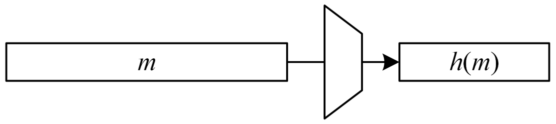
Computer Security: Hashing

B. Jacobs and J. Daemen
Institute for Computing and Information Sciences – Digital Security
Radboud University Nijmegen
Version: fall 2017

Outline

Hash function definition

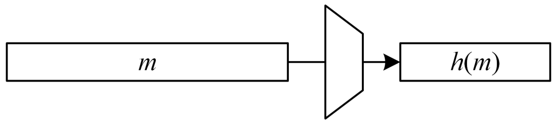
A cryptographic hash function h takes a message m of arbitrary length and yields an outcome $h(m)$ of fixed length



$h: \{0,1\}^* \longrightarrow 2^N$ typical values for $N = 160, 256, 512$

- $h(m)$ is called the hash (value) of m. Alternative names:
  - message digest
  - (cryptographic) fingerprint
  - *verhaspelingsfunctie* (please never use this term)

Hash function schematic and security



(Sketchy) definition of hash function security

A hash function $h(\cdot)$ is secure if every bit of $h(m)$ depends in a complicated way of all bits of m

A secure cryptographic hash function is a very useful primitive . . .

The letter-answering company Random Oracle Inc.

What would the ideal cryptographic hash function look like?

Let us do a thought experiment

Random Oracle Inc.: letter answering service!

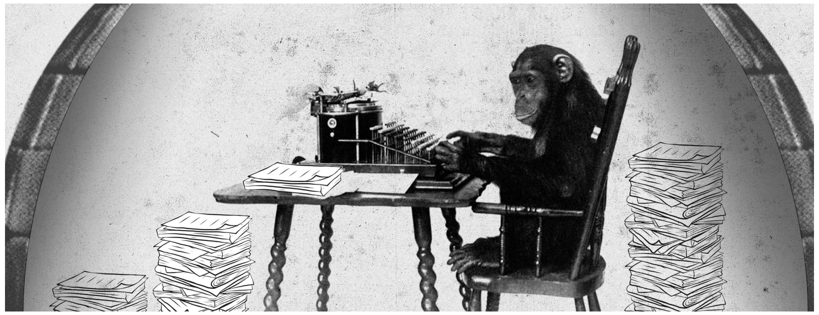
Random Oracle Inc.

1. Message m arrives at Random Oracle Inc.

Random Oracle Inc.

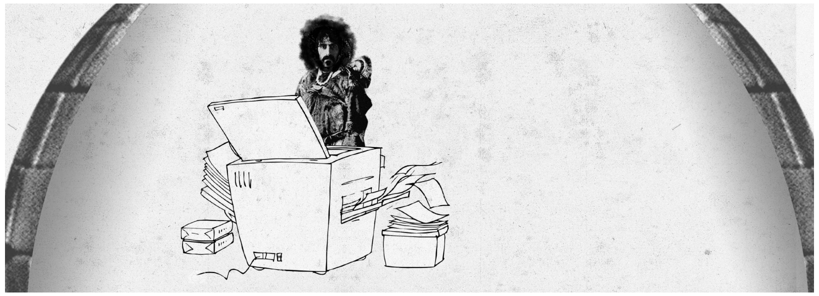
2a. If m was received earlier, manager picks it from archive. Its file will also contain the returned response z

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

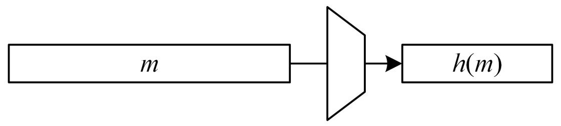
Random Oracle Inc.

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

A hash function h is secure if it behaves as Random Oracle Inc.

