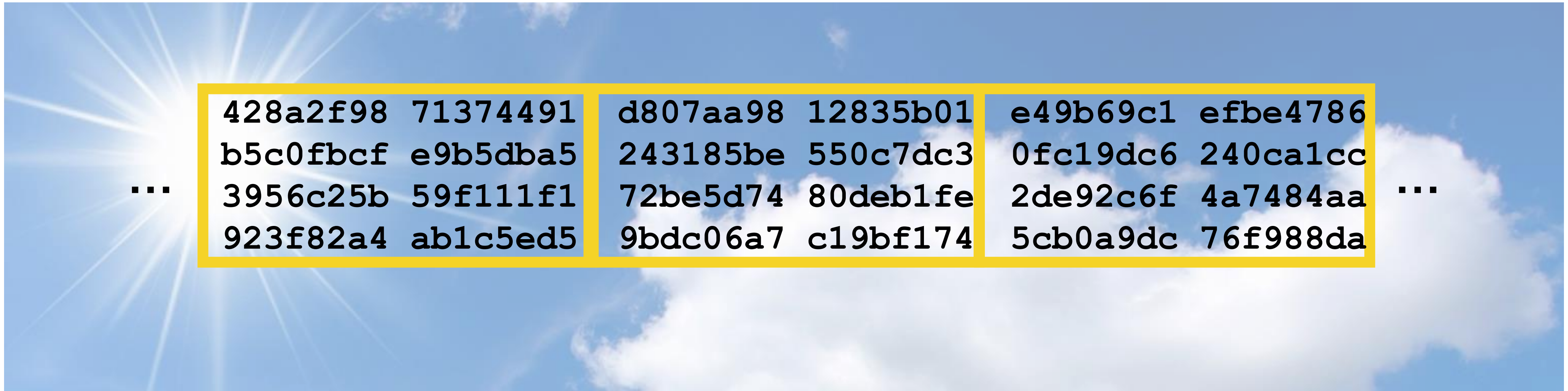
Random Oracle Model (ROM)

- Of course, a *real* hash function doesn't have these ideal properties. (We'll talk about examples.)
- But well designed and properly used, comes close
- The ROM is useful for:
  - Conceptual simplicity;
  - Mathematically rigorous but simple proofs of security;
  - Understanding how to use hash functions.



Random Oracle Model (ROM)

- The ROM implies *preimage resistance*.
- If I give you *randomly selected* image *y* and ask you to find an *x* such that $H(x) = y$, what's the best you can do?
  - If n = 256, expected number of guesses is 2^{256}
- Huge number!
  - 2^{256} is (way) more than the number of atoms in the Earth



In-class exercise

Random Oracle Model (ROM)

- If I ask you to find an x such that $H(x) = y$ ends in 0000 (binary)…
  - How many guesses on average / expectation?
  - Maximum number of guesses?

```
… 428a2f98 71374491 | d807aa98 12835b01 | e49b69c1 efbe4786 …
  b5c0fbcf e9b5dba5 | 243185be 550c7dc3 | 0fc19dc6 240ca1cc
  3956c25b 59f111f1 | 72be5d74 80deb1fe | 2de92c6f 4a7484aa
  923f82a4 ab1c5ed5 | 9bdc06a7 c19bf174 | 5cb0a9dc 76f988da
```

Random Oracle Model (ROM)

From Ron Rivest's MD5 FAQ:

Q. I understand how MD5 works, but I can't figure out how to decrypt the resulting ciphertext. Can you please explain how to decrypt an MD5 output?

A. MD5 is not an encryption algorithm---it is a message digest algorithm. There should be no feasible way to determine the input, given the output. That is one of the required properties of a message digest algorithm.

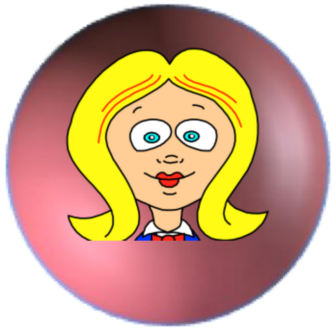
Now for a little digression about birthdays…

Birthday paradox

- There are N (=365) days in an (ordinary) year.
- Suppose there are k people in the room.
- Assume (uniformly) randomly distributed birthdays.
- How large must k be for it to be likely (prob. $\geq 1/2$) that two people share a birthday?

Birthday paradox

- Think of this as a "balls and bins" experiment.

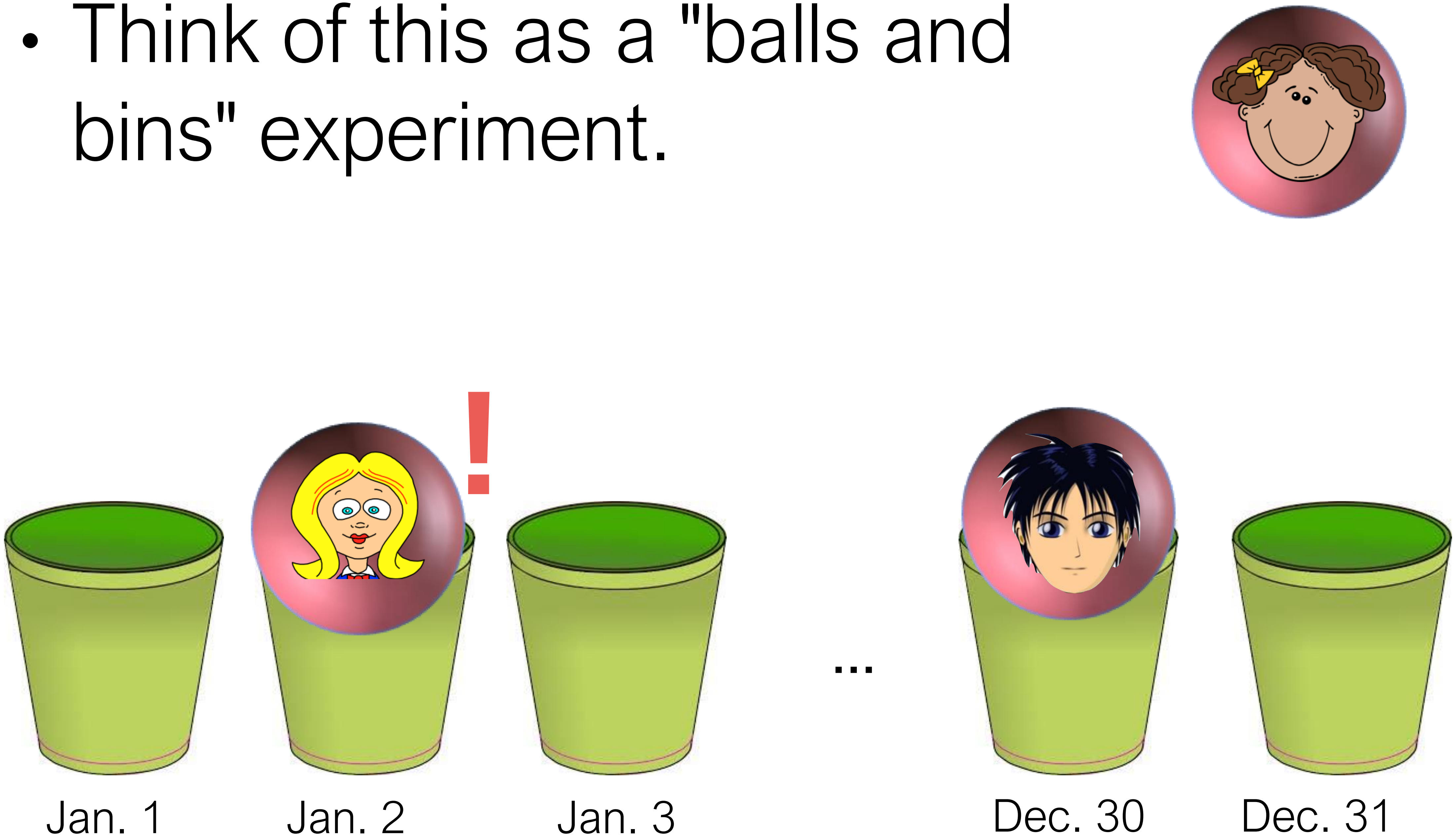


Birthday paradox

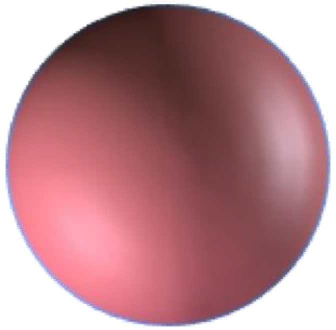
- Think of this as a "balls and bins" experiment.

Birthday paradox

- Think of this as a "balls and bins" experiment.

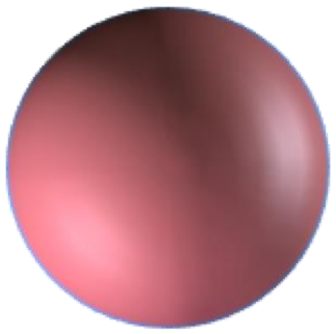


Birthday paradox



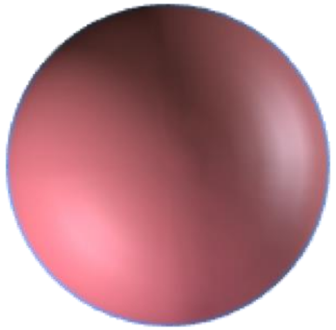
- We're going to throw k balls into N bins
  - i.e., assign k birthdays over a year
- Let X_{ij} denote event that ball j lands in same bin as ball i
  - I.e., $X_{ij} = 1$ if so
  - I.e., $X_{ij} = 0$ if not
- Thus, there's a collision if $X = \Sigma_{i,j} X_{ij} \geq 1$

