Computer Security: Hashing

B. Jacobs

Institute for Computing and Information Sciences – Digital Security
Radboud University Nijmegen

Version: fall 2015

Outline

Hash essentials

- A hash function, often written as h, takes an arbitrary message m and yields an outcome $h(m)$ of fixed length
  Formally,
  $$h\colon \{0,1\}^{\star} \longrightarrow 2^{N} \qquad \text{typically for } N = 128, 160, 256.$$
- Intuitively, $h(m)$ is a garbled version of m, from which one cannot reconstruct m
- $h(m)$ is called the hash (value) of m. Alternative names:
  - message digest
  - (cryptographic) fingerprint
  - Dutch: *verhaspeling*
- A hash is a simple but surprisingly powerful crypto primitive

Hash examples (with `md5sum`)

Applying the hash function md5 to the message

Security is hot

yields the 32 hexadecimal (128 bit) value:

`d6bbdb97f1ac18dec78ac2847d8906f0`

Changing a minor thing yields a completely different outcome:

md5(“Security is hit”) = `c3e9121b600e29736583242a53f8cbd7`

The hash value of (the current 30765 byte version) of this .tex document is: `a1084ca86fe7b77c2d0929e923298815`.

This can be used as fingerprint of the document! Why?

Hash yourself!

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

- `md5sum` *file*
- `openssl md5` *file*

Or, similarly:

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

(Later we shall see hashing in Java)

Protocol with hash example, set-up

- Suppose A and B decide via a phone who has to cook dinner tonight, using coins
- They each toss a coin, and agree:
  - if the outcomes are equal, A prepares the dinner
  - otherwise B does
- How to do this securely, without the possibility to cheat?

  (and without a trusted third party, TTP)

Protocol with hash example, solution

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

$A \longrightarrow B\colon h(C_A, N_A)$	N_A is a nonce chosen by A
$B \longrightarrow A\colon h(C_B, N_B)$	N_B chosen by B
$A \longrightarrow B\colon C_A, N_A$	B checks honesty of A
$B \longrightarrow A\colon C_B, N_B$	A checks honesty of B
	Both can check $C_A \overset{?}{=} C_B$.

Hashing is used here for non-revealing commitment

Why are the nonces necessary? Is the hash in the second message needed?

Properties of hash functions, informally

A “good” hash function should be such that it is difficult (computationally infeasible) to:

1. invert
2. find a second input that hashes to a given hash value
3. find two inputs with the same hash value

Not all properties are needed at the same time in each application. Which properties are used in the coin-protocol?

Because of the finite output 2^N, collisions are inevitable; the important issue is that collisions should not be producable.

Required properties of hash functions, more precisely

A (good, cryptographically secure) hash function h should be:

1. one-way (preimage resistant): given a hash value x, it is difficult to find an m with $h(m) = x$
2. second preimage resistant: given m and thus $h(m)$, it is difficult to find $m' \neq m$ with $h(m) = h(m')$
3. collision resistant: it is difficult to find *any* pair $m \neq m'$ with $h(m) = h(m')$.

Hash function for message integrity

Recall the earlier "hash" version to realise integrity of transfer:

$$A \longrightarrow B\colon\ m, K_{AB}\{h(m)\}$$

Questions:

- Why does this version with hash function *h* also work?
- What is the main advantage of including *h*?
- Which properties of *h* are used?

Hash function implementations

- The basis for hashing is a one-way function
- Intuitive example of one-way computation on 100-bit words:

  Take a 100-bit word/number as input, and square it, giving a 200-bit number. Now take the middle 100 bits as output.

