Cryptographic Hash Functions

Hashing

Basic idea: a hash function maps "long" strings from some set $\mathcal{M} = \{0,1\}^{\leq L}$ onto short strings from some set $\mathcal{H} = \{0,1\}^{\ell}$, where $\ell << L$.

Cryptographic hash functions posses some sort of one-way and collision resistance properties: given just a hash of a message, it is difficult to compute the message itself, or to compute two messages that hash to the same string.

- Unkeyed hash functions: $h\colon \mathcal{M} \to \mathcal{H}$
  - message/file integrity
  - password storage
  - digital signatures
  - ...
- Keyed hash functions, aka MACs: $h\colon \mathcal{M} \times \mathcal{K} \to \mathcal{H}$
  - message integrity and authenticity

Hashing

Basic idea: a hash function maps "long" strings from some set $\mathcal{M} = \{0,1\}^{\leq L}$ onto short strings from some set $\mathcal{H} = \{0,1\}^{\ell}$, where $\ell << L$.

Cryptographic hash functions posses some sort of one-way and collision resistance properties: given just a hash of a message, it is difficult to compute the message itself, or to compute two messages that hash to the same string.

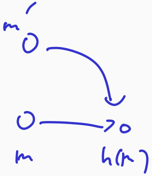
- Unkeyed hash functions: $h \colon \mathcal{M} \to \mathcal{H}$
  - message/file integrity
  - password storage
  - digital signatures
  - ...
- Keyed hash functions, aka MACs: $h \colon \mathcal{M} \times \mathcal{K} \to \mathcal{H}$
  - message integrity and authenticity

From now on: hash function = unkeyed hash function, MAC = keyed hash function

Cryptographic hash functions

Desired properties of cryptographic hash functions:

- h is one-way (or 1st preimage resistant) if, given $h(m)$, it is difficult to compute m
- h is 2nd preimage resistant if, given m and $h(m)$, it is difficult to compute $m' \neq m$ s.t. $h(m') = h(m)$
- h is collision resistant if it is difficult to compute m, m' s.t. $m' \neq m$ and $h(m') = h(m)$



Cryptographic hash functions

Desired properties of cryptographic hash functions:

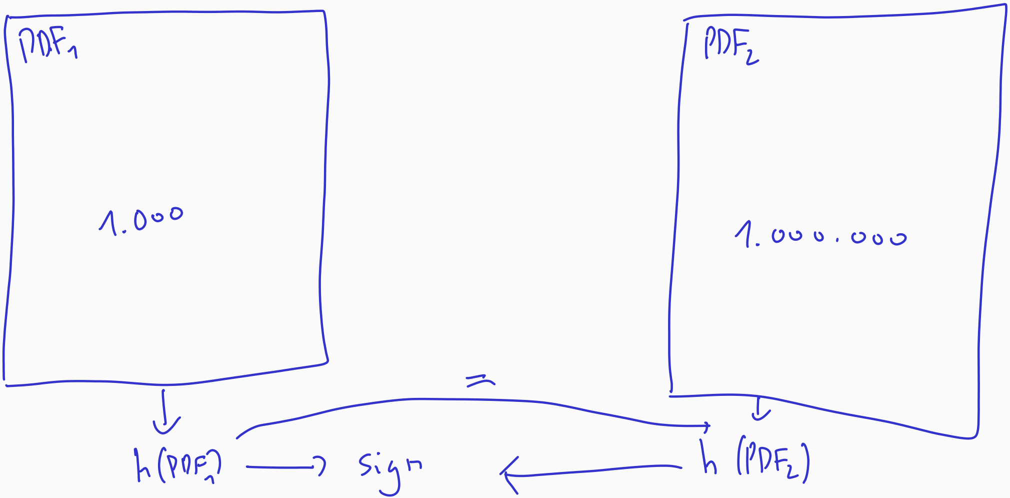
- h is one-way (or 1st preimage resistant) if, given $h(m)$, it is difficult to compute m
- h is 2nd preimage resistant if, given m and $h(m)$, it is difficult to compute $m' \neq m$ s.t. $h(m') = h(m)$
- h is collision resistant if it is difficult to compute m, m' s.t. $m' \neq m$ and $h(m') = h(m)$

(In all cases, we assume the attacker has full access to the details of h.)