Birthday paradox

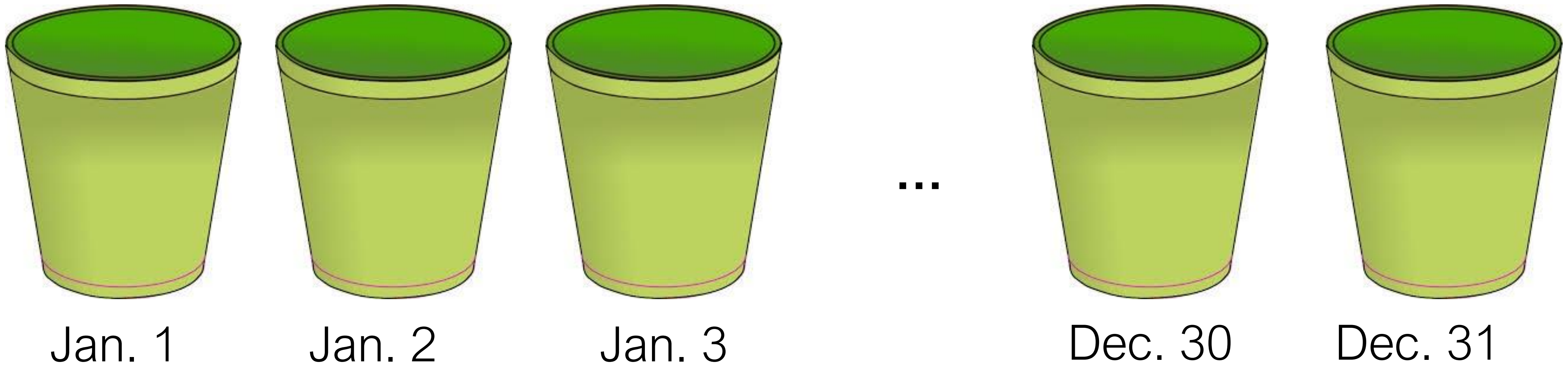


- There's a collision if $X = \Sigma_{i,j}\, X_{ij} \geq 1$
- What's $E[X_{ij}]$?
  - 1/N
- How many distinct (*i*,*j*) pairs of balls?
  - $C(k,2) = k\,(k-1)\,/\,2$

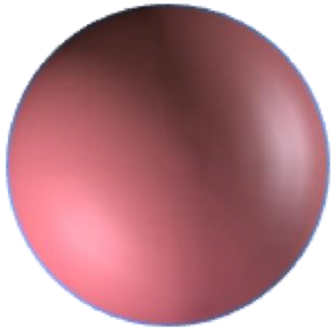
Birthday paradox



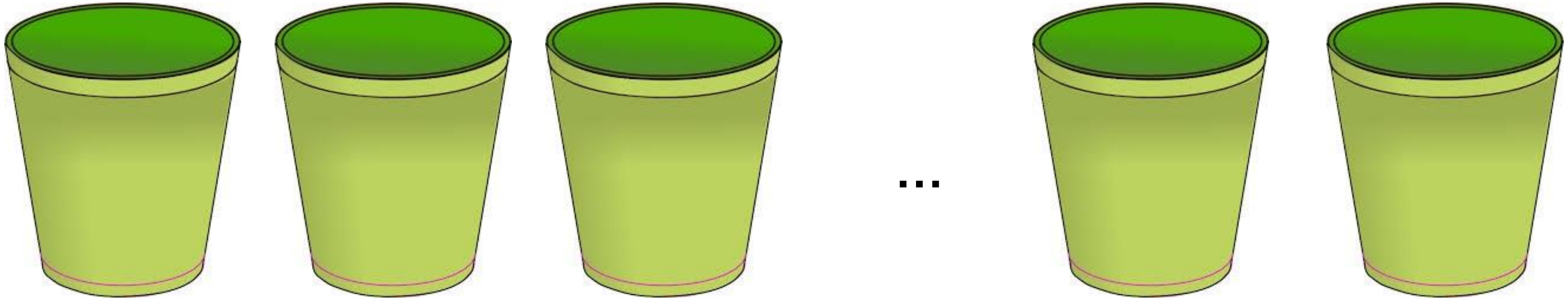
- Thus, $E[X] = C(k,2) / N$ **≈** $k^2 / 2N$
- *Heuristically*, collision w.p. **≈** 1/2 for $E[X]$ **≈** 1/2
  - (At most prob. = 1/2)
- Now, $E[X]$ **≈** 1/2 when
  - $k^2 \approx N$, i.e., $\boldsymbol{k \approx \sqrt{N}}$



Birthday paradox



- There are ***N*** = 365 days in an (ordinary) year
- **√ 365 < 20**
- In fact:
  - Prob ≈ 50.7% for k = 23
- It's pretty much certain that two people in this room share a birthday!
  - For k = 80, ≈ 99.99% chance!
  - (Actually, 41% chance that three people share a birthday!)
- That's the paradox…



Jan. 1 Jan. 2 Jan. 3 … Dec. 30 Dec. 31

Birthday paradox illuminates collision-resistance

- ROM + birthday paradox ⇒ collision-resistance
- How do I find a collision?

```
Do
    • Pick a random x
    • Compute y = H(x) and store it
Until a collision is found
```

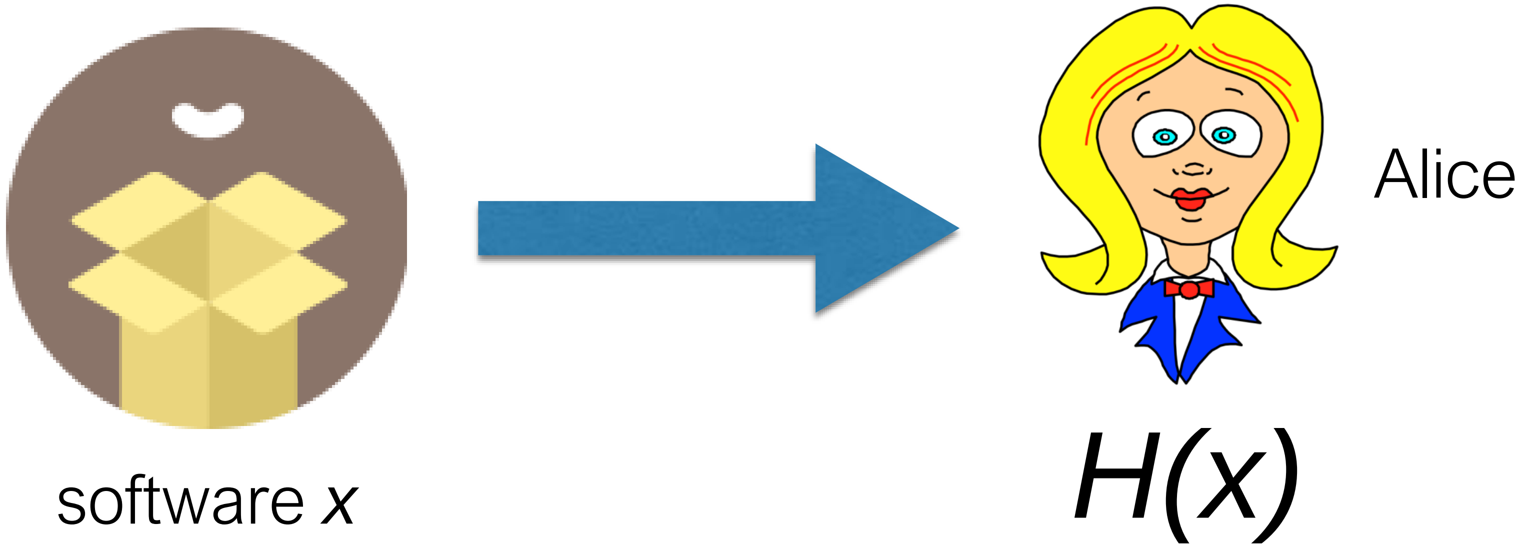
- Given 256-bit hash, like throwing balls into 2^{256} buckets
- By *birthday paradox*, collision w.p. ≈ 1/2 on 2^{128} throws
  - 2^{128} more than, e.g., number of atoms in bodies of all people in NYC
- **General rule of thumb in cryptography:*192-bit security*, meaning 2^{192} work for attacker, is "strong"**

Overall Intuition

- A hash is a **one-way, non-invertible** function of that produces **unique** (with *high likely-hood*), **fixed-size** outputs for different inputs.
- The probability of any bit flipping in the output bit-string should be always ½ for any change (even one bit) in the input (“randomness”).

Applications of hashing

Software verification



Application: software verification

UbuntuHashes

This page contains all of the md5 hashes for the different versions of Ubuntu, including Kubuntu, Edubuntu, Xubuntu and Lubuntu.

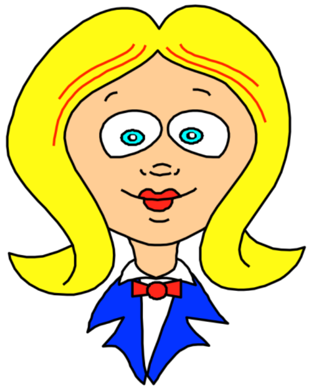
For more information on checking md5 hashes, please refer to HowToMD5SUM. Once you have verified the md5 hash, you may want to refer to the BurningIsoHowto.

Tip: Checking hashes for equality: Type ctrl-F to bring up the find box in your browser. Search for the hash as calculated by the md5sum tool (without spaces or extra characters). Only exact matches will be found.

