Cryptographic Hash Functions
Part I

2MMC10 Cryptology

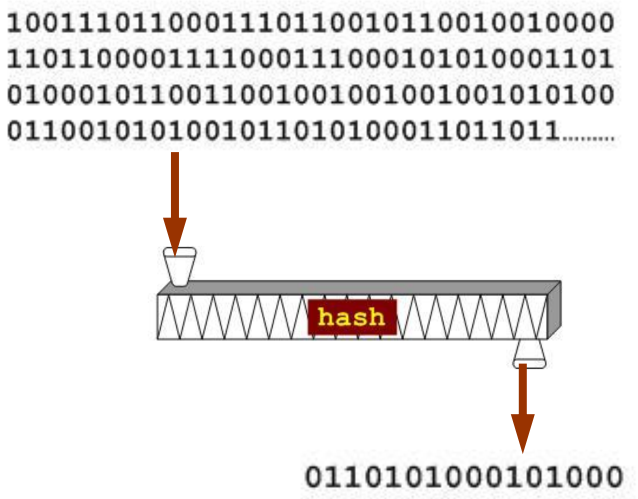
Andreas Hülsing

How are hash functions used?

- integrity protection
  - cryptographic checksum (e.g. software downloads)
  - for file system integrity (Bit-torrent, git)
- password hashing
  - dedicated algorithms like scrypt / argon2 use hash functions as building block
- MAC – message authentication codes
- Digital signature (“public key MAC”)
- Password-based key derivation
- Pseudo-random number generation (PRG)
- …

What is a hash function?
- Applied answer

- **Function** $\mathbf{h}: \{0,1\}^* \to \{0,1\}^n$
- Input: bit string *x* of arbitrary length
  - length may be 0
  - in practice a very large bound on the length is imposed, such as 2^{64} (≈ 2.1 million TB)
  - input often called the *message*
- Output: bit string *h*(*x*) of fixed length *n*
  - e.g. *n* = 128, 160, 224, 256, 384, 512
  - *compression*
  - output often called *hash value*, *message digest*, *fingerprint*
- *h*(*x*) is efficiently computable given *x*
- no secret information, no secret key



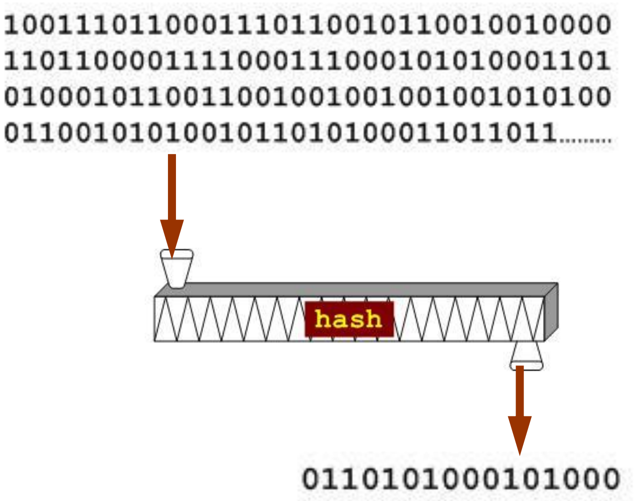
Intermezzo: Formal treatment

- Efficient Algorithm
  - Runs in polynomial time,
    i.e. for input of length n, $t_A \leq n^k = poly(n)$ for some constant k
- Probabilistic Polynomial Time (PPT) Algorithm:
  - Randomized Algorithm
  - Runs in polynomial time
  - Outputs the right solution with some probability
- Negligible: "Vanishes faster than inverse polynomial"
  We call $\boldsymbol{\varepsilon}(\mathbf{n})$ negligible if

$$(\exists \boldsymbol{n_c} > \mathbf{0})(\forall \boldsymbol{n} > \boldsymbol{n_c}): \boldsymbol{\varepsilon}(\mathbf{n}) < \frac{\mathbf{1}}{\boldsymbol{poly}(\boldsymbol{n})}$$

What is a hash function?
- Formal answer

- Efficient keyed function
  h: $\{0,1\}^n \times \{0,1\}^{l(n)} \rightarrow \{0,1\}^n$
- We write $\mathrm{h}(\mathrm{k}, \mathrm{x}) = h_k(x)$
- Key k in this case is public information. Think of function description.



