Computer Security: Hashing

B. Jacobs and J. Daemen
Institute for Computing and Information Sciences – Digital Security
Radboud University Nijmegen
Version: fall 2016

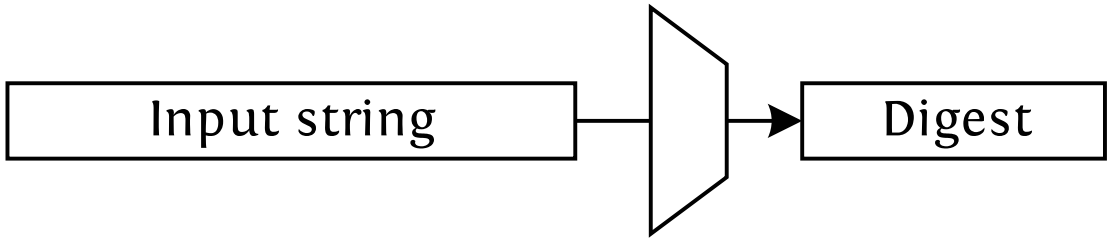
Outline

Hash function definition

Applications and expected properties

Hash function standards

Hash function definition



- A cryptographic hash function h takes a message m of arbitrary length and yields an outcome $h(m)$ of fixed length

$$h\colon \{0,1\}^\star \longrightarrow 2^N \qquad \text{typical values for } N: 160, 256, 512$$

- *Every bit of $h(m)$ depends in a complicated way of all bits of m*
- $h(m)$ is called the hash (value) of m. Alternative names:
  - message digest
  - (cryptographic) fingerprint
  - *verhaspelingsfunctie* (please never use this term)
- A cryptographic hash function is a very powerful primitive ...

The ideal hash function: Random Oracle

- What would the ideal cryptographic hash function look like?
- It is called a Random Oracle (RO)
- Random Oracle can be built but is not practical
- We will try to build hash functions that behave like a RO

Random Oracle Inc.: letter answering service!

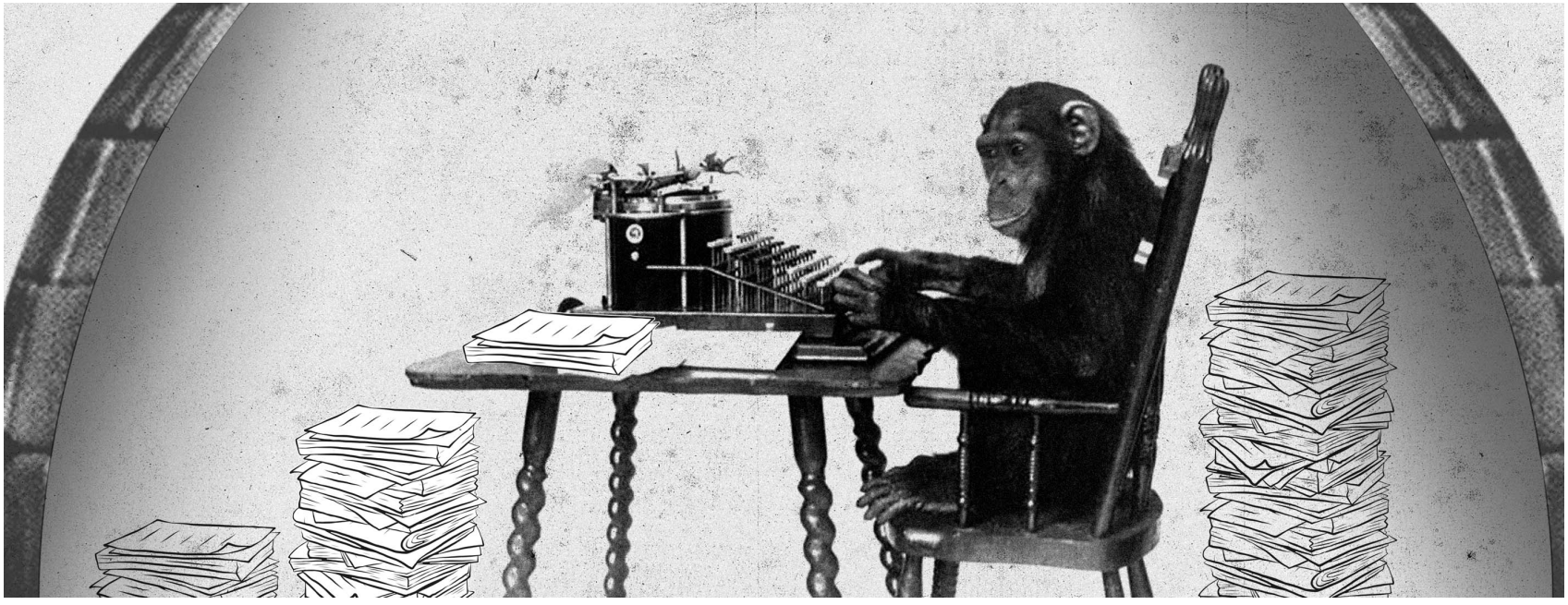
Random Oracle Inc.

1. Message m arrives at Random Oracle Inc.

Random Oracle Inc.