- Always return same result $h(m)$ for same message m
- Results $h(m)$ and $h(m')$ if $m \neq m'$ look random and unrelated

Hash function schematic and security (update)



Informal definition of hash function security

A hash function $h(\cdot)$ is secure if hash results $h(m)$ look random with no apparent relation with their input m

Let us check this with a real-world hash function

SHAKE128 is one of the hash functions defined in the NIST SHA-3 standard

Let us apply SHAKE to the message "Security: interesting!"

SHAKE("Security: interesting!") = `336113159061d2163feaf7ae12ddad58`

If we change a detail:

SHAKE("Security: interesting?") = `6548b7e6db8f86ae7f9e6d020698c5aa`

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

SHAKE(symmetric.tex) = `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*

With Python built-in hashing (versions $\geq$ 3.6 also supports SHAKE128)

```
> 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 that is heads (0) or tails (1)
  - Outcome of Alice: C_A
  - Outcome of Bob: C_B
- They 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

Commitment phase

$$\begin{array}{ll} A \longrightarrow B\colon Z_A = h(C_A \| N_A \| 1) & \text{with } N_A \text{ randomly generated by } A \\ B \longrightarrow A\colon Z_B = h(C_B \| N_B \| 0) & \text{with } N_B \text{ randomly generated by } B \end{array}$$

Revealing phase

$$\begin{array}{ll} A \longrightarrow B\colon C_A, N_A & B \text{ checks honesty of } A : Z_A \overset{?}{=} h(C_A \| N_A \| 1) \\ B \longrightarrow A\colon C_B, N_B & A \text{ checks honesty of } B : Z_B \overset{?}{=} h(C_B \| N_B \| 0) \end{array}$$

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

Coin-tossing by email: requirements

- B cannot derive C_A from Z_A
  - N_A shall be unpredictable
  - having $h(C_A \| N_A \| 1)$ shall not help in finding C_A
  - This is a kind of one-way property
  - This requirement is called preimage resistance
- A cannot find N_A, N'_A, with $h(0 \| N_A \| 1) = h(1 \| N'_A \| 1)$
  - This requirement is called collision-resistance
  - Collisions may exist if Z_A is shorter than N_A
  - Finding collision shall be computationally infeasible

h is collision-resistant if it is infeasible to find m, m' with $m \neq m'$ and $h(m) = h(m')$

Message compression for signing

Setting:

- Bob signs a contract, say in the form of a pdf document
- Classical procedure:
  - Bob inspects the contract on his PC
  - If he agrees, he prints the contract
  - . . . and puts a signature (with ink) on each page
- We want to make this more user-friendly with a *modern* procedure:
  - Bob inspects the contract on his PC
  - If he agrees, he asks the PC to compute a hash over it
  - he prints the hash (fits on one page)
  - . . . and puts a signature (with ink) on it
  - we call this the *paper commitment*

Message compression for signing: requirements I

Setting

- Alice agrees to buy house from Bob for 300K Euro
- Alice puts this in a contract m and sends to Bob for approval
- When Bob sees the contract and it is OK, they compute a hash over it and print it
- Alice and Bob arrange for a meeting with the Notary
  - Both Alice and Bob check the correctness of the printed hash