Security properties: Collision resistance

Collision resistance (CR): For any PPT adversary A, the following probability is negligible in n:

$$Pr[k \leftarrow_R \{0,1\}^n, (x_1, x_2) \leftarrow A(k):$$

$$h_k(x_1) = h_k(x_2) \wedge (x_1 \neq x_2)]$$

Security properties:
Preimage resistance / One-wayness

Preimage resistance (PRE): For any PPT adversary A, the following probability is negligible in n:

$$Pr[k \leftarrow_R \{0,1\}^n, x \leftarrow_R \{0,1\}^{l(n)}, y \leftarrow h_k(x),$$

$$x' \leftarrow A(k, y): h_k(x') = y]$$

Formal security properties: Second-preimage resistance

Second-preimage resistance: For any PPT adversary A, the following probability is negligible in n:

$$Pr[k \leftarrow_R \{0,1\}^n, x \leftarrow_R \{0,1\}^{l(n)}, x' \leftarrow A(k, x):$$

$$h_k(x) = h_k(x') \wedge (x \neq x')]$$

Reductions

- Transform an algorithm for problem 1 into an algorithm for problem 2.
- “Reduces problem 2 to problem 1“
  (I can solve problem 2 by solving problem 1)
- Allows to relate the hardness of problems:

  If there exists an efficient reduction that reduces problem 2 to problem 1 then an efficient algorithm solving problem 1 can be used to efficiently solve problem 2.

Reductions II

Use in cryptography:

- Relate security properties
- „Provable Security“: Reduce an assumed to be hard problem to breaking the security of your scheme.
- Actually this does not proof security! Only shows that scheme is secure IF the problem is hard.

(Intuition: It shows, I can solve my problem by breaking the security of the scheme)

Relations between hash function security properties

Easy start: CR -> SPR

Theorem (informal): If h is collision resistant then it is second preimage resistant.

Proof:

- By contradiction: Assume A breaks SPR of h then we can build a reduction M^A that breaks CR.
- Given key k, M^A first samples random $x \leftarrow \{0,1\}^{l(n)}$
- M^A runs $x' \leftarrow A(k, x)$ and outputs (x, x')
- M^A runs in approx. same time as A and has same success probability. -> Tight reduction

SPR -> PRE ?

Theorem (informal): If h is second-preimage resistant then it is also preimage resistant.

Proof:

- By contradiction: Assume A breaks PRE of h then we can build a reduction M^A that breaks SPR.
- Given key k, x, M^A runs $x' \leftarrow A(k, h_k(x))$ and outputs (x, x')
- M^A runs in same time as A and has same success probability.

Do you find the mistake?

SPR -> PRE ?

Theorem (informal): If h is second-preimage resistant then it is also preimage resistant.

Counter example:

- the *identity function id* : $\{0,1\}^n \rightarrow \{0,1\}^n$ is SPR but not PRE.

SPR -> PRE ?

Theorem (informal): If h is second-preimage resistant then it is also preimage resistant.

Proof:

- By contradiction: Assume A breaks PRE of h then we can build an oracle machine M^A that breaks SPR.
- Given key k, x, M^A runs $x' \leftarrow A(k, h_k(x))$ and outputs (x, x')

We are not guaranteed that $x \neq x'$!

- M^A runs in same time as A and has same success probability.

Do you find the mistake?

SPR -> PRE ?

Theorem (informal, corrected): If h is second-preimage resistant, $l(n) \gg n$, then it is also preimage resistant.