Existence of one-way functions would imply $\mathrm{P} \neq \mathrm{NP}$. Nevertheless, none of the practically used hash functions was so far broken w.r.t. one-wayness.

However, many older hash functions (e.g. MD5, SHA-1) were broken w.r.t. collision resistance, and hence they are no longer deemed secure.

Digital signatures and collision resistance



Additional properties of hash functions

- Non-correlation: small change of m should elicit large and random-looking change of $h(m)$
- Local preimage resistance: given $h(m)$ it should be difficult to obtain e.g. short sub-strings of m

Secure hash functions

Each of the three main security properties of hash functions can be phrased in terms of guessing a suitable certificate (preimage/2nd preimage/collision). A hash function adversary is a probabilistic algorithm which tries to guess such a certificate.

Security of hash functions often evaluated w.r.t. performance of certain baseline adversaries: when guessing, how long do we need to guess (on average) to obtain the certificate?

In the following, ℓ is the hash length:

property	baseline
1st preimage resistance	2^ℓ
2nd preimage resistance	2^ℓ
collision resistance	

Secure hash functions

Each of the three main security properties of hash functions can be phrased in terms of guessing a suitable certificate (preimage/2nd preimage/collision). A hash function adversary is a probabilistic algorithm which tries to guess such a certificate.

Security of hash functions often evaluated w.r.t. performance of certain baseline adversaries: when guessing, how long do we need to guess (on average) to obtain the certificate?

In the following, ℓ is the hash length:

property	baseline
1st preimage resistance	2^{ℓ}
2nd preimage resistance	2^{ℓ}
collision resistance	$\sqrt{2^{\ell}}$ $= 2^{\frac{\ell}{2}}$

Birthday theorem

Birthday "paradox"

Given a group of 23 people. What is the probability that at least two of them were born on the same day of the year?

Birthday theorem

Birthday "paradox"

Given a group of 23 people. What is the probability that at least two of them were born on the same day of the year?

Answer: 50.7%

(For 60 people, the probability is $\geq 99\%$.)

Birthday theorem

Birthday "paradox"

Given a group of 23 people. What is the probability that at least two of them were born on the same day of the year?

Answer: 50.7%

(For 60 people, the probability is $\geq 99\%$.)

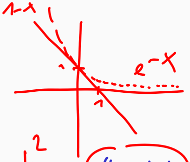
Theorem 1: Birthday theorem

Consider the experiment of randomly drawing (with replacement) items from a set of size N. Denote by $col(N)$ the minimum number of draws that need to be performed so that the probability of drawing some item twice exceeds $\frac{1}{2}$. Then $col(N) \in \mathcal{O}(\sqrt{N})$.

Let $c(k, N)$ be the probability of a drawing something twice in k draws from an N-element set.

$$c(k,N) = 1 - 1 \cdot \left(\frac{N-1}{N}\right) \cdot \left(\frac{N-2}{N}\right) \cdots \left(\frac{N-(k-1)}{N}\right) =$$

$$\forall x \in \mathbb{R} \quad 1 - x \le e^{-x}$$



$$= 1 - 1 \cdot \underbrace{\left(1 - \frac{1}{N}\right)}_{\le e^{-\frac{1}{N}}} \cdot \underbrace{\left(1 - \frac{2}{N}\right)}_{\le e^{-\frac{2}{N}}} \cdots \left(1 - \frac{k-1}{N}\right)$$

$$\ge 1 - e^{-\frac{1}{N}} \cdot e^{-\frac{2}{N}} \cdots e^{-\frac{k-1}{N}} = 1 - e^{-\frac{1+2+\dots(k-1)}{N}} = 1 - e^{-\frac{(k-1)\cdot k}{2N}}$$

$$k^2 \approx (k-1)\cdot k$$

$$\ge \frac{1}{2}$$

$$k \approx \sqrt{N}$$

$$k = 1.2\sqrt{N}$$

Generic birthday attack

Algorithm 1: Generic birthday attack

Input: $\mathcal{M} = \{0,1\}^{\leq L}, \mathcal{H} = \{0,1\}^{\ell}, h \colon \mathcal{M} \to \mathcal{H}$.