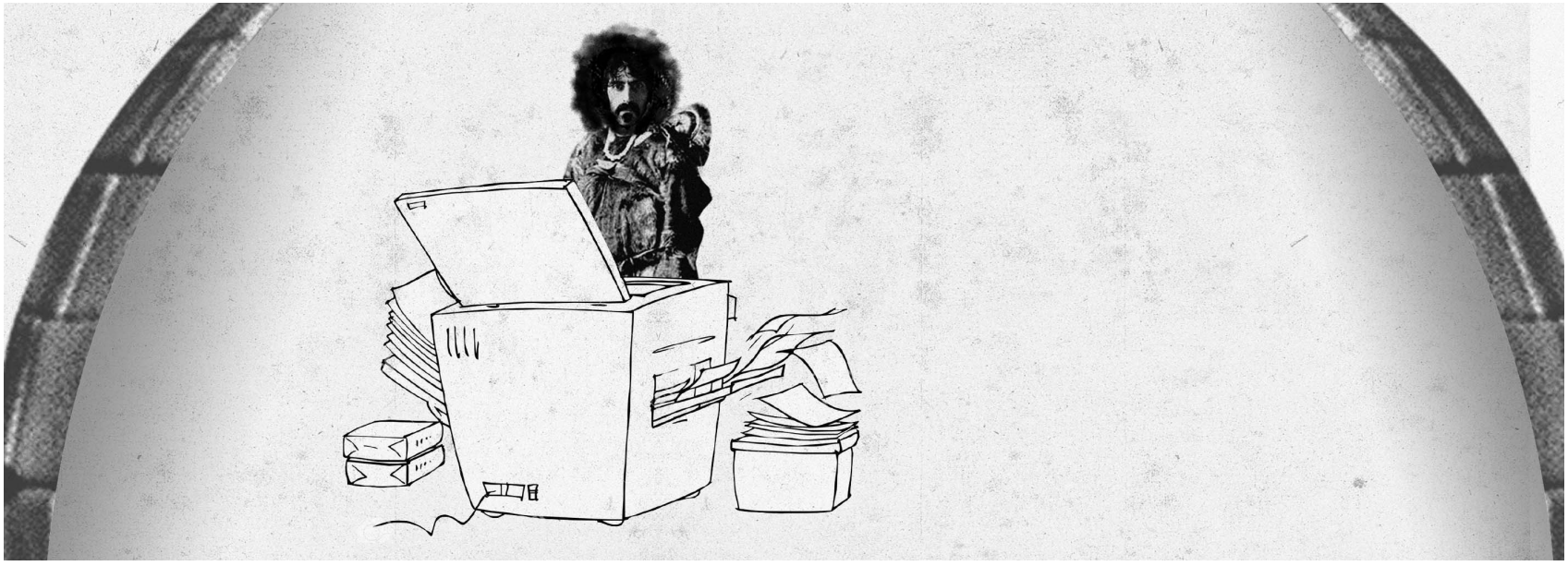
2a. If m was received earlier, manager picks it from archive. Its file will also contain the returned response z

Random Oracle Inc.



2b. If not in archive, employee will (randomly) generate response z

Random Oracle Inc.



3. Manager copies response z, from archive (2a) or freshly typed (2b)

Random Oracle Inc.

4. Manager puts file with (m, z) (back) in archive

Random Oracle Inc.

5. Manager sends response z by courier to sender of m

Random Oracle returns unrelated responses for different messages

Let us check this with a real-world hash function

SHAKE is one of the hash functions defined in the NIST SHA-3 standard

Let us apply SHAKE to the message Security: interesting!

$$\textbf{SHAKE}(\text{“Security: interesting!”}) = \texttt{336113159061d2163feaf7ae12ddad58}$$

If we change a detail:

$$\textbf{SHAKE}(\text{“Security: interesting?”}) = \texttt{6548b7e6db8f86ae7f9e6d020698c5aa}$$

We can also apply it to a file, e.g.

$$\textbf{SHAKE}(\text{symmetric.tex}) = \texttt{ec02b28d5949acc3ecbaca71a10e3871}$$

Hash yourself!

On a (linux) command line you can run your own hash, e.g., as:

- `sha256sum` *`file`*
- `openssl sha256` *`file`*

SHA256 is one of the hash functions defined in the NIST SHA-2 standard

Using Python

- install Python
- get `CompactFIPS202.py` and `SHAKE.py` from the security homepage
- on command line type `Python SHAKE.py` *file*

Or with Python (3) built-in hash functions

```
> import hashlib
> h = hashlib.new("md5")
> h.update(b"Hash that string")
> print(h.hexdigest())
```

Coin-tossing by email

- Suppose Alice and Bob want to agree by email who's cooking tonight, using coins
- They each toss a coin, and agree:
  - if the outcomes are equal, Alice prepares the dinner
  - otherwise Bob does
- How to do this securely, without the possibility to cheat?