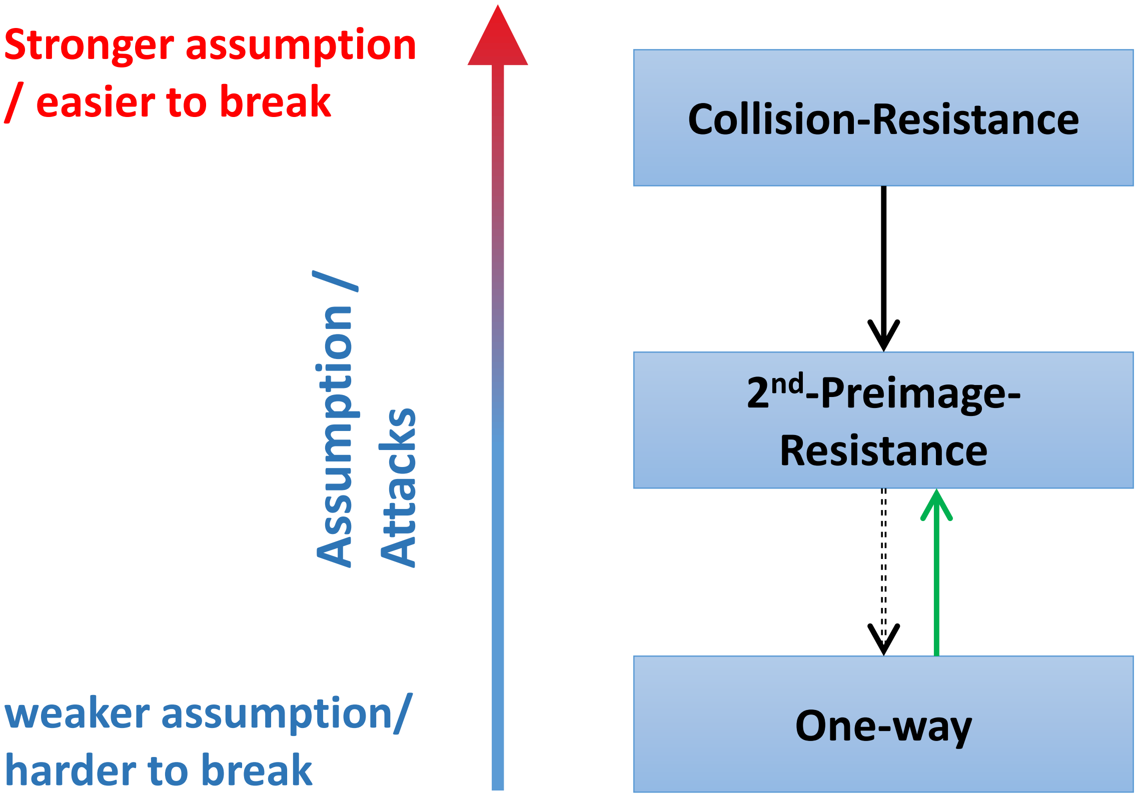
Proof:

- By contradiction: Assume A breaks PRE of h then we can build an oracle machine M^A that breaks SPR.
- Given key k, x, M^A runs $x' \leftarrow A(k, h_k(x))$ and outputs (x, x')
- M^A runs in same time as A and has at least half the success probability.

Same corrections have to

Can replace condition $l(n) \gg n$ by requiring that h is "decisional second preimage resistant".