Output: Pair of messages $m, m' \in \mathcal{M}$ s.t. $h(m) = h(m')$.

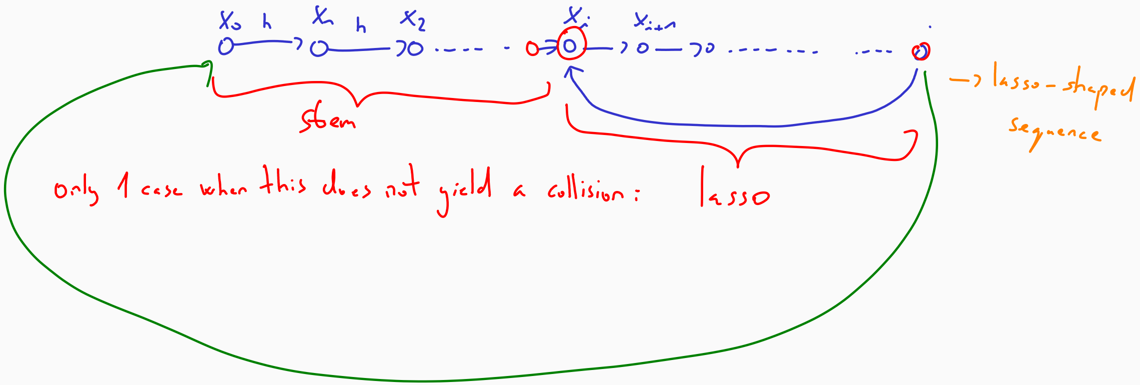
```
M ← ∅;
while true do
    m ← sample(𝓜);
    if ∃ (m', h(m')) ∈ M s.t. h(m') = h(m) ∧ m' ≠ m then
        return (m, m')
    else
        M ← M ∪ {(m, h(m))}
```

According to the birthday theorem, after $2^{\frac{\ell}{2}}$ iterations we get a constant probability p of finding a collision. Hence, the expected runtime is $\leq \frac{1}{p} \cdot 2^{\frac{\ell}{2}}$.

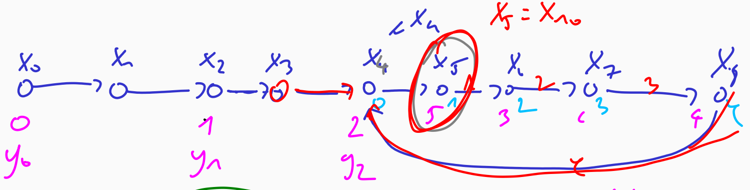
The attack speed depends solely on the hash output length!

Generic birthday attack made practical: Cycle detection

$x_0 = m \quad x_1 = h(m) \quad x_2 = h(x_1) \quad \ldots \quad x_n = h(x_{n-1}) \quad \ldots \qquad \mathcal{H} = \{0,1\}^\ell$



Generic birthday attack made practical: Cycle detection



Theorem 2

A sequence $x_1, x_2, x_3, \ldots$ is lasso-shaped (ultimately periodic) if and only if there exists $n \geq 1$ s.t. $x_n = x_{2n}$.

Practical birthday attack via "Floyd" cycle detection

Algorithm 2: Birthday attack via cycle detection

Input: $\mathcal{M} = \{0,1\}^{\leq L}, \mathcal{H} = \{0,1\}^{\ell}, h\colon \mathcal{M} \to \mathcal{H}$.