  (and without a trusted third party, TTP)

Coin-tossing by email: hash-based protocol

Assume a hash function h, and coin outcomes C_A of A and C_B of B

Commitment phase:

$$A \longrightarrow B\colon Z_A = h(N_A \| C_A) \quad \text{with } N_A \text{ randomly generated by } A$$
$$B \longrightarrow A\colon Z_B = h(N_B \| C_B) \quad \text{with } N_B \text{ randomly generated by } B$$

Revealing phase

$$A \longrightarrow B\colon N_A, C_A \;\; B \text{ checks honesty of } A : Z_A \stackrel{?}{=} h(N_A \| C_A)$$
$$B \longrightarrow A\colon N_B, C_B \;\; A \text{ checks honesty of } B : Z_B \stackrel{?}{=} h(N_B \| C_B)$$

After this 4-email protocol both can check $C_A \stackrel{?}{=} C_B$ and arrange dinner

Coin-tossing by email: requirements

- B cannot derive C_A from z_A
  - N_A shall be unpredictable
  - having $h(N_A \| C_A)$ shall not help in finding C_A: one-way AKA preimage resistant
- B cannot find N_B, N'_B, with $h(N_B \| 0) = h(N'_B \| 1)$
  - Collisions may exist if Z_A is shorter than N_A
  - Finding collision shall be computationally infeasible: collision-resistant

Message compression for authentication

Electronic (read: cryptographic) signatures can be legally recognized

- In Europe: EU directive 1999/93/EC
- Usage and implementation details left up to the countries
- Used for tax declaration, contracts, statements, PV, ...
- Requires certified secure signature-creation device

In practice: a smart card chip on an ID card

- Contain secret key to generate signatures: $S = K_{\text{sign}}\{M\}$
- Low-bandwidth communication protocol from the 80s
- Commands and responses limited to 255 bytes
- Problem: sending 1MB pdf to chip would take over 60 seconds

Message compression for authentication (cont'd)

Solution to the timing issue: hashing!

- PC/tablet/phone computes 256-bit hash of message $Z = h(M)$
- PC/tablet/phone presents Z to chip
- chip returns $S = K_{\text{sign}}\{Z\} = K_{\text{sign}}\{h(M)\}$

Signatures (public-key, see later) are always computed in two stages:

(1) compression of M into $Z = h(M)$ with fast h

(2) generation of signature $S = K_{\text{sign}}\{Z\}$ with slow K_{sign}

Message compression: requirements

Outsider attack:

- Imagine: Bob agrees with Eve to buy her car for 2000 Euro by 2017
- He seals the deal by signing a message M that states this
- He hands the message M and signature S to Eve
- Forgery attack
  - Eve builds M' with same hash as M that says 5000 Euro

$$S = K_{\text{sign}}\{h(M)\} = K_{\text{sign}}\{h(M')\}$$

  - forgery: S is also a valid signature for M'
  - If Bob does not pay, Eve sues him using M' and S as evidence
- Required property: 2nd preimage resistance
  - given M, finding M' with $h(M') = h(M)$ shall be infeasible

Message compression: requirements (cont'd)

Insider attack:

- Imagine: Eve agrees to buy a house from Bob for 300K Euro
- Eve and Bob agree on a contract M and present it to notary
- Notary signs it and it becomes official
- Forgery attack
  - Eve builds M and M' in advance with $h(M) = h(M')$
  - M mentions 300K Euro, M' mentions 200K Euro
  - Eve and Bob meet with notary, present M and notary signs it
  - Later Eve disputes payment amount and shows M' to back it up
- we require collision-resistance

Note:
collision-resistance implies 2nd preimage resistance but not vice versa

Password protection on servers

- It is not wise to store user passwords on a server in the clear:
  - other users (administrators) may abuse them
  - hackers may break into the server and get them
  - google for *password leakage*
- Good solution: replace passwords by actual cryptography
- Usual *solution*: store hashes of passwords
  - after entering a password, the server computes the hash and compares it to the data base entry of the user
  - hash function requirement: given $h(M)$, it is hard to find M: preimage resistance

  The password file then looks like this:

user	password hash
bart	$h(\text{passwd1})$
peter	$h(\text{passwd2})$
⋮	⋮

Password protection on servers (cont'd)

What can a hacker do with such a password file?

user	password hash
bart	$h(\text{passwd1})$
peter	$h(\text{passwd2})$
⋮	⋮

Dictionary attack: hashing plausible passwords until we have a match

- try words from dictionary and common names: extremely fast
- if password policies apply: build passwords from a dictionary combined with numbers, special characters and capitalization: very fast
- try all combination of characters up to some length (e.g. 8): fast
- for reasonable protection: original passphrases of sufficient length

see https://youtu.be/7U-RbOKanYs

Password protection on servers (cont'd)

What can a hacker do with such a password file?

user	password hash
bart	$h(\text{passwd1})$
peter	$h(\text{passwd2})$
⋮	⋮

Attention point: *multi-target aspect*

- $h(\text{passwdGuess})$ can hit any entry in the file
- success probability increases with number of entries
- probability of bad passwords increases with number of entries

Password protection: diversification

- Preventing multi-target aspect by diversifying hash input
- Username-based:
  - include unique username in hash input: $h(\text{username};\text{passwd})$
  - attempt must be of the form $h(\text{username};\text{passwdGuess})$
  - so each attempt is dedicated to a single user's password
  - prevent collisions between usernames on different servers: include servername, e.g., URL
  - prevent leakage of users cycling between passwords: include password serial number

Password protection: salting

- Preventing multi-target aspect by diversifying hash input
- Salt-based:
  - include random *salt* per user: $h(\text{salt};\text{passwd})$
  - if salt is unique per user: same effect as username-based
  - most commonly used, mostly for historical reasons

  A salted password file looks like this:

user	salt	hash
bart	bla	$h(\text{bla}, \text{passwd})$
peter	aap	$h(\text{aap}, \text{passwd})$
⋮	⋮	⋮

Most commonly used — but not by LinkedIn, as became clear when its database of 6.5M logins leaked in June 2012.

