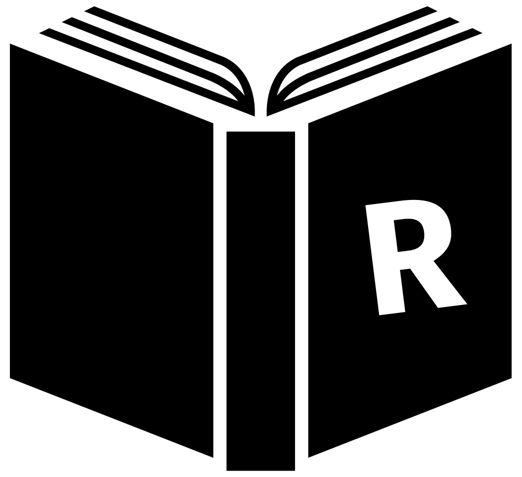
Lecture 18: (Password based) hashing, continued

- Lab 10 has been posted, due *Wednesday 4/24* at 11pm
- Lab 11 will be posted Tuesday 4/23 and due Wednesday 5/1
- Reminder: my office hours have moved to Thursdays at 11am-1pm

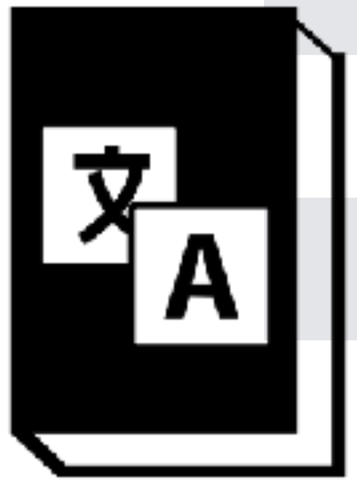
Hash function = 1 public codebook

- Hash function $H : \{0,1\}^\infty \to \{0,1\}^{out}$
- Compresses long messages into short digests
- Most popular example in use today: SHA-256
- *Random oracle* is an ideal public codebook



- Concrete hash functions must provide:
  - Preimage resistance
  - Second preimage resistance
  - Collision resistance

X	*Y*
aba	nr
abs	mb
ace	yd
act	wv
add	je
ado	hg
aft	uv
age	zm
ago	ds
aha	ae
aid	kf
⋮	⋮
zip	cy
zoo	dx



Password-based key derivation function

- Threat we are trying to mitigate: a well-funded attacker who either
  - Brute forces the (not too large) password space
  - Obtains your personal phone or organization's `/etc/passwd` file
- Mantra: generate key on the fly, don't write it down anywhere

PBKDF2: Password → Cryptographic key

```
pbkdf2(string password, string salt, int count):
    string key = ‘’
    U₀ = S
    for(j = 1 to count):
        Uⱼ = prf(password, Uⱼ₋₁)
        key = key ⊕ Uⱼ
    return key
```