14.04 LTS

(Trusty Tahr): April 2014 (Supported until April 2019)

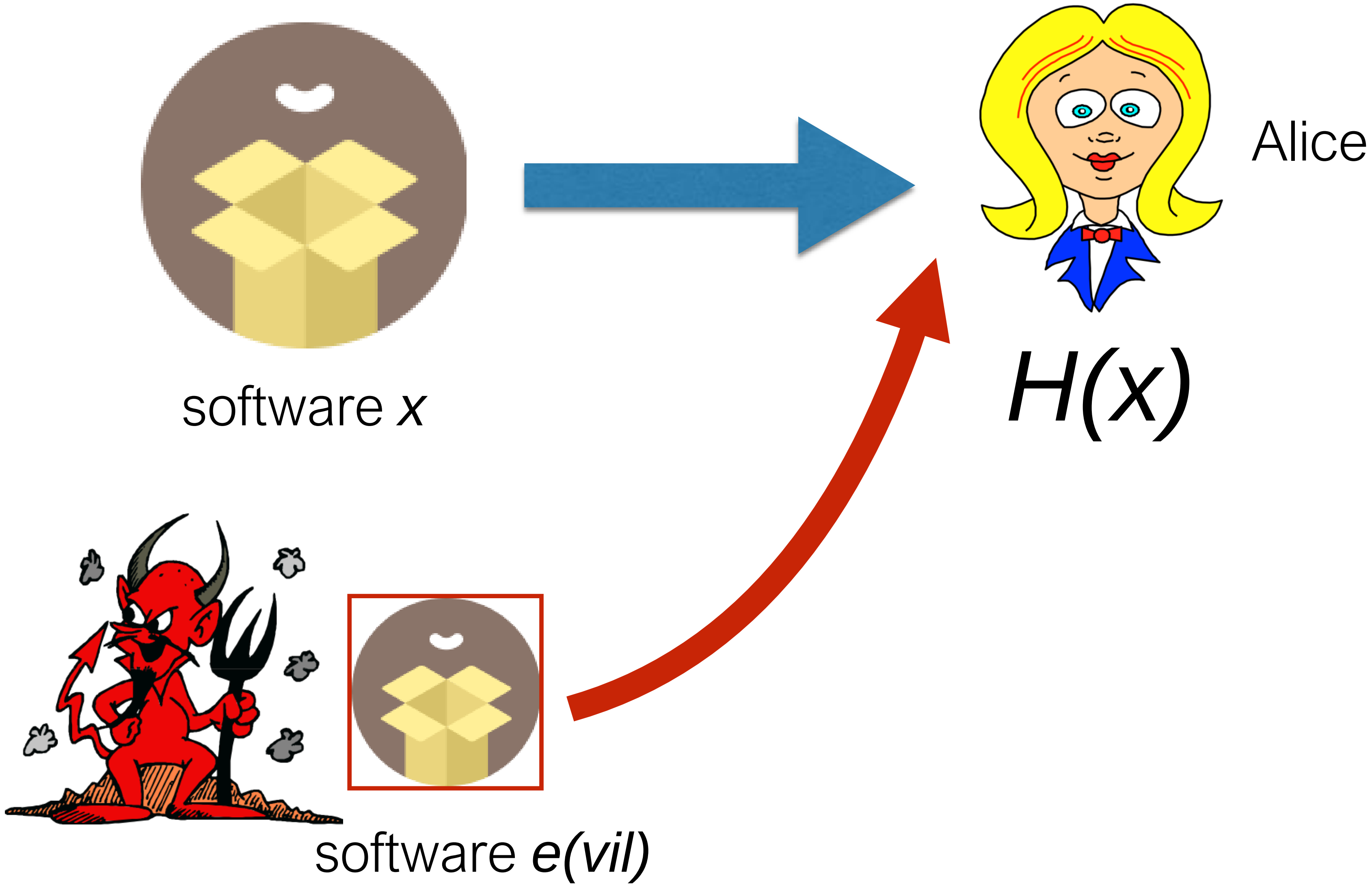
md5 Hash	Version
dccff28314d9ae4ed262cfc6f35e5153	ubuntu-14.04-desktop-amd64.iso
c4d4d037d7d0a05e8f526d18aa25fb5e	ubuntu-14.04-desktop-i386.iso
01545fa976c8367b4f0d59169ac4866c	ubuntu-14.04-server-amd64.iso



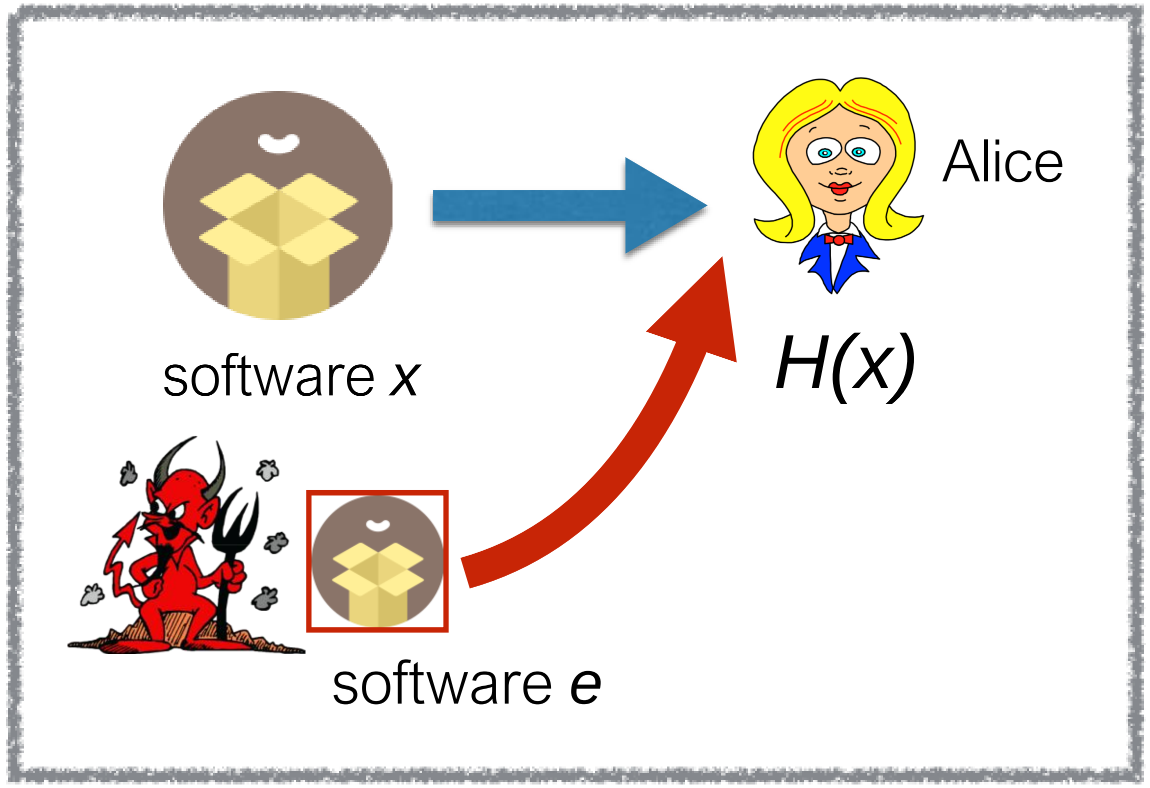
Alice

$H(x)$

Why?

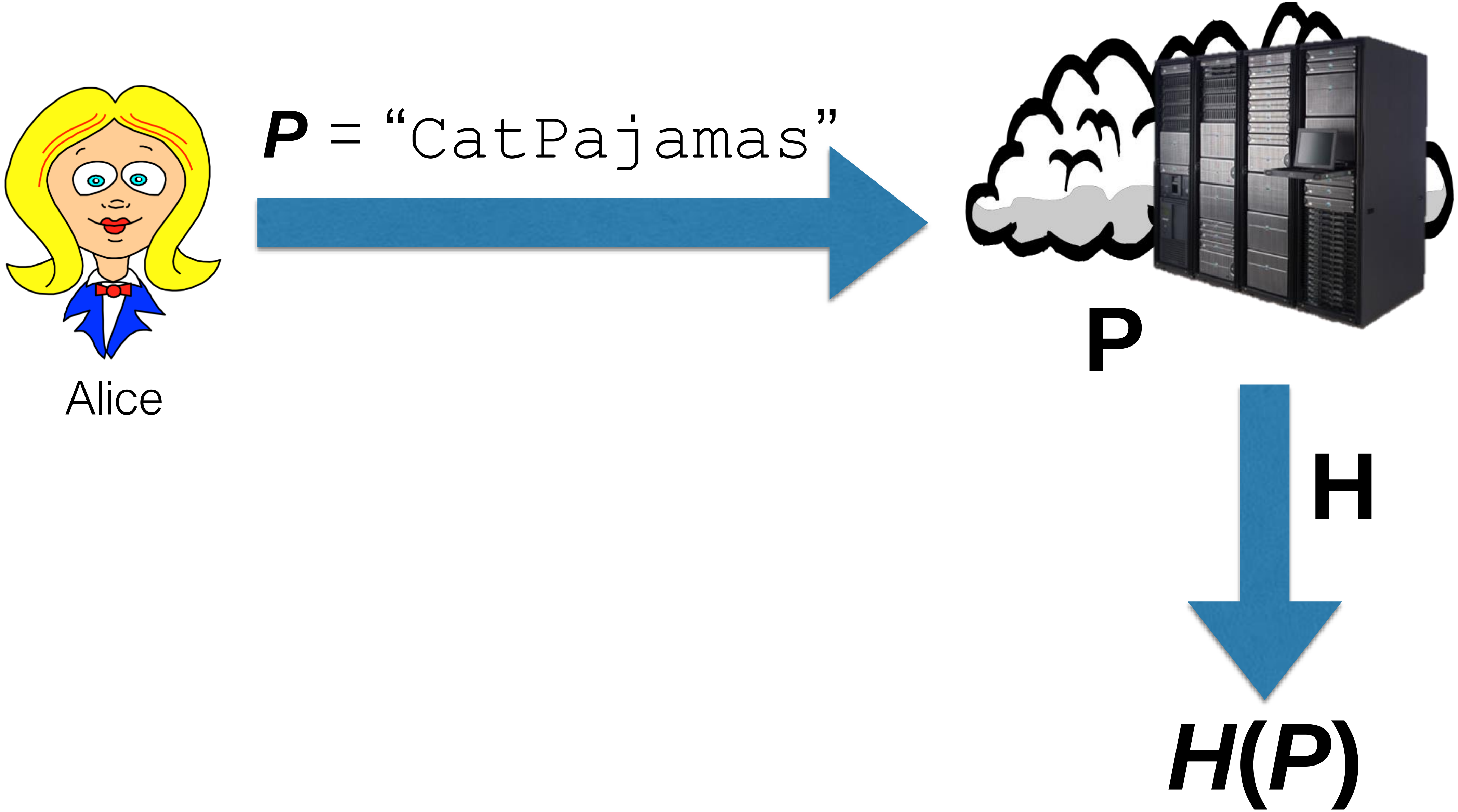


What property of *H* prevents this attack?