Intermezzo: domain separation

- Salting is an application of domain separation
- A hash function h can be used to build two hash functions h_0 and h_1:
  - $h_0(M) = h(0|M)$
  - $h_1(M) = h(1|M)$
  - if h behaves like a RO, both h_0 and h_1 behave like RO's
- Generalization: a hash function can be used to build 2^n hash functions
  - $h_a(M) = h(a|M)$
  - with a any n-bit string
- many protocols fail due to lack of domain separation

Password protection: key stretching

- Described attacks are economical because hashing is cheap
  - designed for speed: desirable in most applications
  - cost decreases over time due to Moore's law
  - plus: dedicated hardware for hashing (due to Bitcoin, see later)
- Strength of password may reduce from 10K Euro in 2010 to < 1 Euro in 2020
- Approach: artificially slow down the hashing: *key stretching*
- Traditional solution: storing x_N computed as

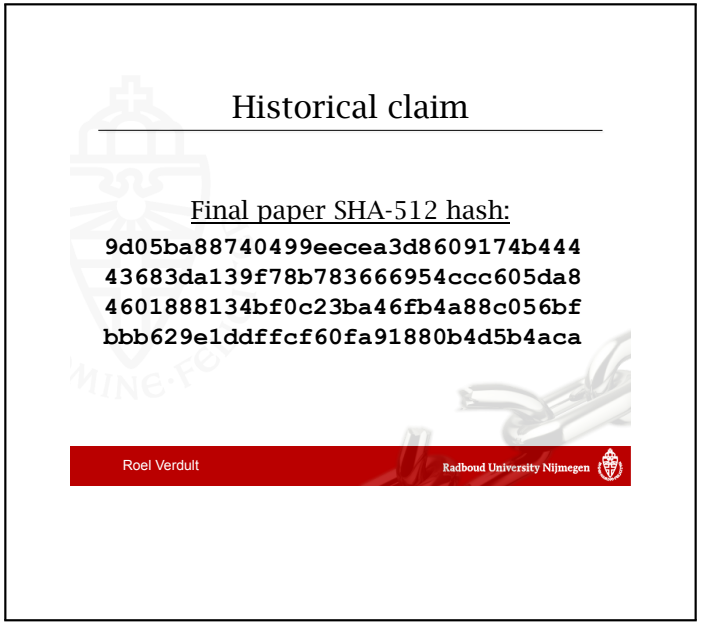
$$x_0 = 0 \quad \text{and } x_{n+1} = h(x_n|\text{password}|\text{salt}).$$

- Modern solution: have dedicated resource-hungry hash functions
  - result of open contest: https://password-hashing.net/
  - principle: *cost/hash shall not decrease with increasing resources*
- Balance between convenience (latency) and security (cracking cost)

Hash application: integrity check

- Suppose you run out of disc space and wish to store a large file M "in the cloud" — so on someone else's computer — but you worry about (detecting) integrity violations
- The solution is:
  - store M remotely
  - keep $Z = h(M)$ locally
- After retrieving the file, say M', verify $h(M') \stackrel{?}{=} Z$
  - if $h(M') = Z$, you know $M' = M$.
  - unless someone found M' with $h(M') = h(M)$
  - this requires 2nd preimage resistance
- The same technique is used in other situations, e.g.
  - downloading software (hash must be stored elsewhere, or be signed)
  - protecting evidence in forensic investigation, etc.
  - trusted platform module (TPM)

Originality claim for banned publication



Last slide of Roel Verdult's Usenix Aug'2013 presentation, after forced withdrawal of the paper on Megamos Chip vulnerabilities. Article was finally published in 2015

Application requires preimage resistance

The traditional hash function properties

Traditionally, from a cryptographic hash function h one expects the following properties:

(1) preimage resistance: given a hash value x, finding an m with $h(m) = x$ shall have expected cost 2^n hash function computations

(2) second preimage resistance: given m, finding $m' \neq m$ with $h(m) = h(m')$ shall have expected cost 2^n hash function computations

(3) collision resistance: finding *any* pair $m \neq m'$ with $h(m) = h(m')$ shall have expected cost $2^{n/2}$ hash function computations

These cost measures are the ones realized by a random oracle.
Collision resistance is only $2^{n/2}$: so-called birthday bound

Birthday paradox

In a group of 23, the chance that two have same birthday is above $\frac{1}{2}$

- Surprising . . . at first sight
- Let's study it: build group by add 1 person at a time
  - 1 person A: probability of birthday clash is 0
  - Add B: probability that it clashes with A is $1/365$
  - Add C: probability that it clashes with A or with B is $2/365$
  - Add i-th person: probability it clashes with one $i-1$ is $(i-1)/365$
- Probability of a birthday clash for i people is sum of all these:

$$\frac{1+2+3\ldots i-1}{365} = \frac{(1+i-1)+(2+i-2)\ldots}{365} = \frac{i(i-1)}{2*365}$$

- This becomes equal to $1/2$ when $i(i-1)/(2*365) \approx 1/2$ or $i \approx \sqrt{365}$

Birthday paradox (once more, with precision)

This slide is for information only!