  - All three sign it and the Notary puts it in his archive
  - The paper commitment is now the legally binding document
- In case of conflict, Alice or Bob can reveal the contract m and the Notary can verify its validity

Message compression for signing: requirements II

Outsider attack

- Upon receipt of m, Bob builds m' with $h(m) = h(m')$
- m' states 500K Euro instead of 300K Euro
- Paper commitment is also valid for m'
- If Alice does not pay, Bob can sue him using m' and paper commitment as evidence

The required property is called second preimage resistance

- given m, finding m' with $h(m') = h(m)$ shall be infeasible

Message compression for signing: requirements III

Insider attack

- In advance, Alice builds m and m' with $h(m) = h(m')$
- m states 300K Euro, m' states 100K Euro
- The paper commitment is now also valid for m'
- If Bob does not sell for 100K, Alice can sue him with m' and paper commitment as evidence

The required property is collision-resistance:

- Finding m and m' with $m \neq m'$ and $h(m') = h(m)$ shall be infeasible

Notes:

- collision-resistance implies 2nd preimage resistance but not vice versa
- which one is needed depends on the attack scenario

Recap: the three traditional hash function requirements

Preimage resistance

Given z finding a message m with $h(m) = z$ shall be infeasible

Second preimage resistance

Given m, finding a message m' with $m' \neq m$ and $h(m') = h(m)$ shall be infeasible

Collision resistance

Finding m and m' with $m \neq m'$ and $h(m') = h(m)$ shall be infeasible

Password protection on servers I

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

Hashed password storage

The password file on the server looks like this:

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

- When a user logs on with his 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

Password protection on servers II

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 II

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: $h(\text{username;passwd})$
  - attempt must be of the form $h(\text{username;passwdGuess})$
  - 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: $h(\text{salt;passwd})$
  - 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
- Hash function h can be used to build two hash functions h_0 and h_1:
  - $h_0(M) = h(M\|0)$
  - $h_1(M) = h(M\|1)$
  - if h is ideal, both h_0 and h_1 are ideal
- Generalization: hash function can be used to build 2^n hash functions
  - $h_a(M) = h(M\|a)$
  - 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 2005 to < 10 Euro in 2020
- Approach: artificially slow down the hashing: *key stretching*
- Traditional solution: storing x_N computed as

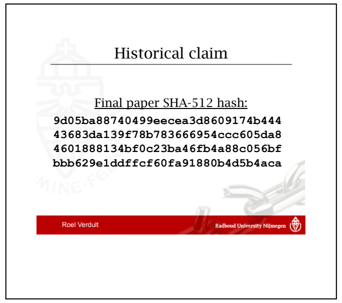
$$x_0 = 0 \quad \text{and } x_{i+1} = h(x_i\|\text{password}\|\text{salt}).$$

- 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 disk 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 $h(m') \overset{?}{=} Z$
  - 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)
  - ensuring integrity of evidence in forensic investigation, etc.
  - trusted platform module (TPM) (see later)

Originality claim for banned publication



Historical claim

Final paper SHA-512 hash:

```
9d05ba88740499eecea3d8609174b444
43683da139f78b783666954ccc605da8
4601888134bf0c23ba46fb4a88c056bf
bbb629e1ddffcf60fa91880b4d5b4aca
```

Roel Verdult

Radboud University Nijmegen

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 requirements, quantified

Traditionally, from an n-bit cryptographic hash function h one expects the following security strength:

(1) preimage resistance: given a string 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 an ideal hash function

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 $i(i-1) \approx 506 = 23 \times 22$.

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:

Collision-resistance

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

MD5 and standards SHA-1 and SHA-2

- MD5 [Ron Rivest, 1991]
  - based on MD4 that was an original design
  - 128-bit digest
  - Became de facto standard due to adoption by Silicon Valley
- SHA-1 [NIST, 1995] (after SHA-0 [NIST, 1993])
  - *designed* at NSA, mostly an improved version of MD5
  - SHA stands for *Secure Hash Algorithm*
  - 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

The MD5 saga

- 1993: weaknesses shown in internal building blocks
- 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

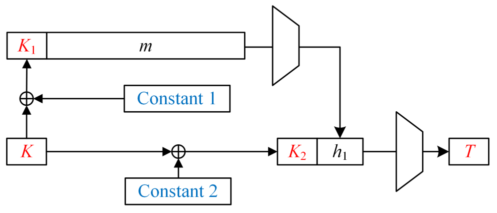
- SHA-1
  - 2004-2007: theoretical collision attacks in effort $\approx 2^{61}$
  - 2017: Marc Stevens (CWI, Amsterdam) et al. do it
  - Collisions explained at `https://shattered.io/`
  - broken but not as bad as MD5
- SHA-2 series: still a solid safety margin despite public scrutiny
  - suffer from theoretical problem: *length extension weakness*
  - like MD5 and SHA-1 did

Length extension weakness

A hash function $h(\cdot)$ has the length extension weakness if it is feasible to compute $h(m \| m')$ knowing only $h(m)$ and m'

Constructing a MAC function from MD5, SHA-1 or SHA-2

The HMAC authentication mode [FIPS 197]:



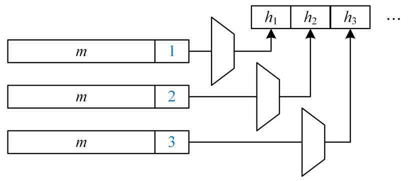
- HMAC makes two (keyed) calls to the underlying hash function
- A single call would make forgery trivially easy due to the length-extension property

Generating a stream (cipher) with MD5, SHA-1 or SHA-2

In many applications we need a long hash output

- when used for deriving multiple keys (SSL, TLS, see later)
- when using for keystream generation, ...

The mode MGF1 [PKCS #1]:



Stream cipher by taking $m = K \| D$ with D the diversifier

- $Z_i = h_i = h(K \| D \| i)$
- this is similar to counter mode of a block cipher

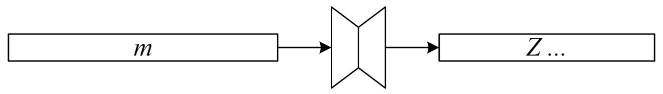
SHA-3: the competition

- 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
  - traditional collision and (2nd) pre-image resistance required
  - 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 publishes the SHA-3 standard: FIPS 202

The SHA-3 standard: FIPS 202

- Length extension problem has been fixed
- FIPS 202 specifies 6 functions in total, all independent
- Four hash functions with same output lengths as SHA-2 equivalents
  - SHA3-224
  - SHA3-256
  - SHA3-384
  - SHA3-512
- novelty: *extendable output functions* (XOF)
  - hash function that can generate output of arbitrary length
  - user determines output length
- Two XOFs:
  - SHAKE128: XOF with inherent security strength of 128 bits
  - SHAKE256: XOF with inherent security strength of 256 bits
- Designers recommend using SHAKE128 for everything

XOF schematic and security



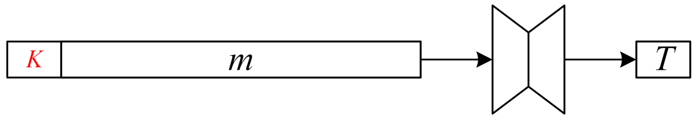
Informal definition of XOF security

A XOF $h(\cdot)$ has an inherent security strength s if it offers the same resistance as an ideal hash function against attacks with workload below 2^s computations

Security strength of a XOF is determined by an internal design parameter usually called *capacity*

Constructing a MAC function from SHAKE128

Just take as input concatenation of key K and message m

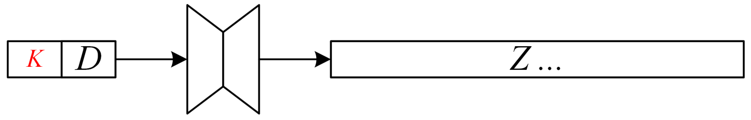


$F(K, M)$ constructed as $\text{XOF}(K \| M)$

Truncate output to desired MAC length

Constructing a stream cipher from SHAKE128

Just take as input concatenation of key K and diversifier D



$F(K, D)$ constructed as $\text{XOF}(K \| D)$

Use output as keystream, as long as you need

Conclusions

- Hash functions are versatile: compression, encryption, MAC, key derivation ...
- A hash function is secure if it behaves like Random Oracle Inc. For n-bit output:
  - generating (2nd) pre-image takes 2^n hash attempts
  - generating collision takes $2^{n/2}$ hash attempts
- Multiple hash functions from a single one by domain separation
- Legacy standard hash functions
  - MD5 and NIST standard SHA-1: broken
  - SHA-2: same philosophy, but still very solid
- SHA-3:
  - very solid
  - one function for all output lengths: SHAKE128
  - simplification of modes of use