Output: Pair of messages $m, m' \in \mathcal{M}$ s.t. $h(m) = h(m')$.

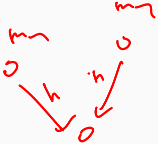
repeat
- $v_1 \leftarrow v_2 \leftarrow v \leftarrow \mathit{sample}(\mathcal{M})$

until $h(v_1) \neq v_1$;

repeat
- $v_1 \leftarrow h(v_1)$; $v_2 \leftarrow h(h(v_2))$

until $v_1 = v_2$;

if $v_1 = v$ **then**
- **return** *error*



repeat
- $m_1 \leftarrow v$; $m_2 \leftarrow v_1$
- $v \leftarrow h(v)$ $v_1 \leftarrow h(v_1)$;

until $v \neq v_1$;

return (m_1, m_2)

Designing collision resistant hash functions

Two major approaches:

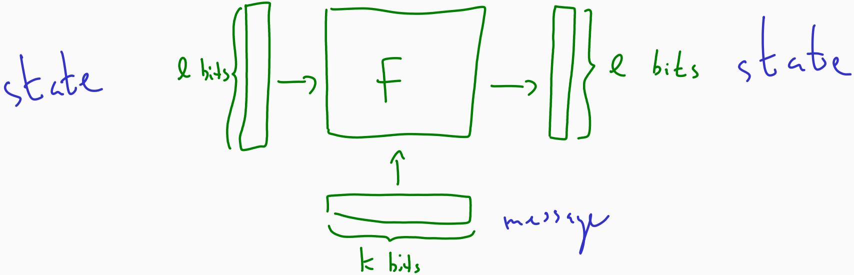
- Merkle-Damgård (MD) construction (e.g. SHA-2)
- sponge construction (e.g. SHA-3)

Merkle-Damgård construction: basics

Basic idea: extend functions that hash short messages into functions that hash messages of arbitrary length.

Compression function

A compression function for a hash space $\mathcal{H} = \{0,1\}^\ell$ is a function $f\colon \{0,1\}^\ell \times \{0,1\}^k \to \{0,1\}^\ell$ for $k \approx \ell$.



Merkle-Damgård construction: basics

Basic idea: extend functions that hash short messages into functions that hash messages of arbitrary length.

Compression function

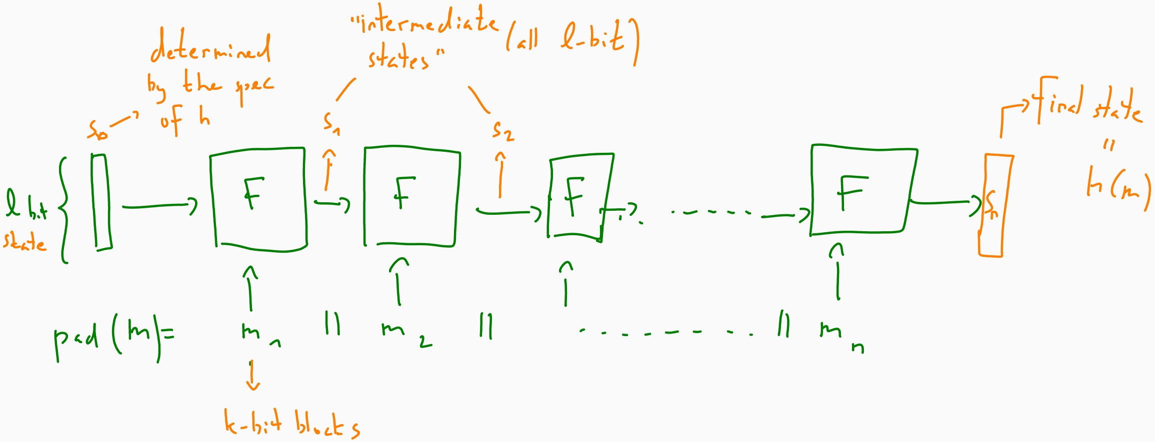
A compression function for a hash space $\mathcal{H} = \{0,1\}^\ell$ is a function $f\colon \{0,1\}^\ell \times \{0,1\}^k \to \{0,1\}^\ell$ for $k \approx \ell$.