- Let us compute the probability of non-collision
- Build group by add 1 person at a time
  - 1 person A: prob. of no clash is 1
  - Add B: prob. of no clash with A is $1 - 1/365$
  - Add C: prob. of no clash with A or with B is $1 - 2/365$
  - Add i: prob. of no clash is $1 - (i-1)/365$
- Probability of no clash is product. Using $1 - \epsilon \approx e^{-\epsilon}$:

$$\prod_{j=1}^{i} \left(1 - \frac{j}{365}\right) \approx \prod_{j=1}^{i} e^{-\frac{j}{365}} = e^{-\frac{i(i-1)}{2 \times 365}}$$

Setting this equal to $1/2$ gives $i(i-1)/2 * 365 = \ln(2)$ or $i(i-1) \approx 506 = 23 \times 22$.

iCIS | Digital Security
Radboud University

Collision probability

In a set of i hashes, the chance that two are equal is about $i^2/2^{n+1}$

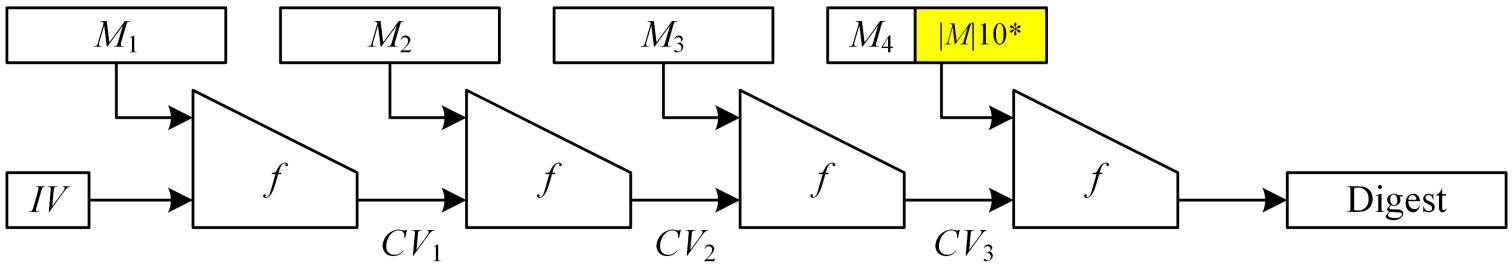
- Same reasoning as birthday paradox:
  - we assume random hashes instead of random birthdays
  - domain size: 2^n instead of 365
- $i^2/2^{n+1}$ becomes close to $1/2$ if $i^2 \approx 2^n$ so:

The expected effort for finding a collision in a good n-bit cryptographic hash function is close to $\sqrt{2^n} = 2^{n/2}$ hash function evaluations.

iCIS | Digital Security
Radboud University

End of the 80s: Merkle-Damgård

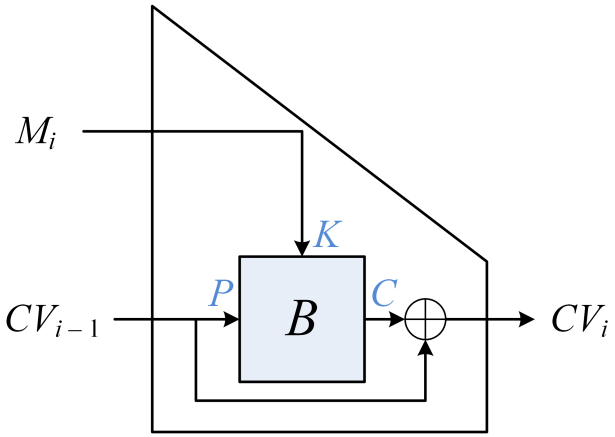
Difficulty: how to build a function accepting input with any length?



- Mode of use of a fixed-input-length compression function f
- Collision-resistance preserving
  - collision in hash function implies collision in f
  - implies fixing IV and code length of message in padding
- Reduces design of big hash function to design of small f
- Dominated hash function design for almost 20 years
- Even today some people think this is the way to do it

Davies-Meyer mode for compression function f

How to build an efficient collision-resistant compression function?