simplified version with output length == 1 block

use any block cipher or MAC

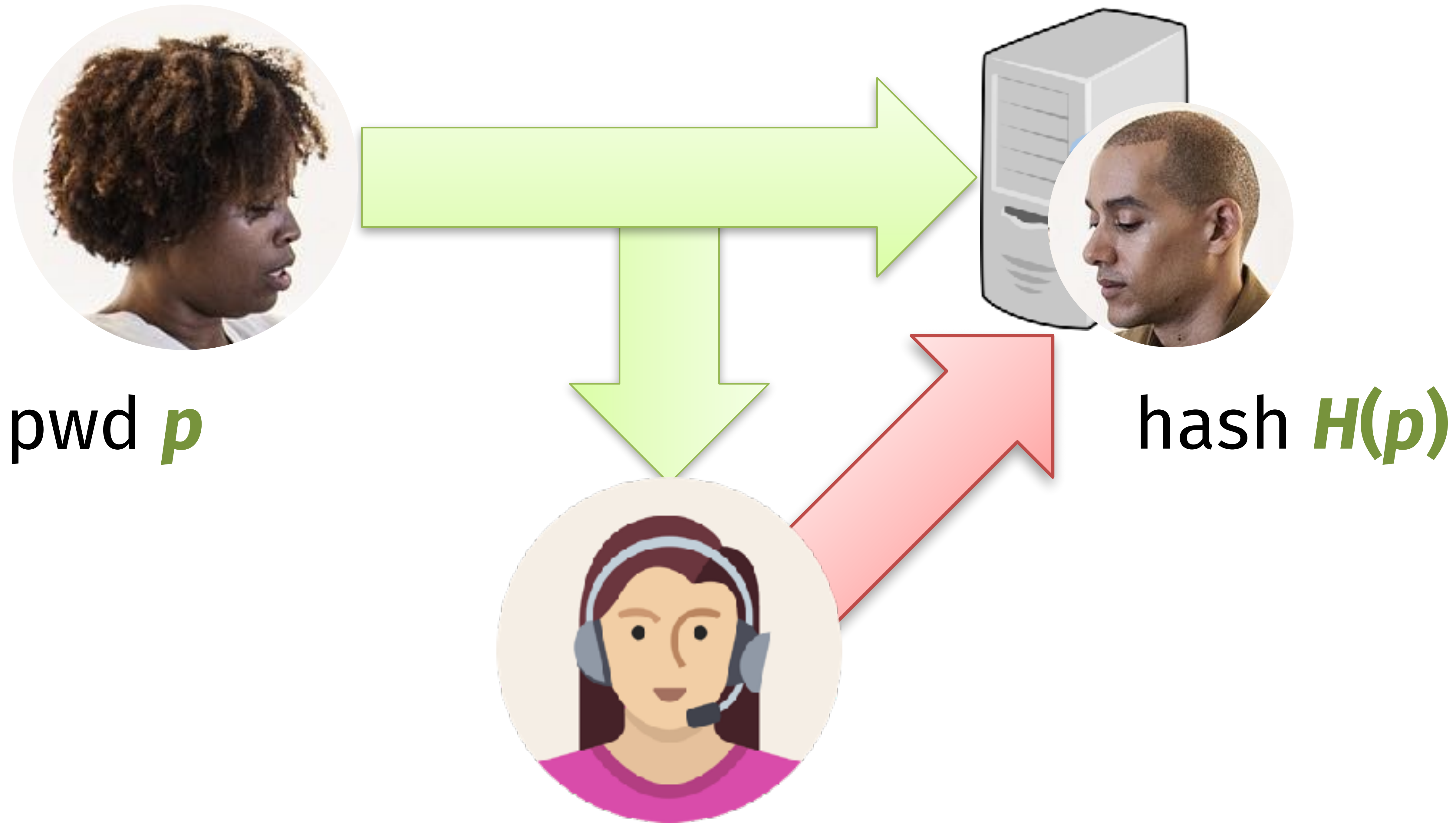
long runtime, and steps are sequential

Why output a crypto key?

- We could have simply built a function that maps password $\rightarrow$ boolean that indicates whether the password is correct or not
- But shared knowledge of a cryptographic key allows you to perform future crypto operations, such as protecting customers' data on your site so only the legitimate client can decrypt it later
- 1-bit checks of claimed_pwd == stored_pwd are more vulnerable: there exist side channels to learn or even directly flip this boolean value

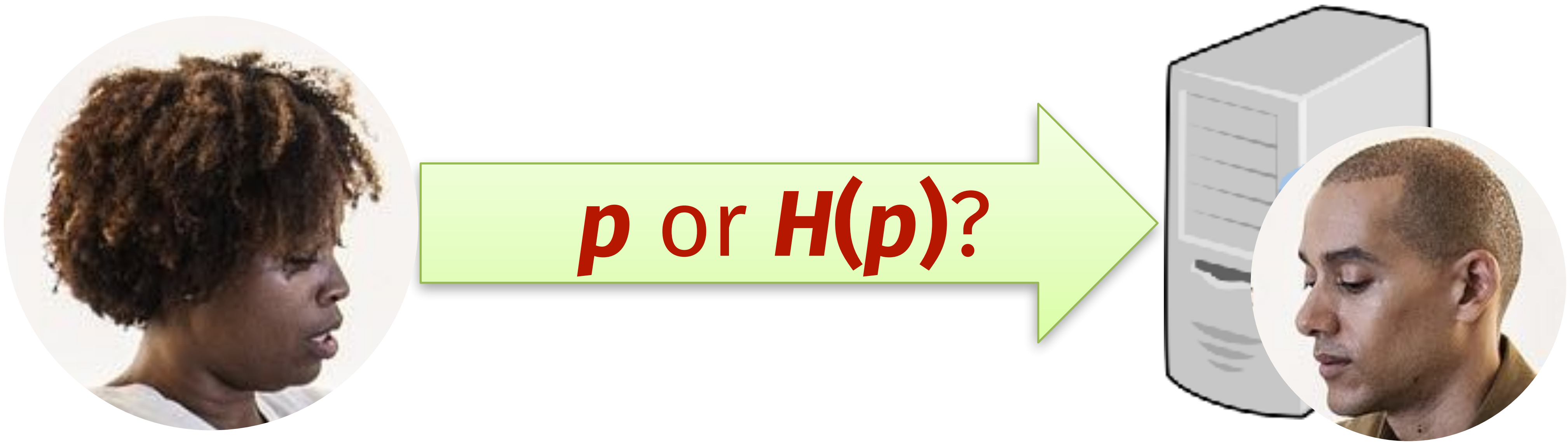
Offline vs online dictionary attack

- PBKDF2 is vulnerable to an *offline* dictionary attack in which Mallory:
  - Compromises the target device to learn salt, count, pbkdf2(pwd, salt, count)
  - Guesses many passwords on her own computing cluster, perhaps in parallel
  - Makes only 1 password guess on the target device
- *Online* dictionary attack requires Mallory to check guesses with server
  - Opportunity for rate limiting



Password dilemma

password ***p***



database of ***H***(***p***),
where ***H*** = pbkdf2, etc

Alice wants to authenticate to <u>bob.com</u>. Does she send p or $H(p)$?