An MD hash function $h\colon \{0,1\}^{\leq L} \to \{0,1\}^\ell$ is specified by the choice of the following parameters:

- A compression function $f\colon \{0,1\}^\ell \times \{0,1\}^k \to \{0,1\}^\ell$ it uses; and
- an MD-compliant padding scheme $pad\colon \{0,1\}^{\leq L} \to \{0,1\}^{\times k}$, where $\{0,1\}^{\times k}$ is the set of all bit strings whose length is a multiple of k.

Computing h(m):

First, use *pad* to pad message m so that its length is a multiple of k. Then:



Merkle-Damgård construction: pseudocode

Algorithm 3: Hashing via an MD hash function h constructed from compression function $f : \{0,1\}^\ell \times \{0,1\}^k \to \{0,1\}^\ell$ and padding scheme *pad*.

Input: $m \in \mathcal{M}$

Output: $h(m) \in \mathcal{H} = \{0,1\}^\ell$

```
m ← pad(m);
s ← a constant specified in the definition of h;
repeat
    b ← the first k-bit block of m;
    s ← f(s, b);
    m ← m with the first k bits removed
until m is an empty string;
return s
```

MD-compliant padding

Definition 1: MD-compliant padding

A padding scheme $pad\colon \{0,1\}^{\leq L} \to \{0,1\}^{\times k}$ is MD-compliant if it satisfies the following three properties for all $m, m' \in \mathcal{M}$:

- $m \neq m' \quad \Rightarrow \quad pad(m) \neq pad(m')$
- $len(m) = len(m') \quad \Rightarrow \quad len(pad(m)) = len(pad(m'))$
- $len(m) \neq len(m') \quad \Rightarrow$ the last k-bit block of $pad(m)$ differs from the last k-bit block of $pad(m')$

Merkle padding: $pad(m) = m \,\|\, 1 \,\|\, 0^d \,\|\, len(M)$

d: smallest positive integer that makes the result's length a multiple of k

$len(M)$: represented as a bitvector of fixed size (typically 64 or 128 bits)

MD construction: security proof

Theorem 3

Let h be a hash function built from a compression function f via the Merkle-Damgård construction. Then, given a collision m, m' for h one can efficiently (i.e. in time proportional to a constant number of evaluations of h) compute a collision for f.

Theorem 3

Let h be a hash function built from a compression function f via the Merkle-Damgård construction. Then, given a collision m, m' for h one can efficiently (i.e. in time proportional to a constant number of evaluations of h) compute a collision for f.

Case 1: $len(m) \neq len(m')$

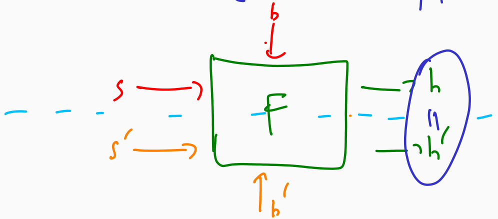
pad(m) ≠ pad(m'), in particular, the last two blocks b, b' of these do differ