- Idea: use a block cipher with feedforward
  - due to Merkle-Damgård proof: collision-resistance preservation
  - without feedforward it is trivial to generate collisions for f
- Initially DES was proposed
  - problem: block size too small for
  - generating a collision takes only $2^{n/2} = 4$ billion DES computations

MD5 and standards SHA-1 and SHA-2

- MD5 [Ron Rivest, 1991]
  - based on MD4 that was an original design
  - Merkle-Damgård and Davies-Meyer using dedicated *block cipher*
  - security based on addition, rotation and XOR: ARX
  - 128-bit digest
- SHA-1 [NIST, 1995] (after SHA-0 [NIST, 1993])
  - *designed* at NSA, mostly a rip-off of MD5
  - SHA stands for Secure Hash Algorithm (wishful thinking)
  - 160-bit digest
- SHA-2 series [NIST, 2001 and 2008]
  - *reinforced versions of SHA-1*, again coming from NSA
  - 6 functions with 224-, 256-, 384- and 512-bit digest
- no motivation or rationale was ever given for any of them
- no actual innovation since 1991

The MD5 saga

- 1993: compression function shown weak
- In the following years it was adopted widely in SSL etc.
- 1996: collisions in compression function
- 2003-2004: great advances in analysis of MD5
- 2004: actual collisions for MD5 found by Prof. Wang
- despite weaknesses, corporate IT co. unwilling to abandon MD5
  - *yes, but these weaknesses are just theoretical*
- 2005: Lenstra, Wang, and de Weger generate fake TLS certificates
- 2008: Nostradamus attack (next slide)
- 2010-2012: Espionage malware Flame creates fake Microsoft update certificates.
- Today MD5 largely replaced by SHA-256 but not everywhere
- Lessons learnt
  - in retrospect MD5 is a very weak hash function
  - put in the field (internet) without considering public scrutiny

Nostradamus attack with MD5

In 2008, before the US-presidential elections, 3 Dutch researchers (M. Stevens, A. Lenstra, B. de Weger) constructed 2 different messages:

$$m_1 = \boxed{\cdots \text{ Obama will be the next president } \cdots}$$

$$m_2 = \boxed{\cdots \text{ McCain will be the next president } \cdots}$$

with the same hash: $\mathbf{md5}(m_1) = \mathbf{md5}(m_2)$.

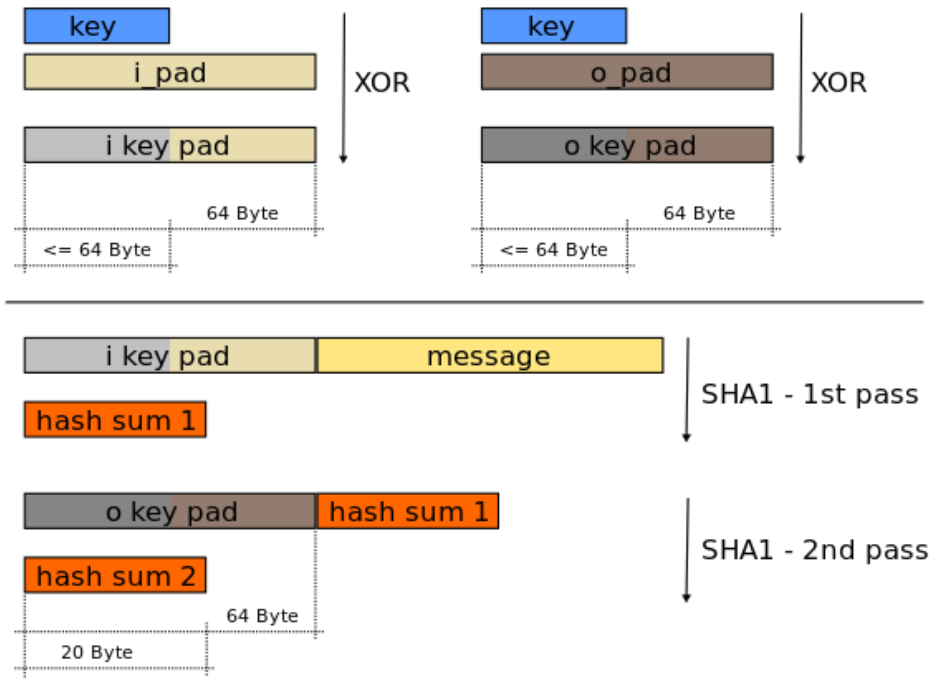
They published this hash and claimed that they could predict the future!
See `www.win.tue.nl/hashclash/Nostradamus`

Security of SHA-1 and the SHA-2 functions

- SHA-1
  - 2004-2007: theoretical collision attacks in effort $\approx 2^{61}$
  - ongoing effort likely to give collisions within 12 months
  - broken but not as bad as MD5
- SHA-2 series: still a solid safety margin despite public scrutiny
- But Merkle-Damgård is not sound as a mode!
- Theoretical problems of Merkle-Damgård:
  - 2nd preimages with effort below 2^n attempts
  - failure to meet Random Oracle security level for other properties
- Real problem of Merkle-Damgård: length extension weakness
  - adversary knowing $h(m)$ but not m can compute $h(m|\text{ pad }|m')$ for any m' of his choice
  - naive MAC function built from hash function $\text{mac} = h(K_{\text{mac}}|m)$
    - secure if h would be a random oracle
    - trivial forgery if h uses Merkle-Damgård

The HMAC authentication mode [FIPS 197]

HMAC is a patch to compensate for the length-extension property: call the hash function twice per mac