- Easy to compute, but is clearly intuitively one-way:
  - given a 100 bit number, finding the preimage/original is difficult
  - there may be several originals (clashes)
- Standard hash functions have publicly known definitions—as usually in crypto.
- NIST recently ran a 5-year competition for a new hash function, see `http://csrc.nist.gov/groups/ST/hash/sha-3`
  - Won by Keccak ("catch-ack"), from Belgium, like AES

Some well-known hash functions

- MD5 with 128 bit output length, designed by Rivest. Now considered insecure, esp. not collision-resistant (shown by Xiaoyun Wang et al).
    - Collisions found for different executables (one malicious)
    - Also for different certificates
- SHA-1 with 160 bit, also broken (by Wang et al)
- SHA-256 or SHA-512 are currently recommended—for the time being.
- SHA-3 = Keccak, new standard since oct.'2012

Predicting the future with broken hash functions

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

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

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

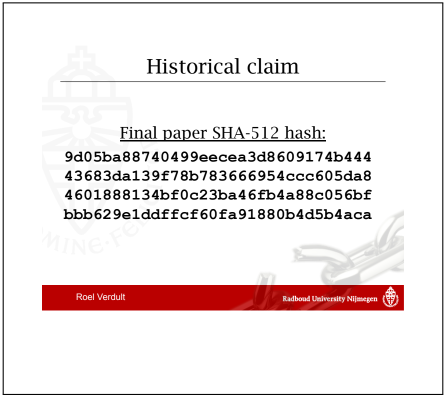
with the same hash: $\mathbf{md5}(m_1) = \mathbf{md5}(m_2)$.

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

Problem: **md5** is not collision-resistant, so it cannot be used for commitment.

(Malware Flame also uses **md5** collisions to create counterfeit Microsoft update certificates.)

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.

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 $h(m)$ locally
- After retrieving the file, say m', you compute $h(m')$ and compare it to $h(m)$
    - if $h(m) = h(m')$ you can be fairly sure that $m' = m$.
- The same technique is used in many other situations, e.g.
    - Downloading software (hash must be stored elsewhere, or be signed)
    - Protecting evidence in forensic investigation, etc.

Hash application: finding child pornography

- Looking for child porn on confiscated computers can be emotionally stressful for police investigators
- Therefore, the Dutch police has compiled a large collection of hashes of known child porn pictures
- Hence the investigation can be automated:
    - calculate hashes of confiscated pictures, and compare results to this data base (which hash property is used?)
    - they even developed a USB stick from which this can be run
- If you wish to remain undetected: change at least one pixel in all your porn material!

Hash application: one-time-pad generation

- Recall that the main disadvantages of one-time-pads are:
  - the key must be at least as long as the message
  - the key should not be re-used
- Possible solution: generate key stream p from fixed length key K, for instance as:

$$p = p_1, p_2, p_3, \ldots$$

  where for instance:

$$p_0 = h(K), \qquad\qquad p_{n+1} = h(K, p_n).$$

- Which properties of h are used?
- Why is doing $h(K), h(h(K)), h(h(h(K))), \ldots$ not wise?

Hash application: unreliable guards

- Consider border guards who have to recognise their own spies coming through occasionally from the other side.
- The spies are highly trained and trusted; the guards are unreliable (they talk too much in the local pubs)
- Solution: give the guards a list of identifiers s for spies together with a corresponding hash value $y_s = h(x_s)$.
- When a spy reports in, (s)he has to tell s and the corresponding x_s. The guard can then compute $h(x_s)$ to check if the spy is genuine.
- If the list of pairs (s_i, y_{s_i}) gets compromised in the local pub, the system still works. Why? Because of which hash property?

Hash application: password storage

- It is not wise to store user passwords on a computer in the clear:
  - other users (administrators) may abuse them
  - they may be stolen after computer intrusion
- The common solution is to store hashes of passwords
  - after entering a password, the computer calculates the hash and compares it to the data base entry of the user
- Remaining attacks:
  - online: restrict number of attempts, or slow down progressively after repeated attempts
  - offline (or dictionary): serious risk, esp. for weak passwords

Hash application: password storage with salt

A so-called salt is used to prevent “uniform” dictionary attacks on a computer’s password file: when a different (known) value is added in each hash, an attacker is slowed down because she has to compute $h(\mathrm{salt}_i, \mathrm{attempt})$ for each entry i in the password file.