$f(s,b) = f(s',b')$

$b \neq b'$

m

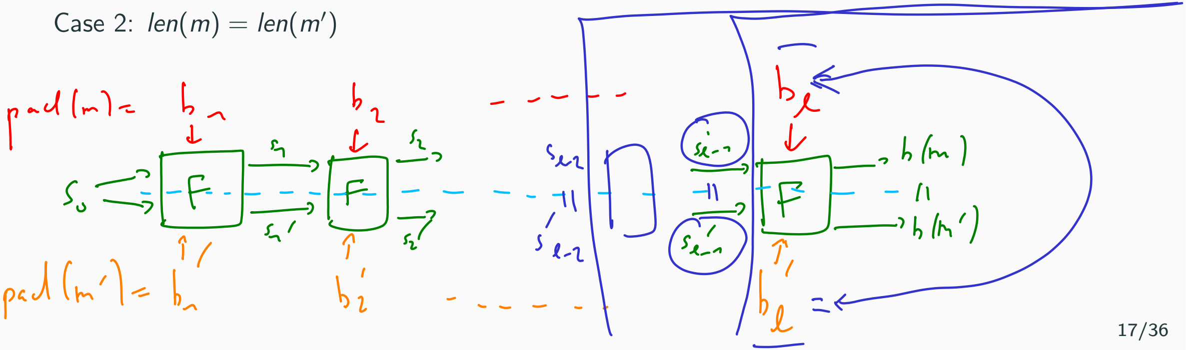
m'



Theorem 3

Let h be a hash function built from a compression function f via the Merkle-Damgård construction. Then, given a collision m, m' for h one can efficiently (i.e. in time proportional to a constant number of evaluations of h) compute a collision for f.

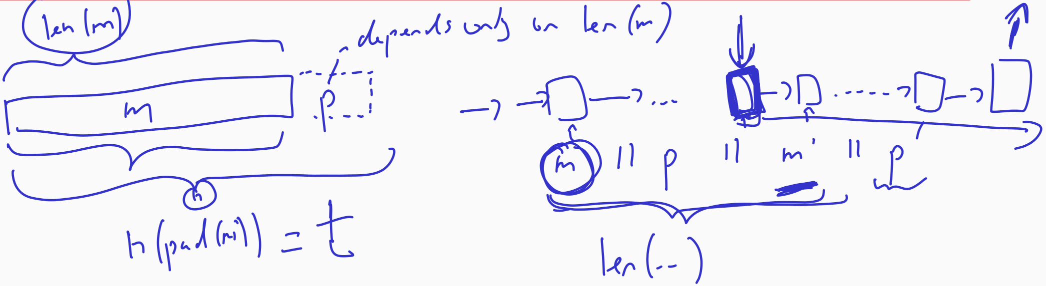
Case 2: $len(m) = len(m')$



MD construction: notable properties

Theorem 4

Let h be a hash function built via the Merkle-Damgård construction, using an MD-compliant padding scheme *pad* s.t. m is always a prefix of $pad(m)$ and the suffix added by *pad* only depends on $len(m)$. Then, given $h(m)$ and $len(m)$, one can compute, for any string m', the hash $h(pad(m) \,||\, m')$. (Even without the knowledge of m!)



Construction of compression functions

A compression function combines two short bitvectors (state and message block) into a single bitvector (next state.)

Construction of compression functions

A compression function combines two short bitvectors (state and message block) into a single bitvector (next state.)

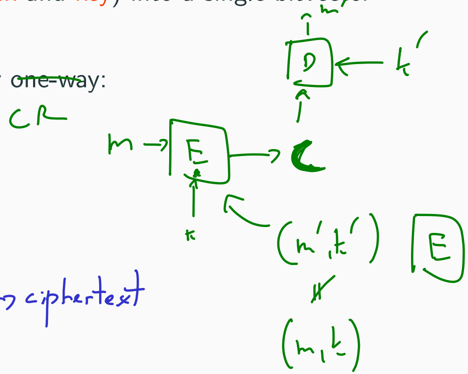
A block cipher combines two short bitvectors (input block and key) into a single bitvector (output block.)

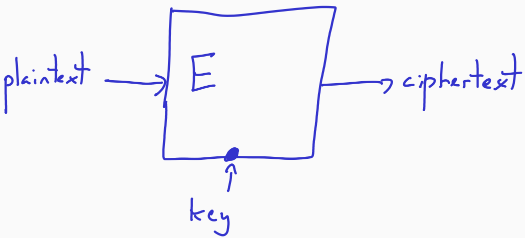
Construction of compression functions

A compression function combines two short bitvectors (state and message block) into a single bitvector (next state.)