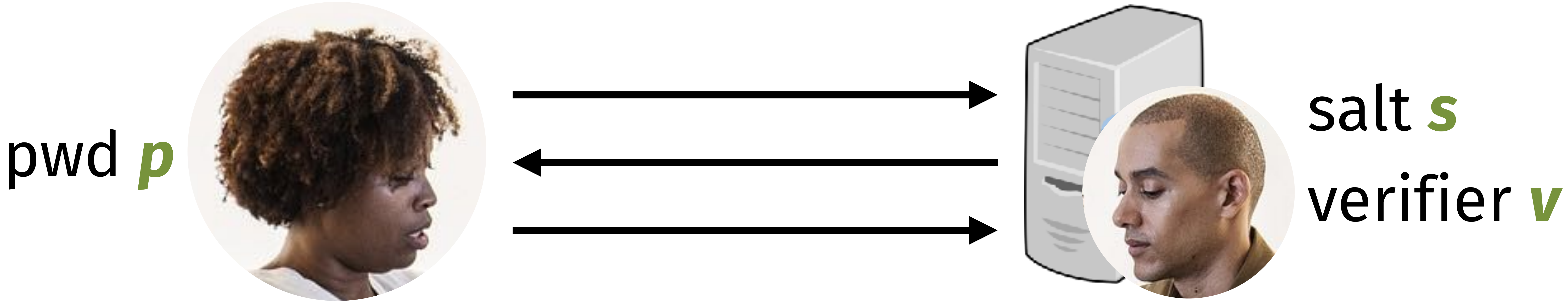
- If Alice sends $H(p)$, then the stored hashed database is very sensitive
- If Alice sends p, then the transmission itself is very sensitive (← done in practice)

"**Cryptography** is how people get things done when they need one another, don't fully trust one another, and have adversaries actively trying to screw things up."

–Ben Adida

Source: benlog.com/2018/01/07/crypto-as-in-crypto/

Objective: verify passwords *without* seeing them!



- Alice knows a password *p* but doesn't want to share it with anyone, even bob.com

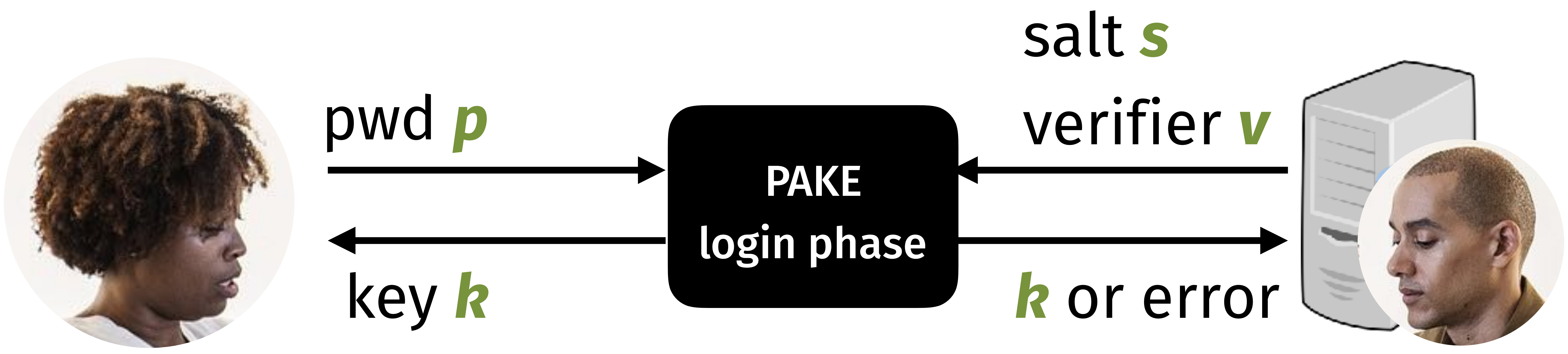
- If bob.com never sees the password then he cannot accidentally store it

KrebsonSecurity
In-depth security news and investigation

21 MAR 19 **Facebook Stored Hundreds of Millions of User Passwords in Plain Text for Years**

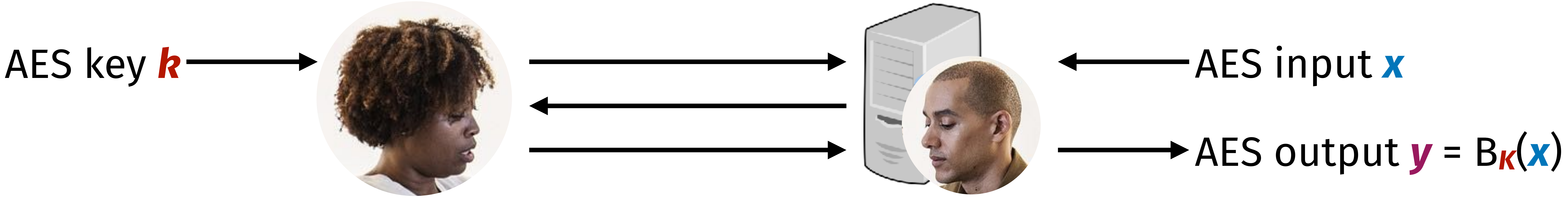
Hundreds of millions of **Facebook** users had their account passwords stored in plain text and searchable by thousands of Facebook employees — in some cases going back to 2012, KrebsOnSecurity has learned. Facebook says an ongoing investigation has so far found no indication that employees have abused access to this data.

Password authenticated key exchange (PAKE)



- Signup phase: real Alice registers a password p, gives verification string v that Bob can use to detect if he’s talking to someone who knows p
- Login phase: the parties interact, after which
  - If Bob is speaking to the real Alice with password p → they get a shared key k
  - If Bob is speaking to Mallory who doesn't know p → he learns this fact
- Security goals: Bob never learns p, and ideally Alice never learns s