The password file then has the following structure.

user	**salt**	**hash**
bart	bla	$h\big(\mathrm{bla}, \mathrm{passwd}\big)$
peter	aap	$h\big(\mathrm{aap}, \mathrm{passwd}\big)$
$\vdots$	$\vdots$	$\vdots$

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

Hash application: stretched passwords

- The password p can be "stretched", by complicating the computation of the value x_N that is stored, via:

$$x_0 \quad = \quad 0 \qquad \text{and} \qquad x_{n+1} \quad = \quad h(x_n, p, \mathsf{salt}).$$

- The number N should be such that computing x_N takes 200-1000 msec on the user's equipment.
- The combination of salt and stretching is implemented in the function MD5 crypt
  - it hashes the password and salt in a number of different combinations to slow down the evaluation speed
  - it is not broken (like MD5), because of the repetition

Lamport's hash

The computer C now stores for each user A in the password file a pair $\langle n \in \mathbb{N}, h^n(\text{passwd}_A)\rangle$ for some $n \neq 0$.

$$\begin{array}{l}
A \longrightarrow C: \text{I'm Alice} \\
C \longrightarrow A: n \\
A \longrightarrow C: \; h^{n-1}(\text{passwd}_A) = x \\
\qquad\qquad \text{Compare } h(x) \text{ and stored value } h^n(\text{passwd}_A), \\
\qquad\qquad \text{and, if equal, grant access and store new pair } \langle n-1, x\rangle
\end{array}$$

Note:

- Login credential is different each time
- Set-up with $n = 10.000$, say; what if $n = 0$?
- A should be able to compute hashes; humans need to use a separate device (like in e-banking).

Birthday attack

- With how many people is the chance bigger than $\frac{1}{2}$ that (precisely) two of them have the same birthday?

  Answer: 23 (see `en.wikipedia.org/wiki/Birthday_problem`)

- Upshot: collisions occur much faster than you would expect. If an element can take on N different values, then you can expect a first collision after choosing about $\sqrt{N}$ random elements

A 50% chance of collision for n-bit hash: only $\sqrt{2^n} = 2^{\frac{n}{2}}$ trials

E.g. for the 128-bit MD5 hash, one can expect a collision after 2^{64} tries.

Birthday attack: explanation of square root

What is the chance that . . .

- two arbitrary bits coindice: $\frac{1}{2}$
- that two k-bit words coincide: $\left(\frac{1}{2}\right)^k = \frac{1}{2^k} = 2^{-k}$
- a k-bit word coincides with a k-bit word out of a set of N words: $N \cdot 2^{-k}$
- two k-bit words out of a set of N coincide: $\frac{N\cdot(N-1)}{2} \cdot 2^{-k}$

When is this (last) chance at least $\frac{1}{2}$, roughly?

$$\begin{aligned} \frac{N\cdot(N-1)}{2} \cdot 2^{-k} > \frac{1}{2} \quad & \overset{\text{roughly}}{\Longleftrightarrow} \quad N^2\, 2^{-k} > 1 \\ & \Longleftrightarrow \quad N^2 > 2^k \\ & \Longleftrightarrow \quad N > \sqrt{2^k} = 2^{k/2} \end{aligned}$$

Sensitivity of (electronic) voting

**"It's not the people who vote that count.
It's the people who count the votes."**

Attributed to Joseph Stalin

Electronic voting

- Voting involves requirements that are hard to combine:
  - transparancy
  - verifiability
  - accessibility
  - secrecy of individual vote
  - admit only elligible voters
  - at most one individual vote
- It is a popular topic in security research
- Important distinction in e-voting
  - using computers in poll station
  - internet voting
- We shall look at one example of a simple Dutch e-voting system (RIES), that uses hashes and symmetric encryption
- For more info, read *Electronic Voting in the Netherlands: from early Adoption to early Abolishment*