A block cipher combines two short bitvectors (input block and key) into a single bitvector (output block.)

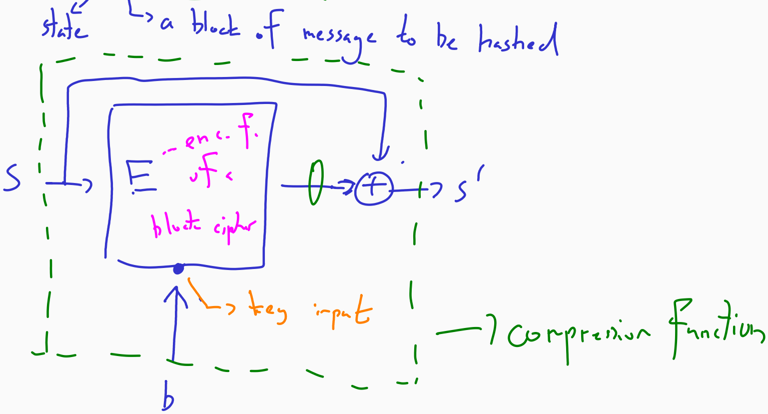
However, block ciphers are not designed to be inherently one-way:





Davies-Meyer

$$f(s, b) = E(b, s) \oplus s$$

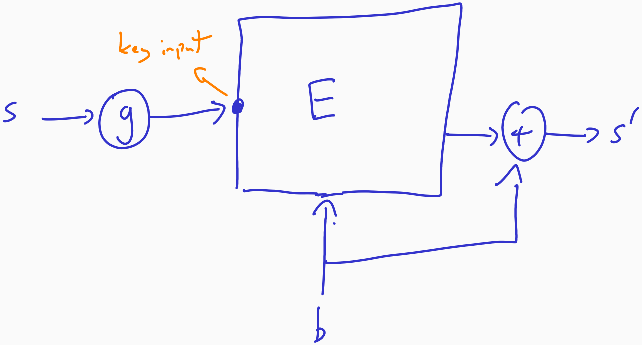


Secure compression functions from block ciphers

Matyas-Meyer-Oseas

$$f(s, b) = E(g(s), b) \oplus b$$

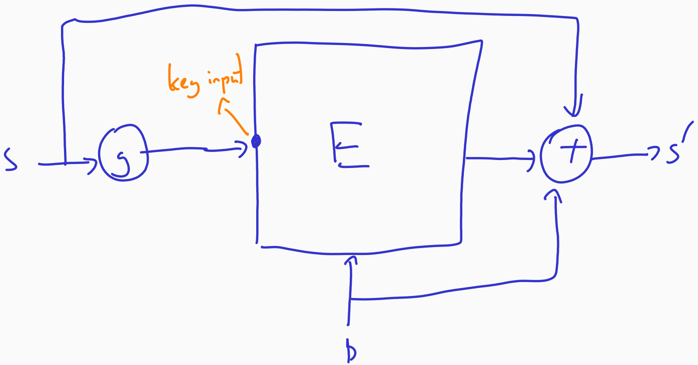
(Where g is a function mapping state bitvectors to keys for E).



Miyaguchi-Preneel

$$f(s, b) = E(g(s), b) \oplus b \oplus s$$

(Where g is a function mapping state bitvectors to keys for E).



Reasoning about the security of Davies-Meyer et al.

Theorem of the type "If E is the encryption function of a secure block cipher, then Davies-Meyer compression function constructed from E is collision resistant" are unlikely to hold, since block cipher security assumes the key is out of adversary's control: this does not hold in the hash function scenario.

There are different, stronger and less realistic notions of block cipher security under which theorems of the above form can be proved. Not all reasonable block ciphers satisfy these requirements, so such proofs are of limited practical consequence (and hence we omit them). This is however one of the reasons why custom-made block ciphers are used for hashing (instead of block ciphers used for encryption, such as AES and friends).