Building block: Oblivious pseudorandom function

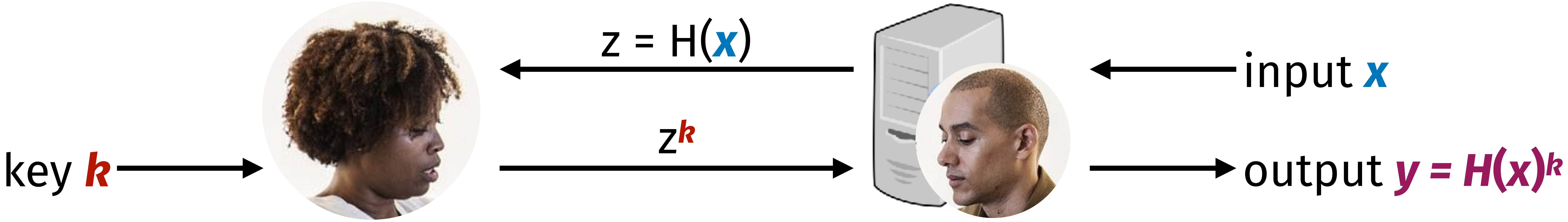


- Let's take a step back, address a different-looking question
  - Alice has a key, Bob has a message
  - Can we compute a block cipher on this key + message without sharing the key?
- Turns out the answer is **yes!**

Why an Oblivious PRF might help

```
pbkdf2(string password, string salt, int count):
    string key = ‘’
    U0 = S
    for(j = 1 to count):
        Uj = prf(password, Uj-1)
        key = key ⊕ Uj
    return key
```

Constructing an Oblivious PRF (but not for AES)



- $B_k(x) = H(x)^k$ is pseudorandom when calculated over a group where discrete logs are hard (e.g., modular arithmetic, elliptic curves)
  - Note: it requires ~milliseconds to compute, rather than ~nanoseconds of AES
- The above protocol is an oblivious method to calculate B
  - Hardness of discrete log prevents Bob from learning k from $H(x)^k$
  - Preimage-resistant hash function H prevents Alice from reversing z to learn x, if Bob chooses x at random (which might be okay in certain circumstances)

Oblivious computing

- This algorithm is fundamentally different from everything we have seen so far in this course: it protects sensitive data *while computing*, even from the other people we are communicating with

- Will see many more examples of oblivious computing next week

- From Oblivious PRFs, can build many other useful crypto primitives

- One example: “blind signatures” in which Alice can sign a message without knowing what she has just signed

  - Cloudflare’s Privacy Pass: reduces CAPTCHAs when using Tor

  - Anonymous e-cash

Secure Remote Password (SRP) protocol

We can obliviously compute the PAKE primitive directly!

pwd **p**

choose a randomly

ID_{Alice}, $A = g^a$ →

← s, $B = c\,v + g^b$

public constant

ephemeral secret "one time pads" v

salt **s**, verifier $\mathbf{v = g^{H(p,s)}}$

choose b randomly

$u = H(A, B)$
$x = H(p, s)$
$S = [B - c\,g^x]^{(a + ux)}$
$K = H(S)$

$u = H(A, B)$
$S = S = (Av^u)^b$
$K = H(S)$

claim: shared key

Claim: if Alice knows p, then Alice and Bob compute the same K
(how can they test whether they have the same key?)

Question: How do we actually build hash functions?

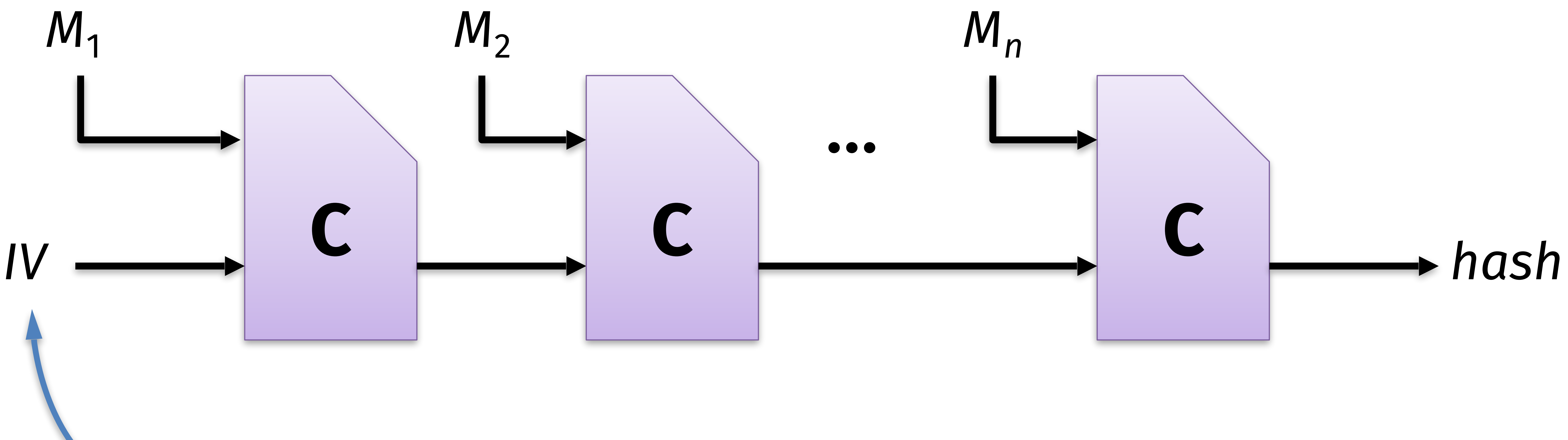
- In January, we saw
  - How to build block ciphers like AES
  - How to build hash functions from a compression function
- But we never saw how to build a compression function; let’s rectify that
- Also, we will discuss a different technique to construct hash functions

Reminder: Merkle-Damgård paradigm

Build a variable-length input hash function from two primitives:

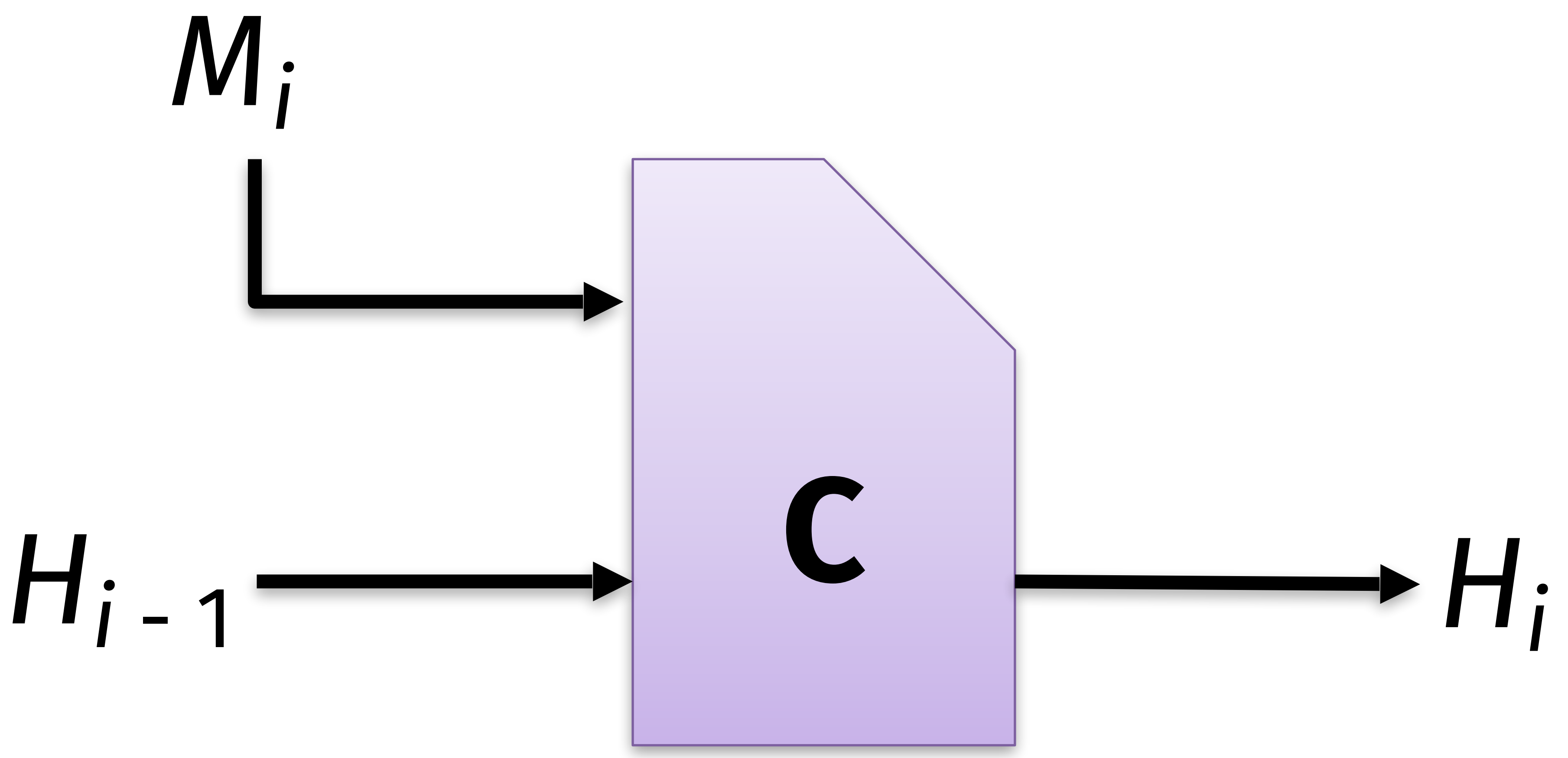
? 1. A fixed-length, *compressing* random-looking function