The mode MGF1 [PKCS #1] and stream encryption

- In many applications we need a long hash output
  - when used for deriving multiple keys (SSL, TLS, see later)
  - when using for keystream generation, ...
- Mode: Mask Generating Function 1 (MGF1)
  - Compute $h_1 = h(M|1)$, $h_2 = h(M|2)$, $h_3 = h(M|3)$, ...
  - $h = h_1|h_2|h_3|\ldots$
- Stream cipher by taking $M = K|\text{nonce}$
  - $Z_i = h(K|\text{nonce}|i)$
  - this is similar to counter mode

iCIS | Digital Security
Radboud University

SHA-3: the competition

- 2005-2006: MD5 and SHA-1 crisis
- 2008: NIST kicks off the open SHA-3 competition
- Requirements
  - *more efficient than SHA-2*
  - output lengths: 224, 256, 384, 512 bits
  - security: collision and (2nd) pre-image resistant
  - specs, code, design rationale and preliminary analysis
  - patent waiver
- Three-round public process
  - round 1: 64 submissions, 51 accepted
  - round 2: 14 semi-finalists
  - round 3: 5 finalists
- October 2012: NIST announces Keccak as SHA-3 winner

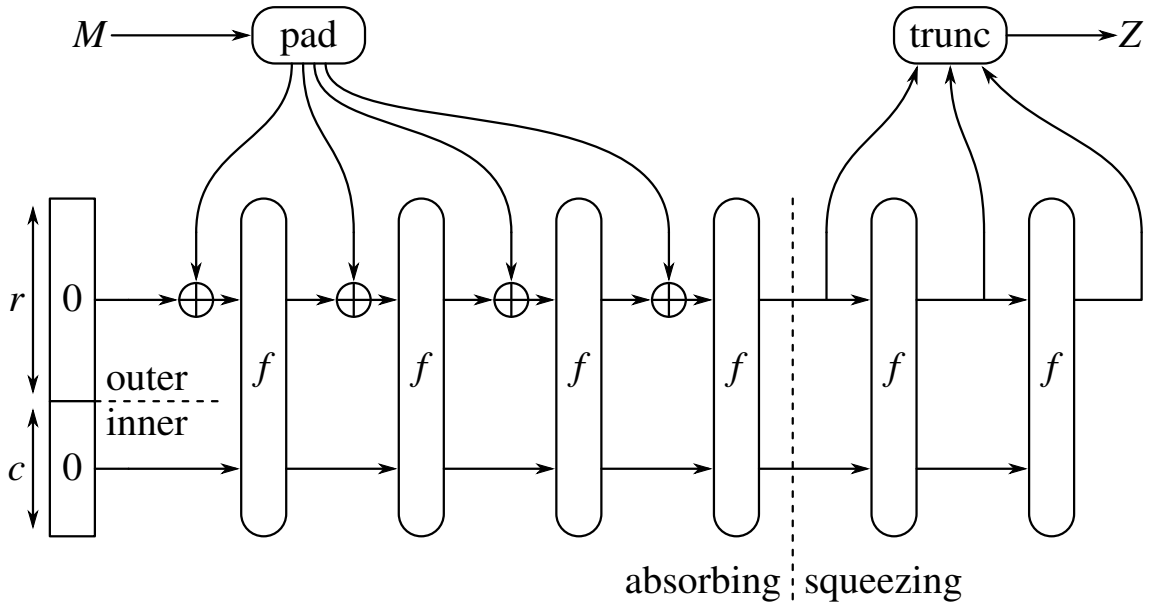
  Designed by [Bertoni, Daemen, Peeters, Van Assche, 2007]
- August 2015: NIST finally publishes the SHA-3 standard: FIPS 202

Keccak: permutation-based hashing

- Basic goals:
  - mode iterating a fixed-length primitive
  - security proof to exclude weaknesses such as length extension
  - simplest possible primitive
  - built-in support for long outputs
- Approach: tabula rasa
  - permutation instead of block-cipher-based compression function
  - bitwise logical operations (XOR, AND, NOT) instead of ARX
- Resulting mode: the sponge construction

  [Bertoni, Daemen, Peeters, Van Assche (Keccak team) 2007]

The sponge construction



- Builds a hash function from a b-bit permutation f, with $b = r + c$
  - r bits of *rate*
  - c bits of *capacity* (security parameter: secure up to $2^{c/2}$)
- Arbitrary-length output: eXtendable Output Function (XOF)

Underlying primitive f: cryptographic permutation

- Similar to a block cipher but without key input
  - *block cipher with fixed key*
  - *every output bit depends on all input bits in a complicated way*
- Keccak is a sponge function with permutations Keccak-f
  - 7 permutations with width 25, 50, 100, 200, 400, 800 and 1600
  - very different from MD5, SHA-1 and SHA-2
  - very different from AES and DES

Keccak(r, c)

- Sponge function using the permutation Keccak-f
  - 7 permutations: $b \in \{25, 50, 100, 200, 400, 800, 1600\}$
  - Only restriction $r + c = b$ for one of these values

    ... from toy over lightweight to high-speed ...
- SHA-3 instance SHAKE128: $r = 1344$ and $c = 256$
  - permutation width: 1600
  - security strength 128
- Lightweight instance: $r = 40$ and $c = 160$
  - permutation width: 200
  - security strength 80: what SHA-1 should have offered
- Security status 2016: huge safety margin