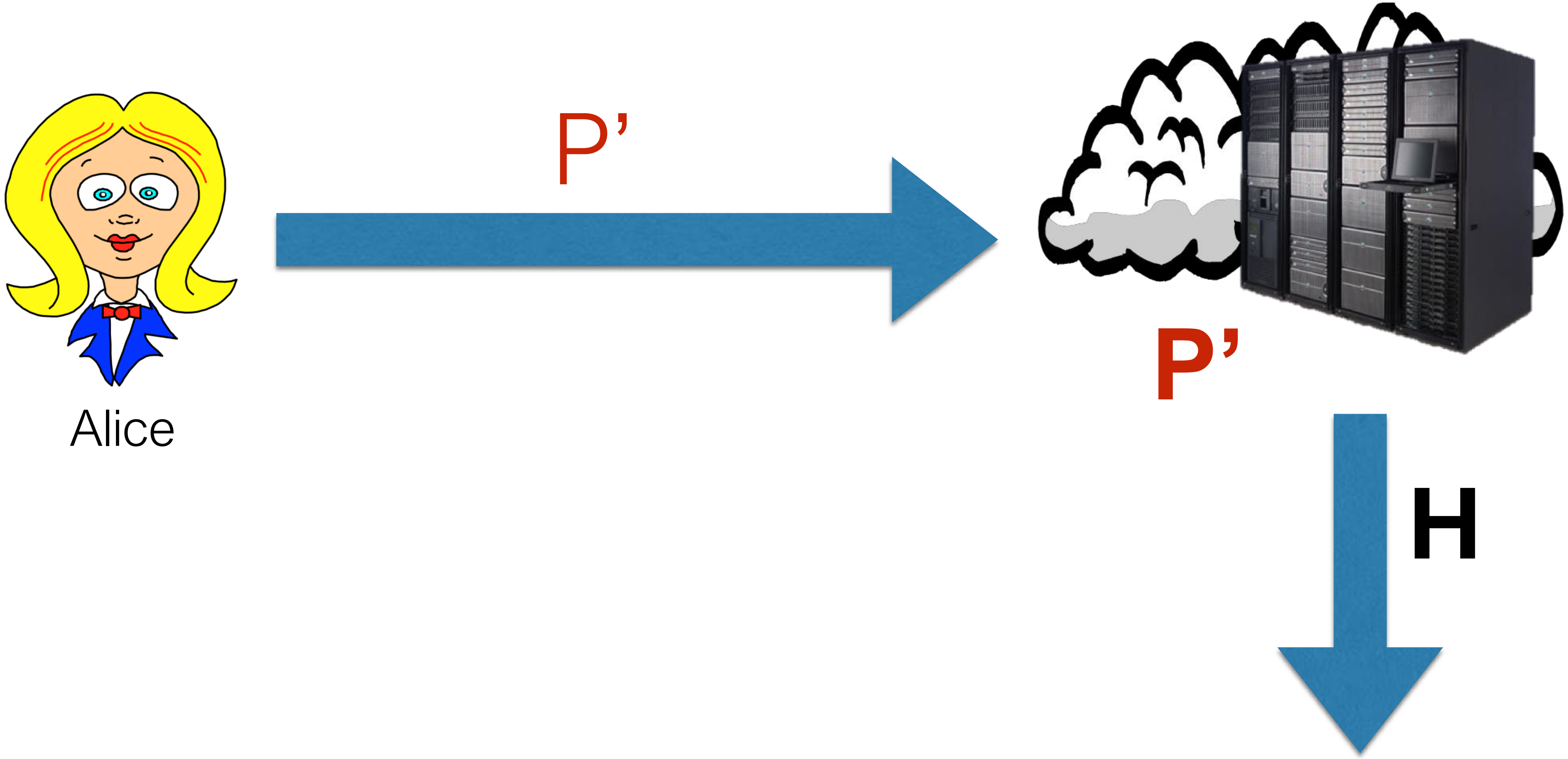


- Case 1: Software vendor / attacker sends *x* to most users, but *e*, with backdoor, to victims.
  - Collision-resistance!
    - Attacker needs to create *x*, *e* such that *H*(*x*) = *H*(*e*).
- Case 2: Evil organization (other than vendor) distributes *e* to victims.
  - Attacker needs to create *e* such that *H*(*e*) = *H*(*x*).
  - Preimage resistance! (In ROM)

Application: Password hashing

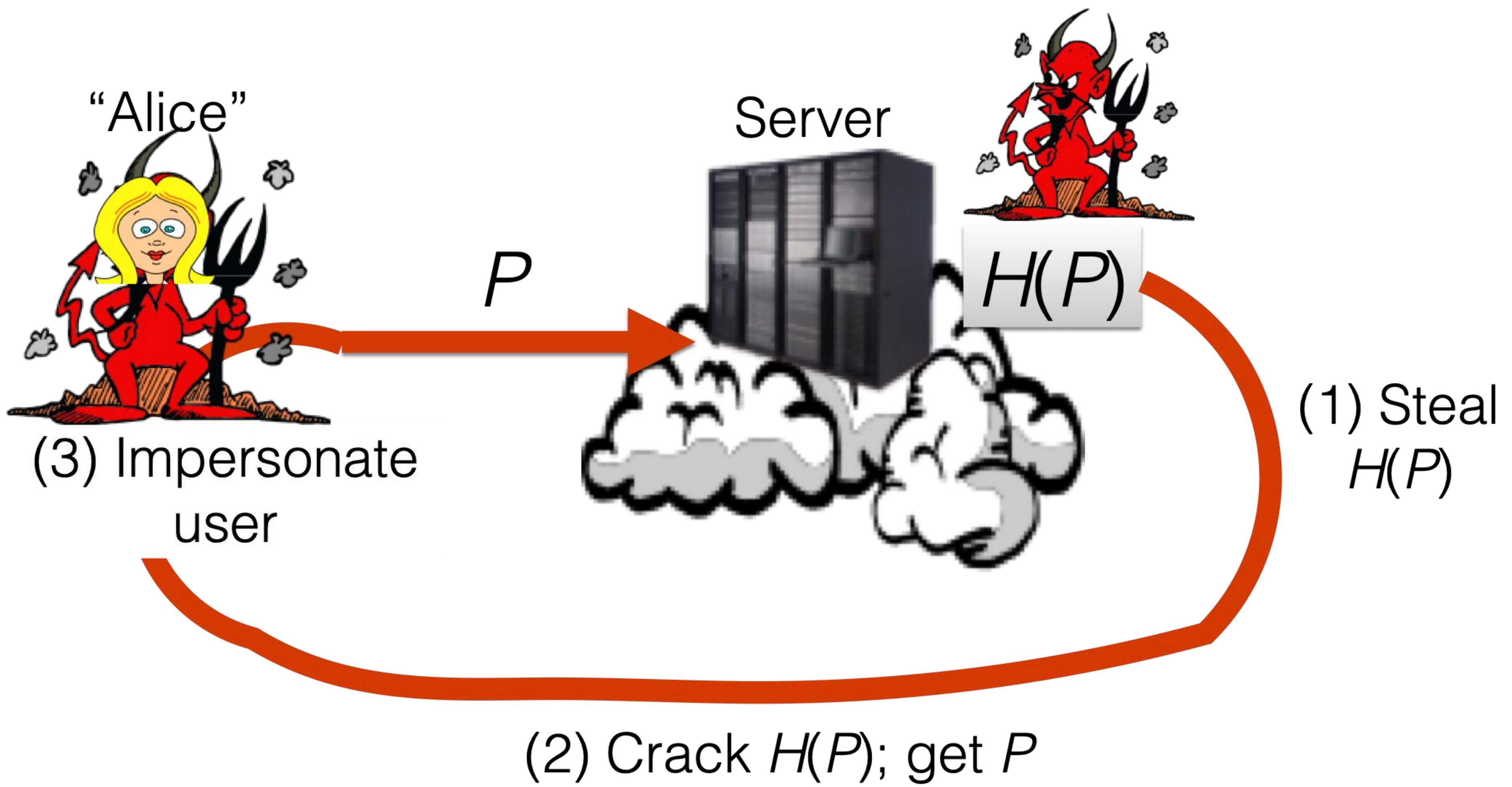


to verify an incoming password…



$$H(P') \stackrel{?}{=} H(P)$$

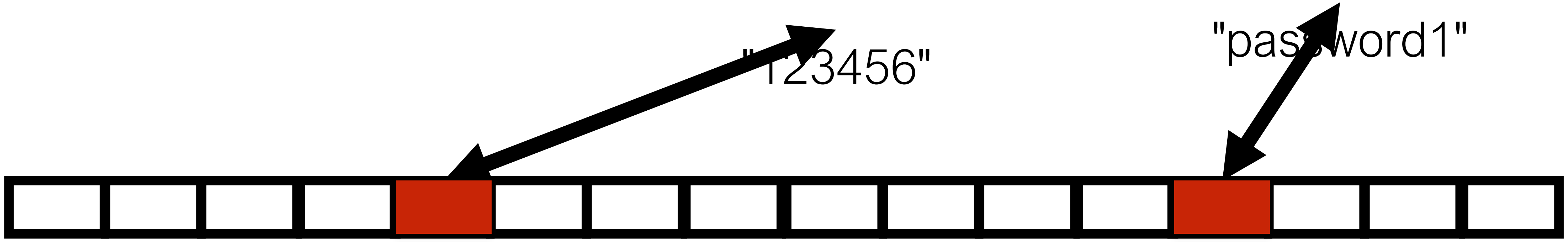
Password attack path



Attacker repeatedly guesses P' until $H(P') = H(P)$ and thus $P' = P$

How can attackers crack hashes?

- Preimage resistance? Isn't *H* hard to invert?
- Yes, but only for *random* digest (ROM tape cell) *y*
- Hash image *H*(*P*) of common password *P* isn't randomly generated!
- Attacker can search space of such hashes
  - *H*("123456"), *H*("password1"), etc.



Worse still…

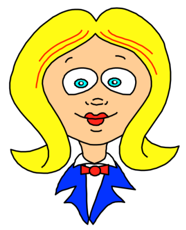
If everyone uses *same* hash function H, then attacker:

- Compiles dictionary of common password / hash pairs (P, $H(P)$)
- Given $H(P)$, looks up P in the dictionary!