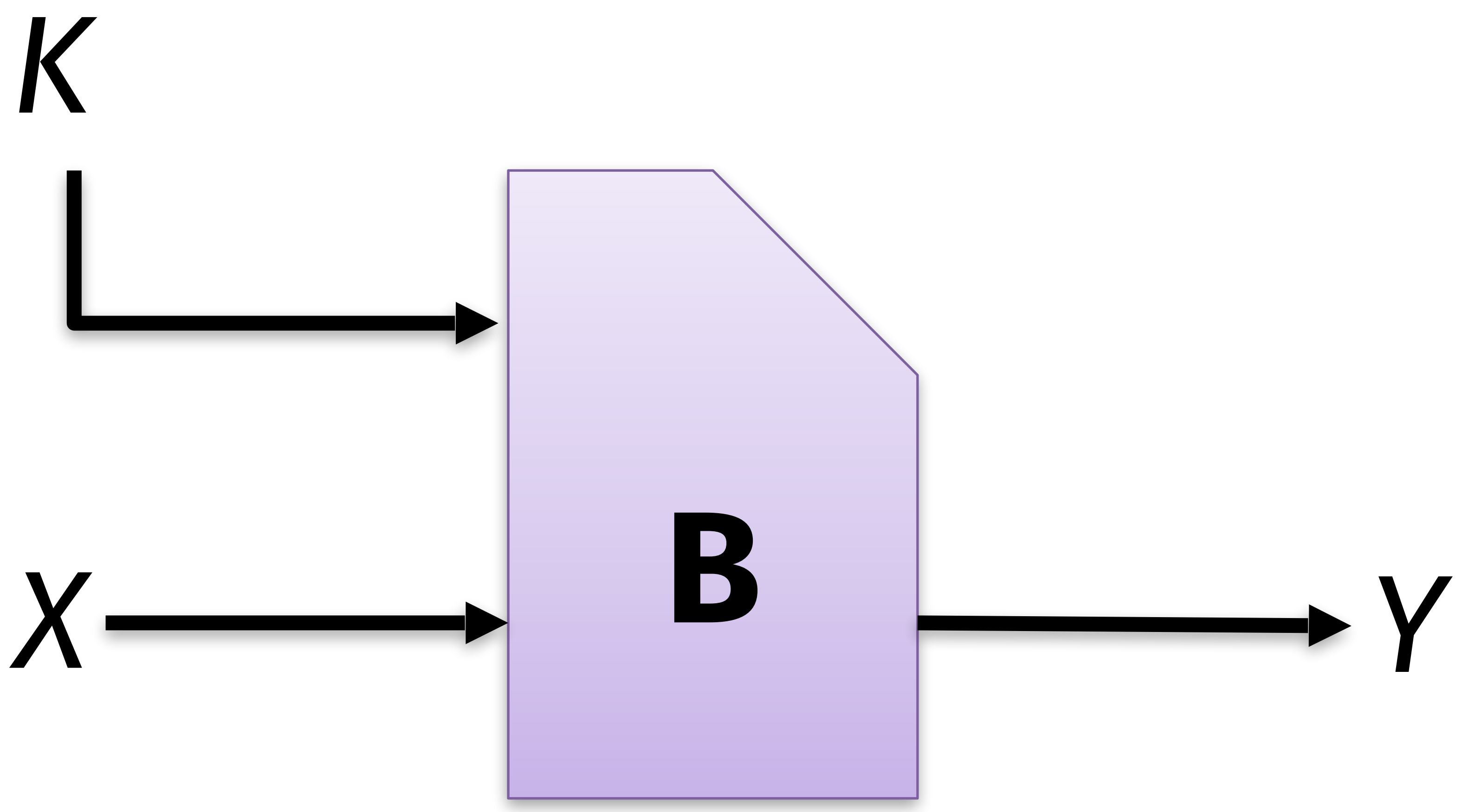
✔ 2. A mode of operation that iterates this function multiple times in a smart manner



IV for hash function is typically fixed in spec, not user

How do we build a compression function?





Answer: use a block cipher?

1. Rabin's *Digitalized Signatures* (1978)

Idea: form a hash function through iterated DES

Begin with some constant IV

$$H(M) = \mathrm{DES}_{m_\ell}(\mathrm{DES}_{m_{\ell-1}}(\cdots(\mathrm{DES}_{m_1}(h_0))\cdots))$$