Background

- RIES = Rijnland Internet Election System
  - Rijnland: Dutch authority for water management
- Goals
  - Simple, cheap but reasonably secure internet voting system
  - Increase election turnout
- System should be at least as secure as their older ordinary mail voting system
- Independent audits by TNO, Cryptomathic, SURFnet, Madison Gurkha, RU & TU/e, Fox-IT
- RIES was withdrawn/abolished in 2008
  - still, it forms a nice, large scale application of hashing

RIES actual use

- In 2004/5 for regional waterboard elections (with $> 1M$ potential voters; $100 - 200K$ actual)
- 2006: parliament elections, for expats ($\pm 20K$ voters)
- 2008: intended for joint regional waterboards
    - But not deployed due to (action group) opposition and security vulnerabilities
- Among the largest, actually used e-voting systems, worldwide
- Produced valueable experience about how (not) to run medium/large internet elections

The RIES System

- Designed (and patented) by Maclaine Pont
  - Based upon mastersthesis by Robers (1998)
- Clever but elementary use of hashes
  - h = MDC = Modification Detection Code
    - key-less hash (128 bit, designed by IBM)
  - MAC = Message Authentication code
    - hash with personal secret key
    - it acts as encryption here, and will be described as $K\{-\}$
- RIES is transparent:
  - "Pre-election" and "post-election" tables imply verifiability
  - not only of personal vote, but also of votes of others

The RIES System: main idea

- Each voter i gets a secret key K_i
  - keys are generated in advance, not connected to voters
  - practically, key is printed on (non-personal) invitation to vote
  - key must be entered via a webpage
  - crypto calculations done in browser, in Java Script
- Each candidate j has identity C_j
- Pre-election table contains all combinations $C_j \leftrightarrow h(K_i\{C_j\})$
  - table is published before election
- Voter i sends pre-image $K_i\{C_j\}$ of such hash
  - Only voter i can generate pre-image involving personal key K_i
- Assume vote server receives vote $v = K_{350}\{C_5\}$
  1. it looks for the hash $h(v)$ in the pre-election table
  2. if found, the corresponding candidate $C_5 \leftrightarrow h(v)$ gets a vote

Verification of individual vote

After the election all received votes $v_i = K_i\{C_{j_i}\}$ are published in a post-election table

Voter *i* can then check own vote

1. Looking up own vote $v_i = K_i\{C_{j_i}\}$ in post-election table
2. computing $h(v_i)$ and looking up this value in pre-election table
3. Checking corresponding candidate C_{j_i}

Verification of outcome

Anyone can check total outcome

1. Collect all votes $v_1, \ldots, v_n$ from post-election table
2. Compute hash on each vote $h(v_1), \ldots, h(v_n)$
3. Look up each hashed vote $h(v_i)$ in pre-election table, with corresponding candidate
4. Add up all resulting votes, for each candidate.

Nijmegen's Digital Security group performed these checks for actual RIES elections and confirmed the official outcome

Downfall of RIES

- Fundamental design flaw: organisers of the election can (in principle) vote on behalf of everyone
  - hence many organisational controls needed
  - do you trust the key generators?
- Similarly, printer & distribution company can link voters and keys, and thus break secrecy
- June'08: open source release showed vulnerabilities (like SQL injection, found by Gonggrijp)
- Brute force vulnerabilities in crypto
  - Personal keys are only 56 bits long (usability compromise)
  - Fox-IT showed: only 20 hours needed to get such key K_i from pre-election table entry $h(K_i\{C_j\})$
- July'08: ministry decides not to allow RIES any longer!