Summary: Relations



Stronger assumption / easier to break

Assumption / Attacks

weaker assumption/ harder to break

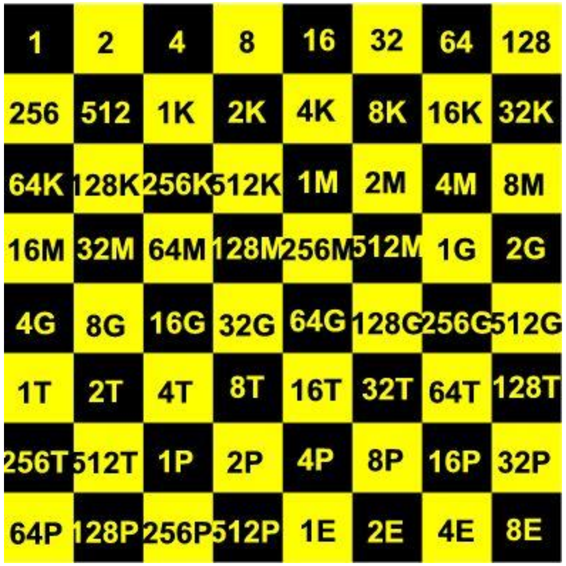
Collision-Resistance

2nd-Preimage-Resistance

One-way

generic (brute force) attacks

- assume: hash function behaves like random function
- **preimages and second preimages can be found by random guessing**
  search space: $\approx n$ bits, $\approx 2^n$ hash function calls
- **collisions can be found by birthdaying**
  - **search space: $\approx \frac{1}{2}n$ bits,**
    $\approx 2^{\frac{1}{2}n}$ hash function calls
- **this is a big difference**
  - **MD5 is a 128 bit hash function**
  - **(second) preimage random search:**
    $\approx 2^{128} \approx 3\text{x}10^{38}$ MD5 calls
  - **collision birthday search: only**
    $\approx 2^{64} \approx 2\text{x}10^{19}$ MD5 calls



1	2	4	8	16	32	64	128
256	512	1K	2K	4K	8K	16K	32K
64K	128K	256K	512K	1M	2M	4M	8M
16M	32M	64M	128M	256M	512M	1G	2G
4G	8G	16G	32G	64G	128G	256G	512G
1T	2T	4T	8T	16T	32T	64T	128T
256T	512T	1P	2P	4P	8P	16P	32P
64P	128P	256P	512P	1E	2E	4E	8E

birthday paradox

- birthday paradox

  given a set of t ($\geq\ 10$) elements

  take a sample of size k (drawn with repetition)

  in order to get a probability ≥ ½ on a collision

  (i.e. an element drawn at least twice)

  k has to be $>\ 1.2\sqrt{t}$
- consequence

  if $F : A \rightarrow B$ is a surjective random function

  and $|A| \gg |B|$

  then one can expect a collision after about $\sqrt{|B|}$ random function calls

meaningful birthdaying

- random birthdaying
  - do exhaustive search on $n/2$ bits
  - messages will be ‘random’
  - messages will not be ‘meaningful’
- Yuval (1979)
  - start with two meaningful messages m_1, m_2 for which you want to find a collision
  - identify $n/2$ independent positions where the messages can be changed at bit level without changing the meaning
    - e.g. tab ←→ space, space ←→ newline, etc.
  - do random search on those positions

implementing birthdaying

- naïve
  - store $2^{n/2}$ possible messages for m_1 and $2^{n/2}$ possible messages for m_2 and check all 2^n pairs
- less naïve
  - store $2^{n/2}$ possible messages for m_1 and for each possible m_2 check whether its hash is in the list
- smart: Pollard-ρ with Floyd's cycle finding algorithm
  - computational complexity still $O(2^{n/2})$
  - but only constant small storage required

Pollard-ρ and Floyd cycle finding

- Pollard-ρ
  - iterate the hash function: a_0, $a_1 = h(a_0)$, $a_2 = h(a_1)$, $a_3 = h(a_2)$, …
  - this is ultimately periodic:
    - there are minimal t, p such that $a_{t+p} = a_t$
    - theory of random functions: both t, p are of size $2^{n/2}$
- Floyd's cycle finding algorithm
  - Floyd: start with (a_1,a_2) and compute (a_2,a_4), (a_3,a_6), (a_4,a_8), …, (a_q,a_{2q}) until $a_{2q} = a_q$; this happens for some $q < t + p$