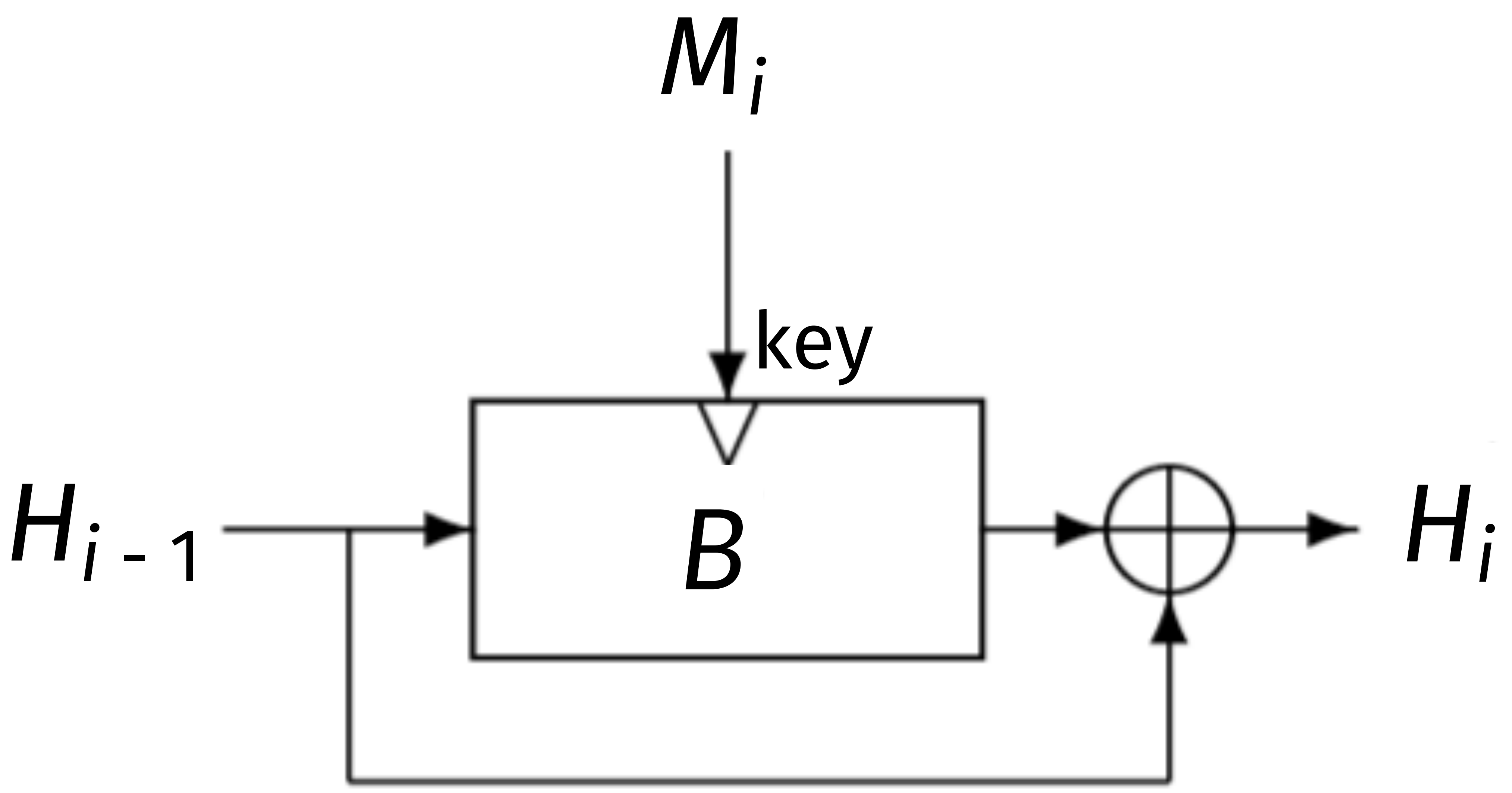
Interpret message blocks as DES keys

Q: Is it okay to use a message in place of a block cipher's key?

A: In general, no! While messages have structure and may even be adversarially-controlled, we have been assuming so far that keys are totally unpredictable to the adversary. … But let's go with this anyway.

2. Davies-Meyer

<u>Deceptively compact picture</u>



SHA-2's compression function has a Davies-Meyer design

<u>Detailed math</u>

H0 = some pre-defined constant

H1 = B(M1, H0) ⊕ H0

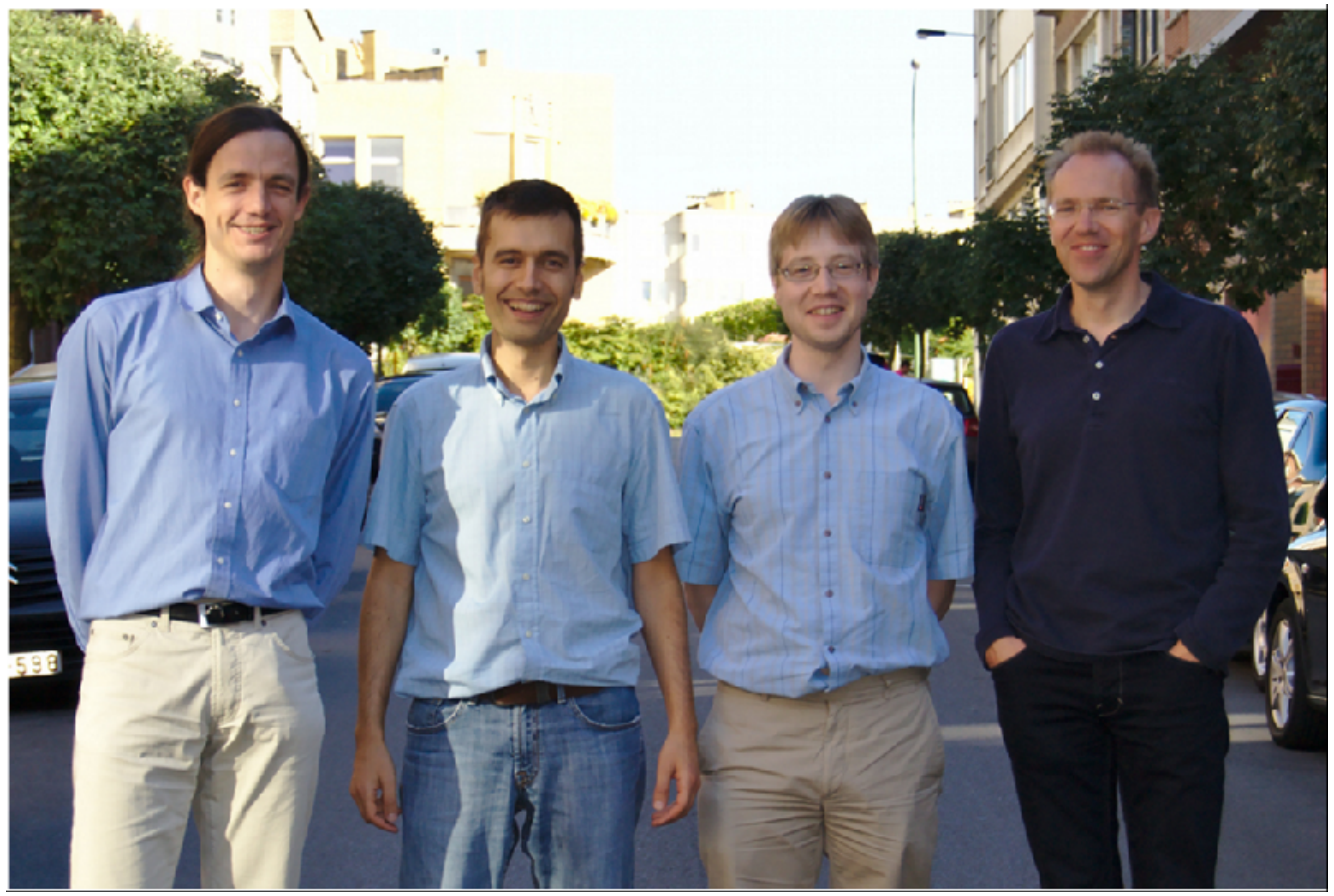
H2 = B(M2, H1) ⊕ H1
= B(M2, B(M1, H0) ⊕ H0)
⊕ B(M1, H0) ⊕ H0

H3 = B(M3, H2) ⊕ H2
= B(M3, B(M2, B(M1, H0) ⊕ H0)
⊕ B(M1, H0) ⊕ H0)
⊕ B(M2, B(M1, H0) ⊕ H0)
⊕ B(M1, H0) ⊕ H0

...and so on!