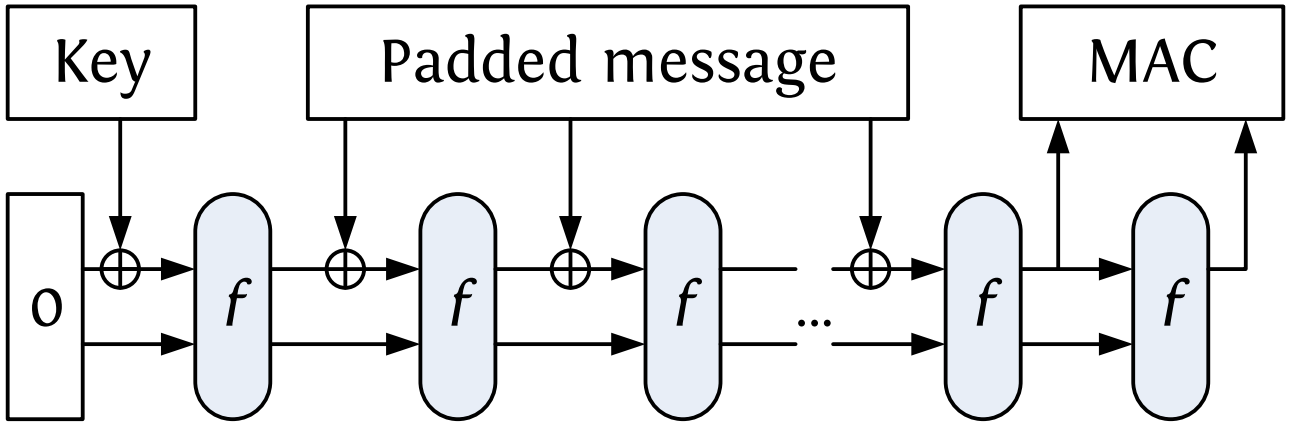
The XOFs and hash functions in FIPS 202

- Four drop-in replacements with same output lengths of SHA-2
- Two *extendable output functions* (XOF)
- All use the 1600-bit version of Keccak-f

Following is for information only

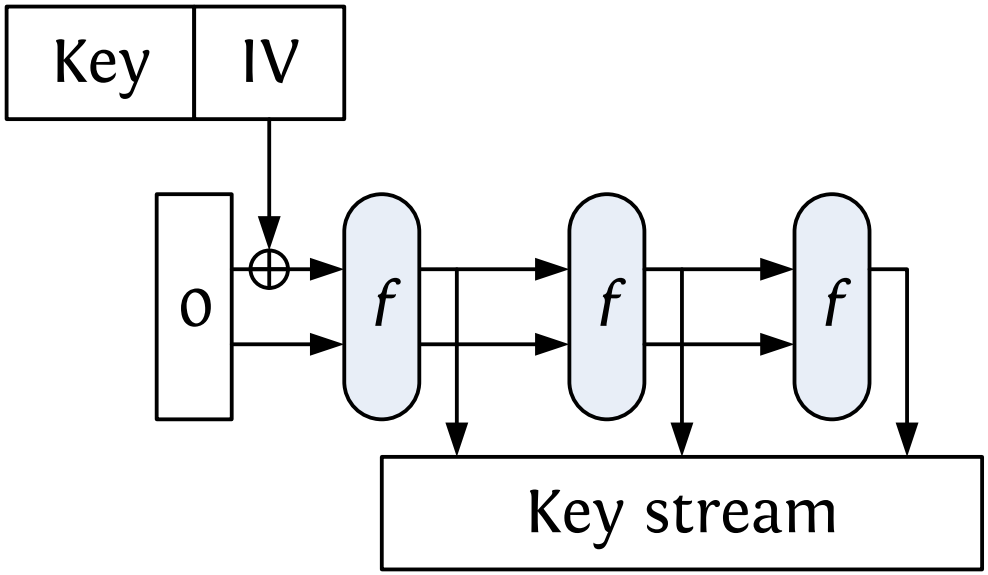
XOF	**SHA-2 drop-in replacements**
Keccak$[c = 256](M\|11\|\mathbf{11})$	
	first 224 bits of Keccak$[c = 448](M\|\mathbf{01})$
Keccak$[c = 512](M\|11\|\mathbf{11})$	first 256 bits of Keccak$[c = 512](M\|\mathbf{01})$
	first 384 bits of Keccak$[c = 768](M\|\mathbf{01})$
	first 512 bits of Keccak$[c = 1024](M\|\mathbf{01})$
SHAKE128 and **SHAKE256**	**SHA3-224** to **SHA3-512**

Mode for MAC (and key derivation)



- No more need for HMAC

Stream cipher mode



- many output blocks per IV: similar to OFB
- 1 block per IV: similar to counter mode

Conclusions

- you can do a lot with hash functions
  - compression, encryption, MAC, key derivation ...
- A good hash function behaves like a random oracle
  - for an n-bit output:
    - Generating (2nd) pre-image takes 2^n hash attempts
    - Generating collision takes $2^{n/2}$ hash attempts
  - multiple independent hash functions from a single one by domain separation
- Standard hash functions
  - MD5 and NIST standard SHA-1: broken
  - SHA-2: same philosophy, still very solid
  - SHA-3: new generation
    - SHAKE128 and SHAKE256: variable output length
    - simplification of modes of use