Password	Hash
123456	d807aa98 12835b01 243185be 550c7dc3…
password1	72be5d74 80deb1fe 9bdc06a7 c19bf174…

Password-Cracking Dictionary

$H(P)$ = d807aa98 12835b01 243185be 550c7dc3…



Alice's account

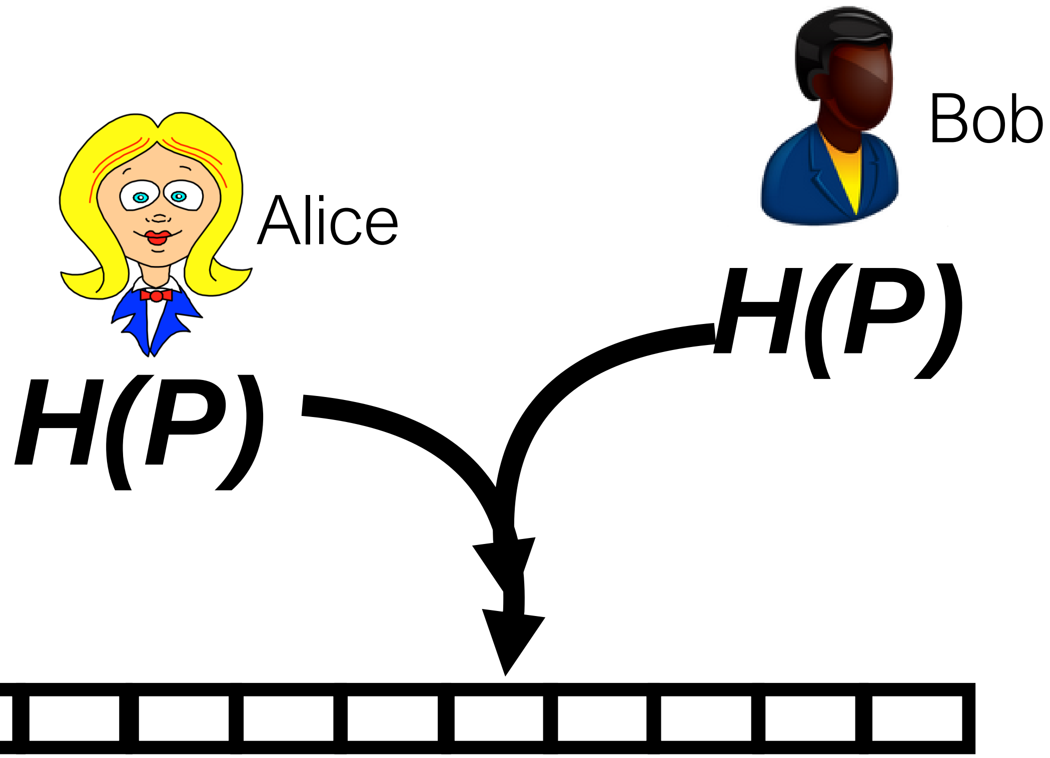
Salting password hashes

- Can we somehow use different hash function for every user?
- Idea: Hash with unique per-user bitstring called ***salt***
- Server stores not $H(P)$ for Alice but $(salt_{\text{Alice}}, H(salt_{\text{Alice}} \,||\, P))$
- Approximates different hash for each user
- Salt not secret!
  - Otherwise, can't verify passwords

Salting password hashes

In the ROM…

without salt.

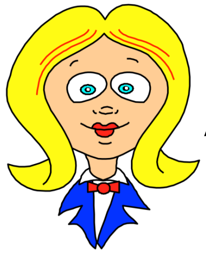


Same attack dictionary works across different users

Salting password hashes

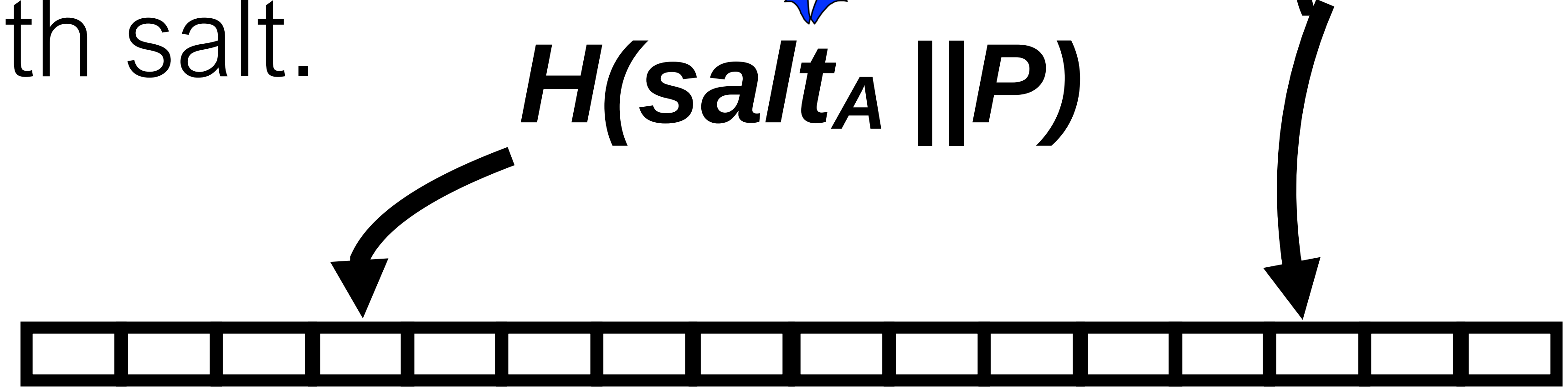
In the ROM…

$H(salt_B \,||P)$



with salt.

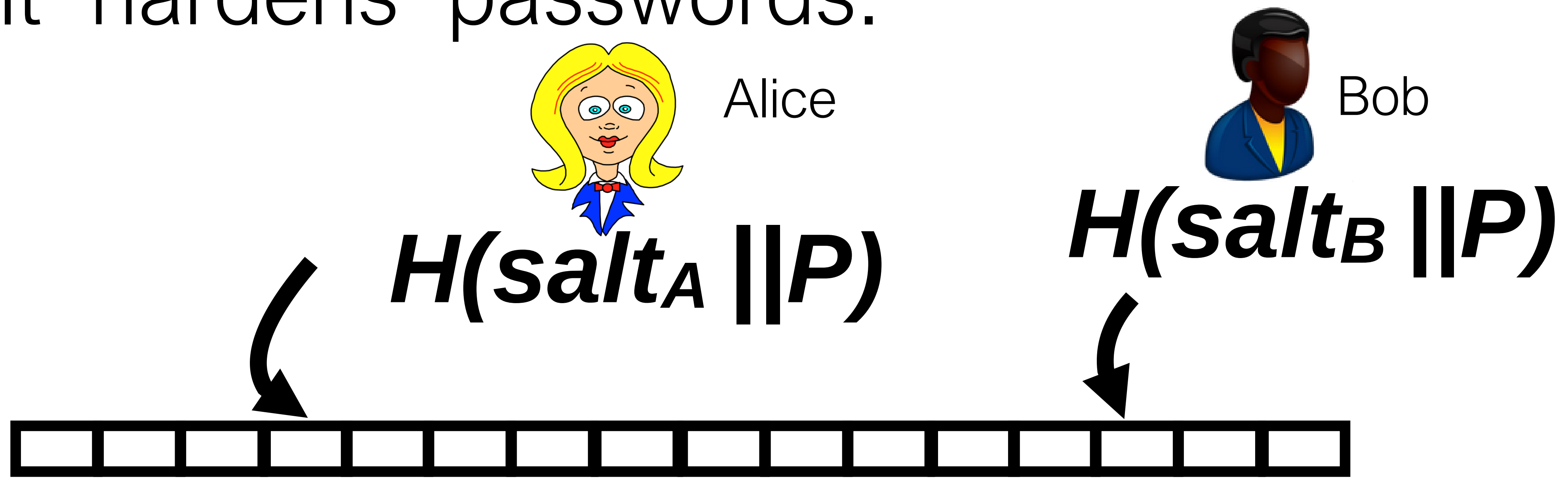
$H(salt_A \,||P)$



Different, independent parts of tape → no shared dictionary

Salting password hashes

- Now, dictionary attack no longer possible!
- Attacker must do online, brute-force guessing.
- Salt "hardens" passwords.



Different, independent parts of tape → no shared dictionary

salted-password database

```
SELECT Username, PasswordHash, Salt FROM dbo.[User]
```