SHA-3: quest for a Merkle-Damgard alternative

- 2004: Weakness found in Merkle-Damgard, eventually would break SHA-1 in 2017
- 2007: Call for submissions
- 2008: 64 submissions received
- 2009-12: Three workshops, one before each cutdown: 64 → 51 → 14 → 5 → 1
- Oct 2012: Keccak announced as winner, created by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche
- Aug 2015: NIST publishes Federal Information Processing Standard (FIPS) 202 standardizing Keccak

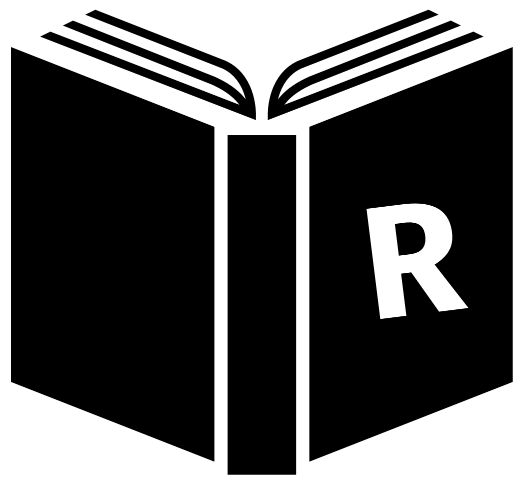


NIST’s AES call

“Algorithms will be judged on the extent to which their output is indistinguishable from a ***random permutation*** on the input block.”

NIST’s SHA-3 call

“The extent to which the algorithm output is indistinguishable from a ***random oracle***.”



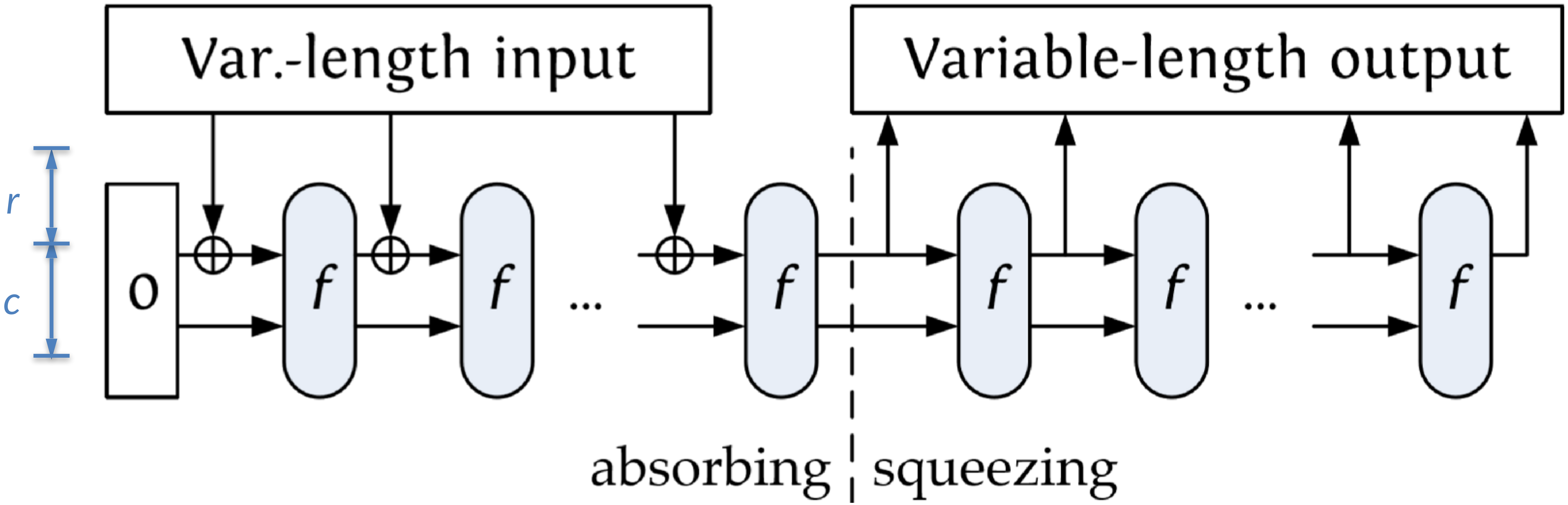
Why NIST chose Keccak, in their words

1. “Offers acceptable performance in software, and *excellent performance in hardware*.”

2. “Has a *large security margin*, suggesting a good chance of surviving without a practical attack during its working lifetime.”

3. “A fundamentally new and different algorithm that is entirely *unrelated to the SHA-2 algorithms*.”

Sponge functions



Split state into two components

- r = rate, which influences speed
- c = capacity, which influences security

Arbitrary input and output length

- More flexible than M-D hash functions
- Facilitates design of higher-level crypto