Variations in road pricing

- Zone-based
  - for instance in London & Stockholm
  - based on Automatic Number Plate Recognition (ANPR)
- Point-to-point
  - on motorways in France, Italy, ...
  - via (electronic) gates
  - since 2005 in Germany for trucks (*LKW-Maut*, via DSRC)
- Pay-as-you-drive
  - Advanced plans in NL aborted (for now); possibly elsewhere (Be, EU, ...)
  - Satellite-based (GPS, Galileo)

Pay-as-you-drive road pricing

- Replaces “flat road tax” by “distance related pricing”
- Pricing may depend on:
    - type of road
    - type of car (esp. emission characteristics)
    - time of day (esp. rush hour, via *spitstarief*)
    - location
- Aims, apart from fairness,
    - congestion steering/reduction
    - environmental impact reduction
- More refined steering & control possible than with fuel price.

Issues in road pricing

- Reliability
- Cost-effectivity (aim in NL: overhead $< 10\%$)
- Ease of use / transparancy
- Fraud resistance (e.g. GPS can be manipulated/shielded, power supply can be interrupted, ...)
- Ease of enforcement
- Ease of dispute resolution
- Security (protection against attacks, manipulation, ...)
- Privacy
- User acceptance, requiring trust!

There will be many hostile users

(Think of tachometer fraud by truck drivers)

Road pricing: technical set-up

- Cars get a special box, called OBE, for "on-board equipment", or in Dutch: *registratievoorziening*.
- ... which can at least:
  - determine its own position, via GPS or Galileo
  - communicate with backoffice, via GSM, GPRS, Wifi, ...
  - calculate & store data
- Tariff map needed for fee calculation on basis of "trajectory parts"

Big Question

- Where to store (privacy-sensitive) trajectory information?
  - in the back-office of the authorities / service providers (who use it for billing and/or marketing/profiling)
  - in the vehicle, i.e. in the OBE (so OBE contains map-data for aggregation)
- This is an architectural decision about information flow
- But also about division of power in society (balance citizen – state)

Architecture is politics
(M. Kapor, EFF)

Centralised $\leftrightarrow$ decentralised architectures

- Centralised (think: in the cloud)
  - Data outside user control: privacy depends heavily on organisational measures
  - Single point of failure makes system vulnerable
  - Convenience for user
  - Easier maintance & policy enforcement
  - Informational control leads to societal control (profiling/datamining)
- Decentralised (think: on own device)
  - Privacy-friendly, in-context storage of data
  - More responsibility/activity on user side required
  - Fraud resistance possibly more difficult

Question: which architecture more secure?

Privacy requirements articulated in NL plans

- Car owner has access to own location data, via OBE.
- Authorities possess only:
  - aggregated data used for billing
  - enforcement data (photos, communication messages)

  These data are stored for at most 5 years.
- Commercial service providers may store & use location data, but only after explicit permission of client

Overview: three OBE names

1. Thin (*dun*)
   - OBE sends all location data to central server
   - likely preference of commercial parties
2. Fat (*dik*)
   - OBE aggregates itself
   - was forseen in minstery's track (*garantiespoor*)
3. Well-rounded (*volslank*)
   - OBE sends only **hashes** to central server

1. Thin OBE: essentials

- OBE activities restricted to:
  - calculation of trajectories
  - passing on these trajectories to the back-office, say every minute
- OBE does not aggregate
- Easy enforcement via passive spot checks: take photo and compare it (later) to location data sent to back-office

1. Thin OBE: pros and cons

- + Simple and transparant architecture & simple and cheap OBEs
- + Failure of physical OBE protection not catastrophic
- +/− Central storage enables (real-time) location-based ‘services’ (but also additional checks, like speed checks)
- − Much communication (cost) involved
- − Privacy only procedurally protected, depending on policy of service provider
- − Central database introduces risks:
  - data compromise may embarass people (look for politicians who visited prostitute areas)
  - data protection relevant for personal security (e.g. whereabouts of people under threat)
  - single point of failure / bottleneck
  - (real-time) road tap possibility

2. Fat OBE: essentials

- OBE aggregates itself, and passes only aggregated data on to the back-office
  (For instance: NL is divided into red, green, blue . . . roads, each with their own tariff; the OBE communicates, say every month, how many kilometres have been driven on which colour, in which time segment.)
- OBE must thus contain map-data & timing for aggregation
  (which must be securely updated, occasionally)
- OBE must contain trusted element (smart card), for secure storage, communication & updates
- Spot checks are non-passive and complicated:
  - Two-way communication, while driving by
  - requesting most recent trajectory data
  - noticable, and likely to generate warning to other drivers