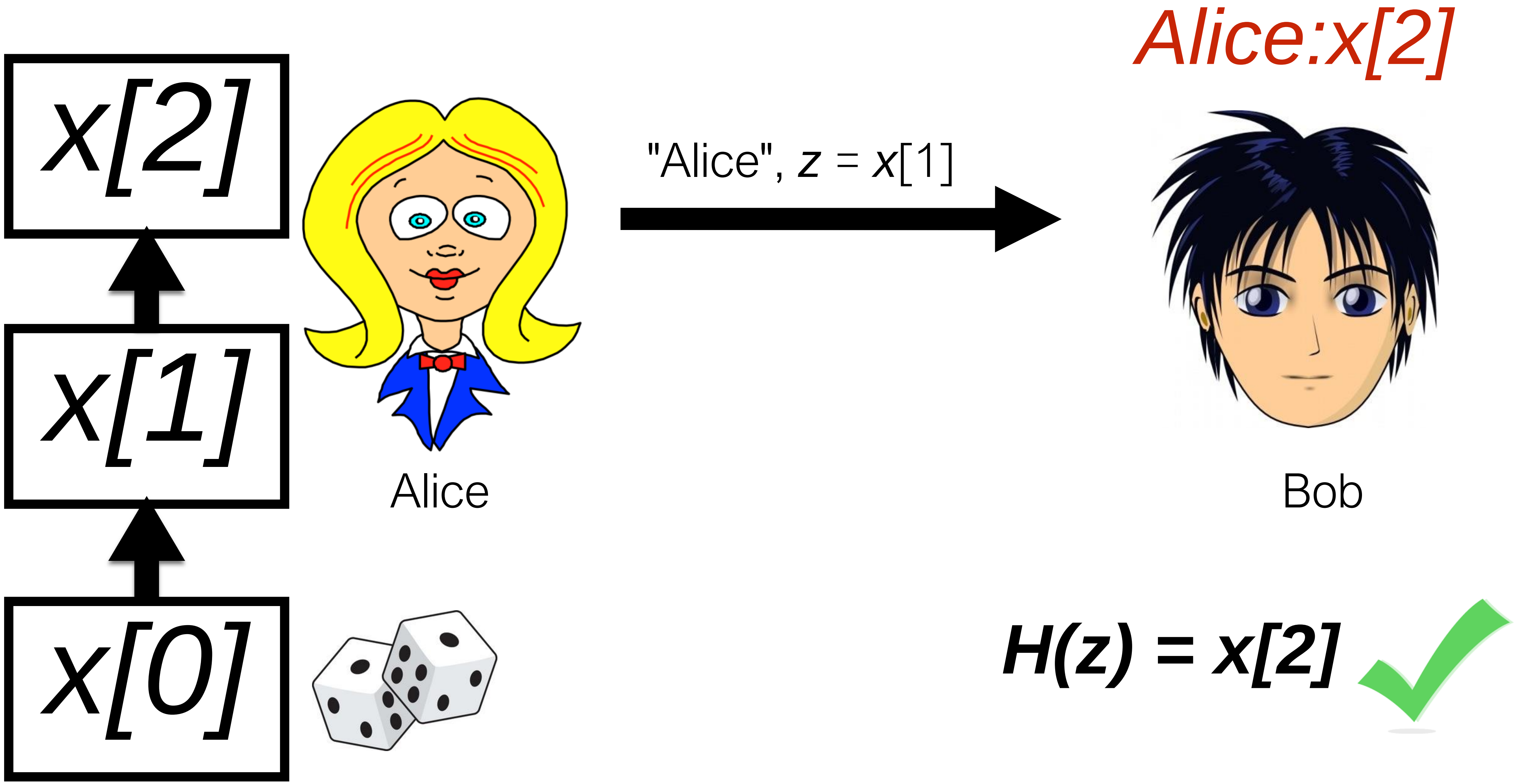
Results | Messages

	Username	PasswordHash	Salt
1	User1	104f4807e28e401c1b9e1c43ac80bdde	nkV38+/eHsI=
2	User2	827e877ba7fa4676ee4903f2b60de13a	NwHowZ63RVw=
3	User3	e901b26b3ec928db2753150d04736c44	Z8uDOfE90gE=
4	User4	72997d54dbe748964c64656cba01e1c8	SKXPm84F2bU=
5	User5	9207f5635d2622e94e2a67b0190c89a8	ppjsgG33riI=
6	User6	07168a0e6f3102a6ee3df50f3355d49c	vINYqVBbtPU=
7	User7	d78c6606bed3d2e4262df59b29e0bfc2	pQQdD514I/E=
8	User8	c71dcf5a4be211294014537c255ac48a	v+x3ypPTCig=
9	User9	2ad3269ee1f97858f7ff236a02b3a32e	SOwixgcWgvA=
10	User10	bb0ae47e5b95b896568bc014ac63b9c1	+Bz6pI/G6DQ=
11	User11	b72c7ec38b64ca39fee15a931f3f5260	UDfOA0DyQQQ=
12	User12	2e658552d8fe83fcd7820bff7fb2cee7	fvhDCo17aAk=
13	User13	c5cef9d547088594e022a6581bc44ea6	YaDJlrHZMnk=
14	User14	ab9a873186c52d0daf11c8a193dc6f9c	8cLo46CTPUE=
15	User15	30027afd712c3cc235459a0f1a45bea5	bLSAogm+RT4=
16	User16	50e195fd7f0d53dc0072e56e54f17f50	7yBcpKnRkpc=

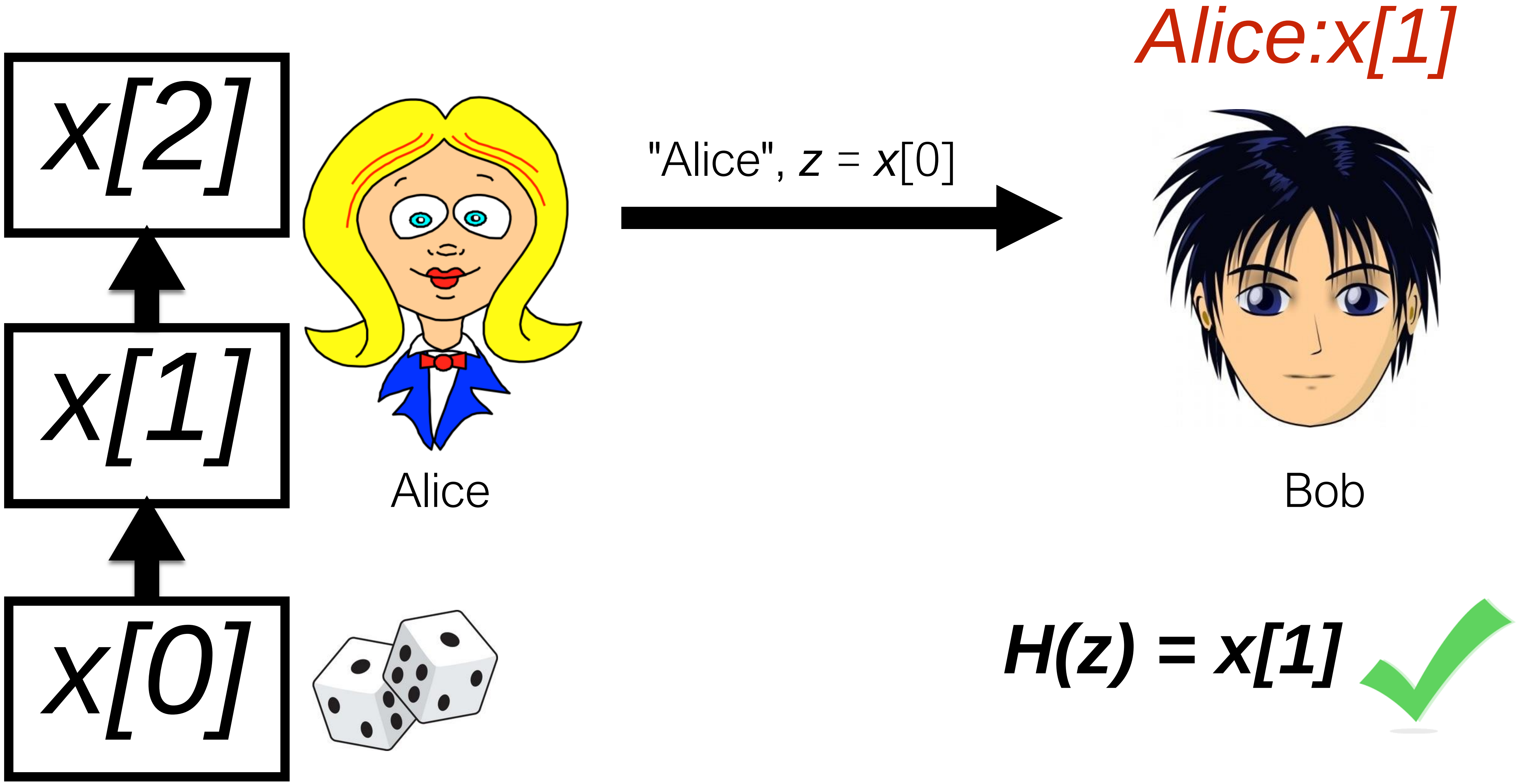
Another defense: Resource-intensive hashing

- Idea: Make *H* slow (but feasible) to compute
- Approach: Many iterations of *H* to slow process
  - E.g., store $H^{2048}(P)$
  - Computationally intensive
    - Pro: slows attacker
    - Con: slows user
  - Example: bcrypt, default in BSD (based on Blowfish cipher)
- Newer approach: Heavy use of fast memory (cache)
  - Examples: scrypt, Argon2
  - Used today in cryptocurrencies (e.g., Litecoin, Ethereum)

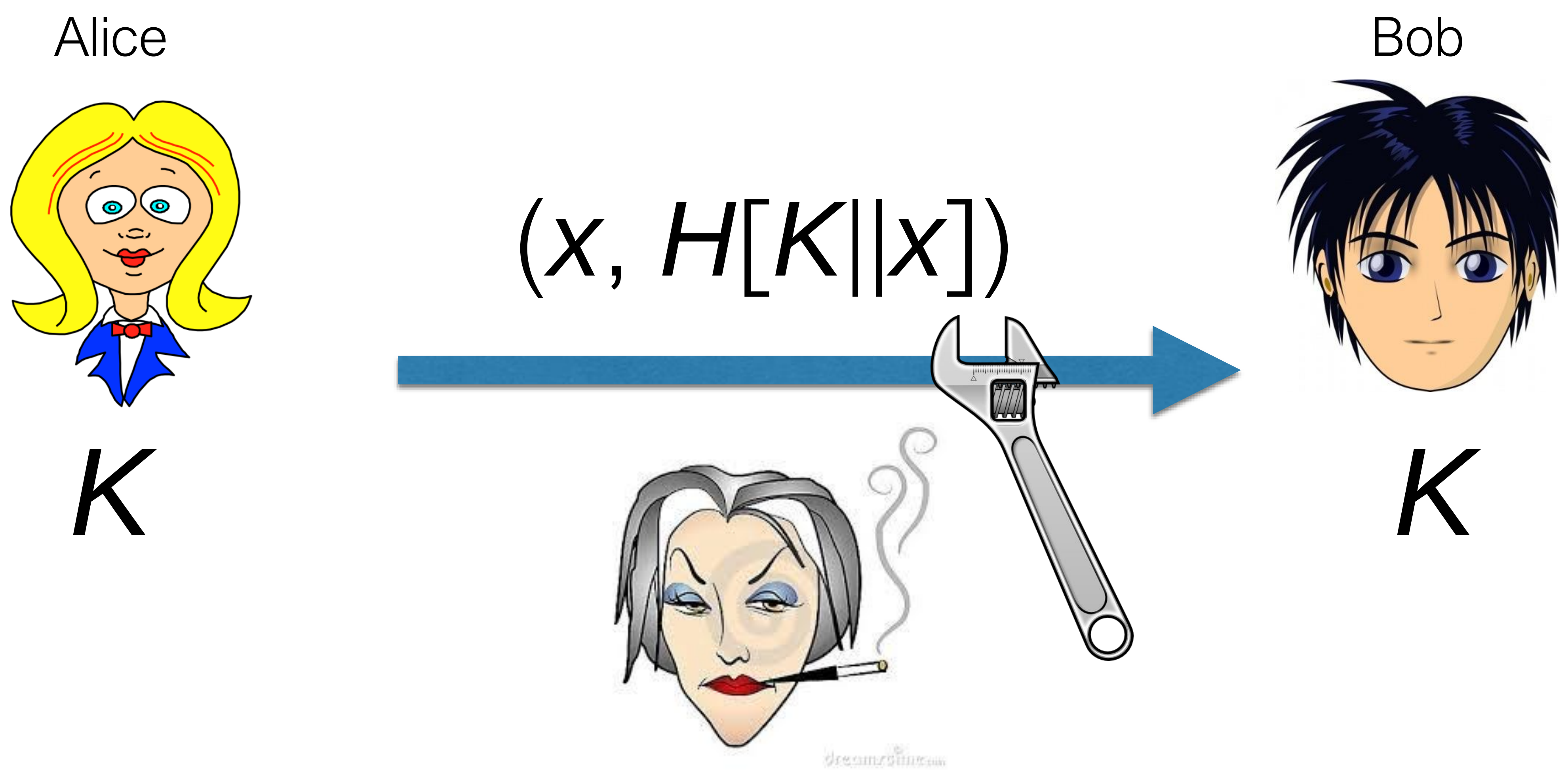
Hashing application: User authentication (S/KEY)