2. Fat OBE: pros and cons

+ Privacy technically protected, via decentralised storage and aggregation
– Complicated and expensive OBE
– OBE must be fully trusted: succesful (physical) attack on OBE is catastrophic
– Complicated, non-passive spot checks

3. Well-rounded OBE: essentials

- OBE regularly sends hashes of its trajectory parts to the back-office
- These hashes reveal nothing, but commit the OBE/car
- Spot check can be passive, via photo: OBE must later show that spot check location was in pre-image of a hash in the back-office
- Fee calculation can be done by anyone: OBE, PC of car owner, (several) service providers, etc.
- Fee verification can also be done “locally”
  (details omitted here)

3. Well-rounded OBE: hash details

Each day d, there is a message:

$$\textbf{OBE} \longrightarrow \textbf{BackOffice} : \langle \text{vehicle-id}, d, \text{hash-of-the-day}_d \rangle$$

This hash-of-the-day$_d$ is a nested hash message:

$$\text{hash-of-the-day}_d = h\Big(h(\text{TP}_{d,1}) \mid \cdots \mid h(\text{TP}_{d,1440})\Big)$$

where

$$\text{TP}_{d,i} = \text{trajectory part during minute } i \text{ on day } d$$

(Each day has $24 \cdot 60 = 1440$ minutes, so $1 \leq i \leq 1440$)

This hash-of-the-day is a short message, say 256 bits, which completely fixes the trajectory of the day. It is a non-revealing commit

3. Well-rounded OBE: road side checks

- Suppose a certain vehicle is photographed during minute i at day d at location p.
- After the vehicle's OBE has sent in the hash-of-the-day (for d), the authorities can:
  - ask for the preimage $h(\mathrm{TP}_{d,1}) \mid \cdots \mid h(\mathrm{TP}_{d,1440})$ of the outer hash (this reveals nothing, yet)
  - select the relevant hash $h(\mathrm{TP}_{d,i})$, by counting bits, and ask for its preimage
  - upon receiving this trajectory part,
    - check the hash
    - check that the photo location p is in this trajectory part
- This may look complicated, but can be fully automated

3. Well-rounded OBE: pros and cons

+ Privacy technically protected
+ Flexible approach,
  - allowing many different realisations, with/without commercial service providers
  - allows (inter)nationally uniform system (including spot checks) with different options chosen by clients
+ Breakdown of physical OBE protection is not catastrophic

+/− Spot checks easy & (necessarily) passive, but verification requires careful timing (after all hash commits) and explicit revealing action

+/− Requires open standard for trajectory parts
(proprietary in many current GPS systems)

− Difficult to explain to general audience

Road pricing conclusions

- A little crypto can give a lot of privacy . . .
- even after a few lectures only!
- More information in: W. de Jonge and B. Jacobs, *Privacy-friendly Electronic Traffic Pricing via Commits* `http://www.tipsystems.nl/files/ETPprivacy.pdf`
- This is also an active research area, with several alternative solutions.

Security basics & hash in Java

- Java provides extensive support for secure programming (see later), including several libraries:
  - `java.security.*` (used here)
  - bouncy castle, . . .
- Java is very verbose, but provides good abstraction
- For hashing there is the `MessageDigest` class with operations
  - `MessageDigest.getInstance("MD5")` : creates the message digest.
  - `.update(plaintext)` : calculates the hash with a plaintext string.
  - `.digest()` : reads the hash

Hashing in Java: code snippet

```
MessageDigest md =
    MessageDigest.getInstance("SHA1");
String s = "Hash that string";
md.update(s.getBytes());
byte[] hashvalue = md.digest();
```

Hashing in Python

This is a little bit less verbose:

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