Hashing application: User authentication (S/KEY)



Message-Authentication Code
(MAC)
(Naive)



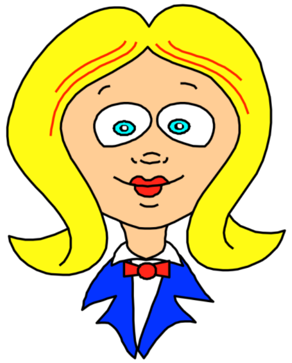
Commitment

Commitment

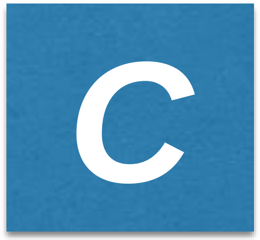
- Suppose:
  - Alice chooses short message m
    - E.g., $m \in \{0,1\}$, i.e., one bit
  - Alice gives us $C = H(m)$
- Easy to compute m from $C = H(m)$
  - Like brute-force password cracking
  - $C = H(0)$ or $H(1)$?
- Can Alice somehow use H to *hide* m?

Commitment

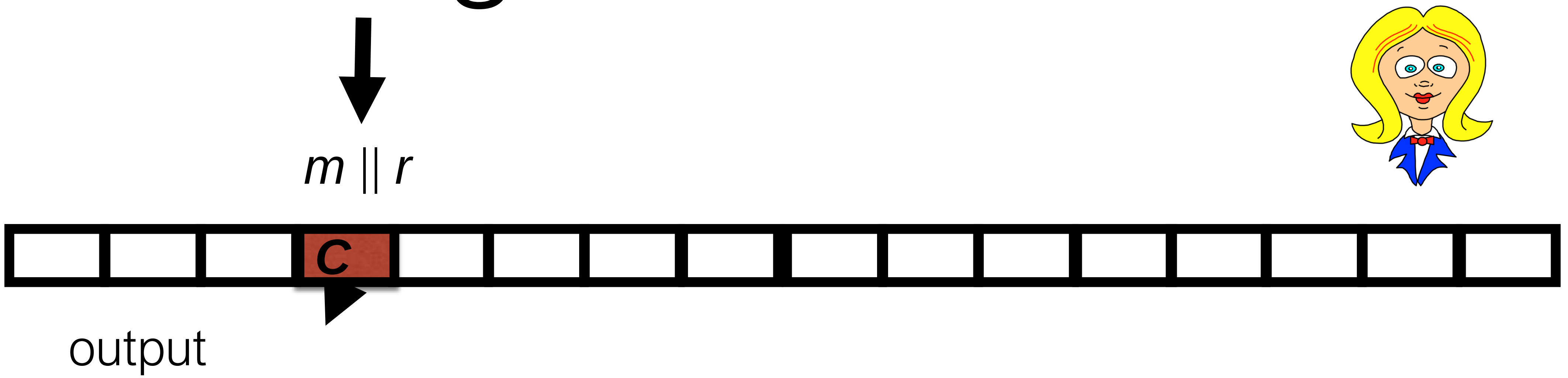
- Alice chooses random, secret key r
  - E.g., $r \Leftarrow \{0,1\}^{128}$
- Suppose Alice gives us $C = H(m \mid\mid r)$…
  - …but not r
- Now hard to compute m!
- Why?



Alice



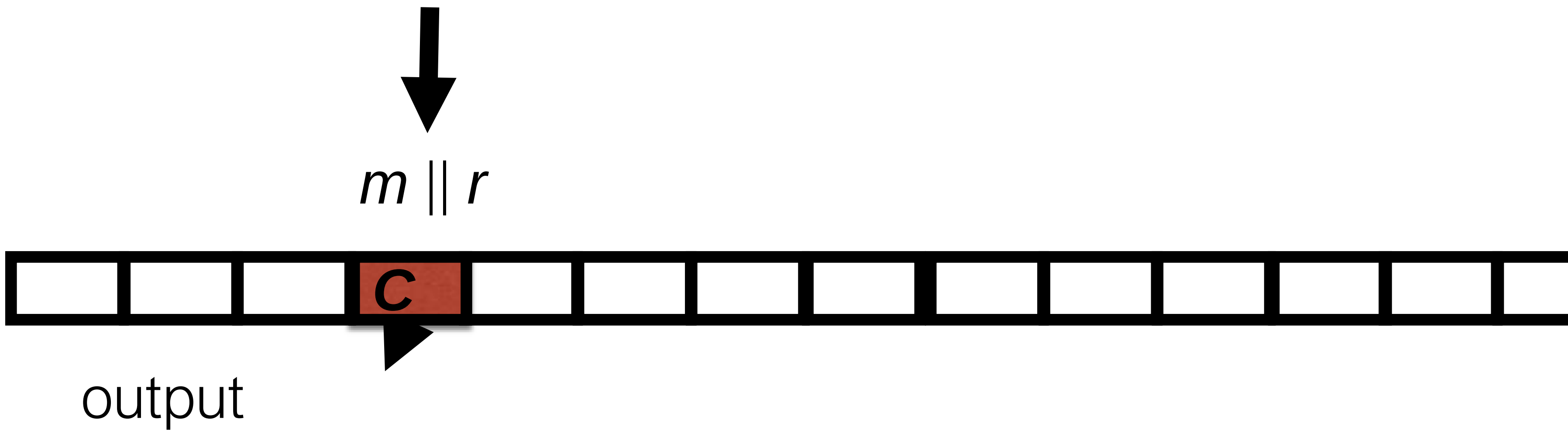
Hide and go seek in the ROM

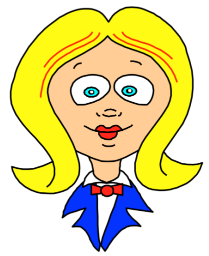


C

- Alice computes *C* by querying cell for *m* || *r*
  - Call this the "red cell"
- To confirm *m*, need to **find the red cell** in the tape
  - Can't tell if cell is red unless queried!

Hide and go seek in the ROM

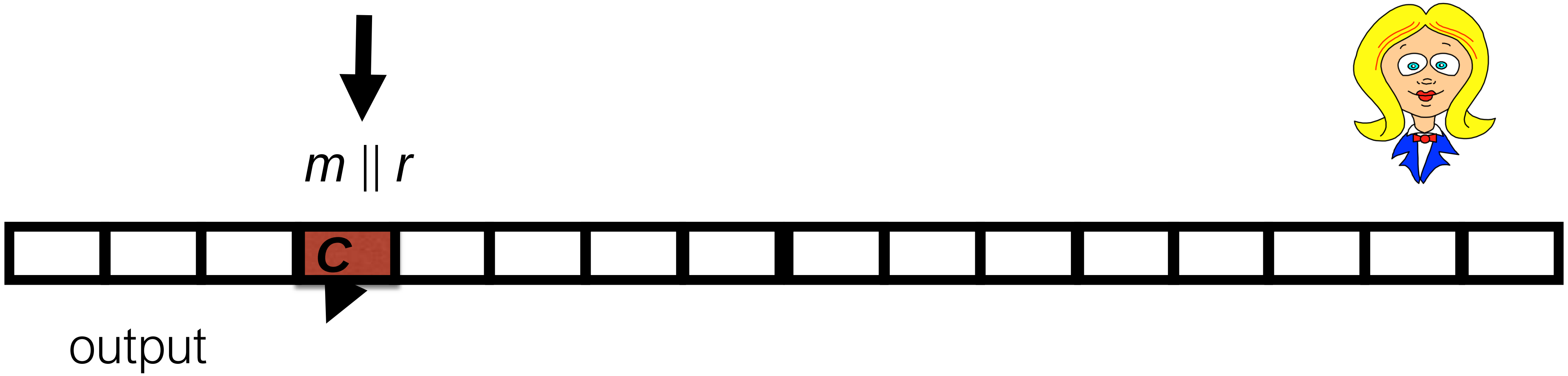




c

- But there are many, many possible values (2^{128}) of *r* !
  - So many candidate red cells
  - Far too many candidates to search exhaustively!
- Commitment is (computationally) *hiding*

But…



C

If Alice reveals ("decommits") *m*, *r*, we can verify commitment by checking:

$$C = H(m \mid\mid r)$$

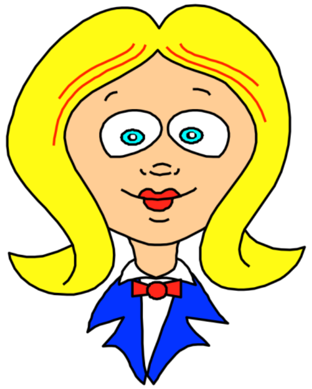
What if Alice changes her mind / cheats?

- Alice commits to $m = 1$
  - i.e., computes $C = H(1 \,||\, r)$
  - Gives us C
- Alice wants to decommit $m = 0$
  - i.e., give us 0, s such that $C = H(0 \,||\, s)$
- Infeasible!
- Why?
  - Breaks collision resistance of H!
  - Alice must find $H(1 \,||\, r) = H(0 \,||\, s)$

A good commitment scheme is…

- *Efficient*: Easy to compute C
- *Hiding*: Hard to compute m from commitment C
- *Binding*: Hard to change m for given commitment C
  - I.e., hard to decommit some $m' \neq m$

Commitment application: Time capsule



Alice

m = "Alice is Satoshi Nakamoto"

random secret key r

$C = H(m \,||\, r)$

Year 2007

Commitment application: Time capsule

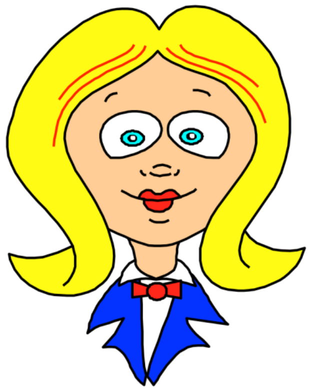
Alice

2008

Who is Satoshi Nakamoto?

Years 2008-17

Application: Time capsule



Alice

Year 2017

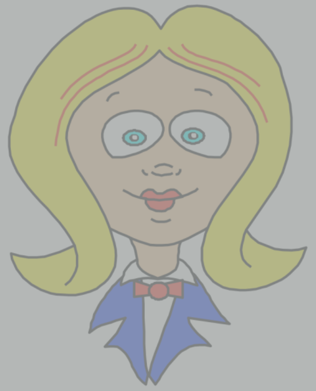
***m* = "Alice is Satoshi Nakamoto"**

random secret key *r*

m, r

$C = H(m \mid\mid r)$

A



Ali

m = “Alice is Satoshi Nakamot

random secret key r

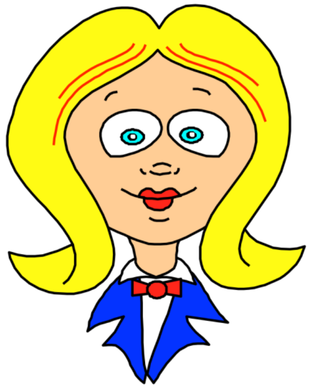
New evidence from 2007!

C

$H(m \,||\, r)$

Commitment application: Time capsule

- Why does it matter that:
  - Commitment is *hiding*?
  - Commitment is *binding*?



Alice

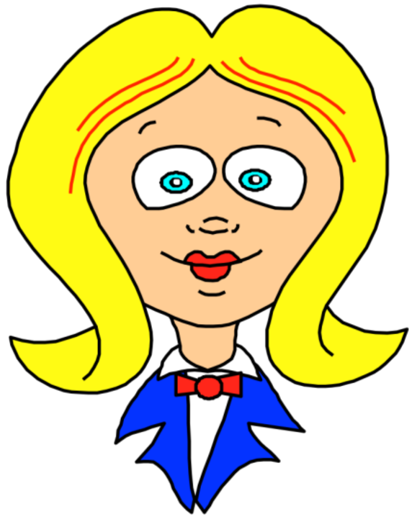
Commitment application: Fair coin toss… over the telephone

Alice

Bob

A bad way to do it…

Commitment application: Fair coin toss… over the telephone



Alice

C = H("heads" || r)

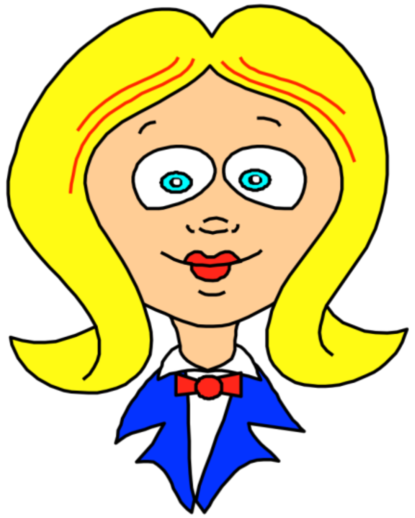
"heads"!

"heads", r

Bob

C = H("heads" || r)

Now the game is exactly like that played in person



Alice

Alice flips coin and covers with her hand

Bob guesses "heads"!

Alice lifts her hand

Bob

Other applications of hashing

- Digital signatures
  - “Compress” message to reduce signing computation
- Tamper-prevention
  - E.g., International Criminal Tribunal for Rwanda evidence
- Bitcoin
  - “Proof of Work” involves use of hash function (SHA-256)
- Many, many other uses

Real hash functions (and a caveat)

Some common hash functions

- MD5
  - Highly influential design by Ron Rivest in 1991
  - Strong attack against collision-resistance shown in 2004 (Wang and Yu)
  - Attack exploited in the wild in Flame malware in 2012
    - Used to create rogue Microsoft certificate because…
  - Still in common use through 2012!

Example MD5 Digest

md5_digest("The quick brown fox jumps over the lazy dog") =
9e107d9d372bb6826bd81d3542a419d6

md5_digest("The quick brown fox jumps over the lazy cog") =
1055d3e698d289f2af8663725127bd4b

Some common hash functions

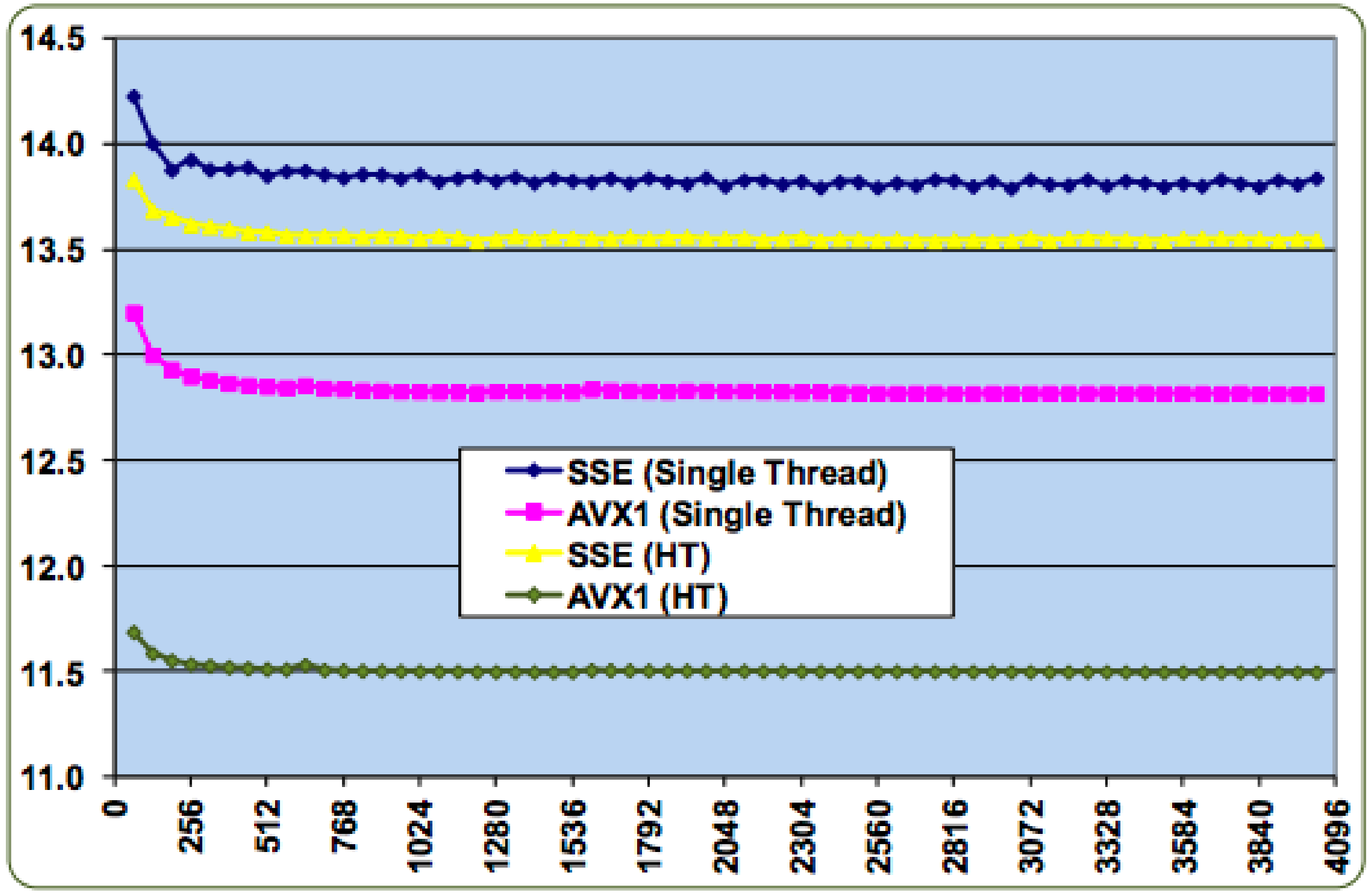
- SHA-1
  - Designed by NIST / NSA; Influenced by MD5
  - Some theoretical weaknesses shown in 2005 by Wang, Yin, and Yu
  - First collision demonstrated on 23 Feb. 2017 by Google / CWI
    - Nine quintillion (9,223,372,036,854,775,808) SHA1 computations in total
    - 6,500 years of CPU computation to complete the attack first phase
    - 110 years of GPU computation to complete the second phase
  - Still in pretty common use!

Some common hash functions

- SHA-2 family
  - In common use (e.g., SHA-256 used to authenticate Debian GNU/Linux software packages, in Bitcoin, etc.)
  - Includes: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256
  - Designed by NSA
  - Published in 2001

Performance with Intel SSE instruction

Figure 2: Fast SHA-256 Performance in Cycles/byte as a function of Buffer size (bytes)[2]



At the time of writing this paper, there are no widely available processors that support the rorx instruction.

SHA-3

- Public competition organized by NIST in 2007 to develop new cryptographic hash algorithm
- 64 entrants (Oct. 2008)
- 5 finalists (Dec. 2010)
- Winner: Keccak (Oct. 2012)
  - Bertoni *et al*.
- Standardized by NIST as SHA-3
- SHA-3 is a "backup" algorithm
  - No known weaknesses of SHA-2, e.g., length-extension attacks
  - Different design principle (“sponge”) than Merkle-Damgård
  - Evidently useable for commitment directly, without HMAC

Hashes to (not) use

- Do not use at all the following:
  - **MD5, SHA-0/1, any other obscure “secret” ones**
- For use in civilian/.com setting (until 2025):
  - **SHA-256/512, SHA3**

Hashing for passwords

- Argon2
  - Winner of Password Hashing Competition in 2015
  - One of a number of ***memory-hard*** hash functions
  - Why memory-hardness?
    - General-purpose vs. special-purpose hashing hardware
- Balloon hashing
  - Like Argon2, but with memory-hardness that's ***proven*** in the ROM

Takeaways

Hash functions are powerful!

- Ideal model: Random Oracle Model
  - “tape in the sky”
- Well-known hash function: SHA-256 (SHA-2 family)
- Applications we saw today
  - Password protection
  - File integrity
  - User authentication
  - Coin flipping over the phone
- Lots of other applications!