Block ciphers used for compression in practice

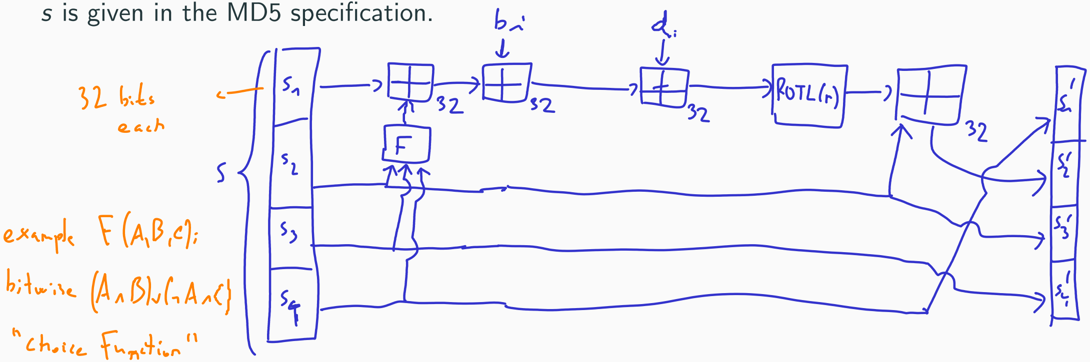
Another reason for custom-made block ciphers for hashing: Block ciphers primarily aimed at encryption (AES & friends) are optimized for the scenario when the same key is used to encode many blocks in a row. Hence, they do not perform well when keys change rapidly, as in the MD construction.

Hence: Practical hash functions use tailor-made "block ciphers" as compression functions.

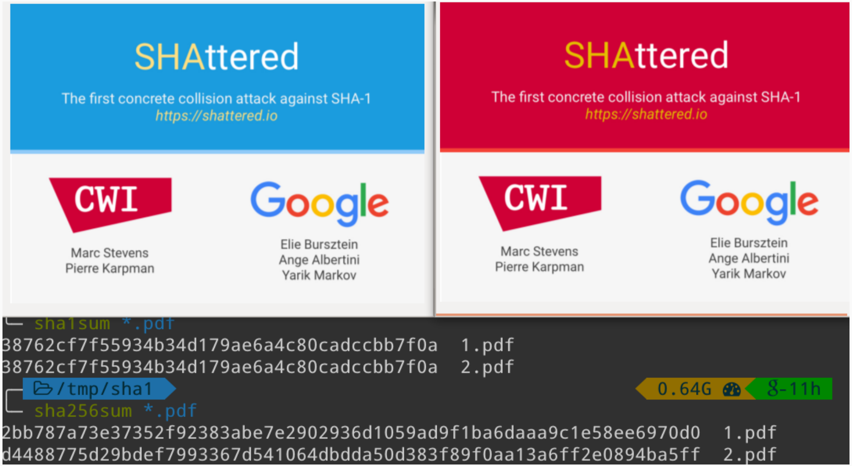
- major obsolete MD-based hash functions: MD5, SHA-1
- state-of-the-art MD-based hash function: SHA-2

MD5 block cipher

128-bit state, 512-bit message block Davies-Meyer compression function. Computing $E(b, s)$ consists of 4 rounds: for each round, we break b into 32-bit blocks $b_1, \ldots, b_{16}$ and pass s through a series of 16 of the following operations (d_i are nothing-up-my-sleeve constants differing for each of the 16 rounds, F is a non-linear function differing for each round, the order in which the blocks b_i are fed to the operations varies between the rounds). The initial value of s is given in the MD5 specification.



Similar in spirit to MD5, but uses 160-bit hash length. Collisions were found, so not considered secure (source: Stevens, Karpman, Bursztein, Albertini, Markov: `https://shattered.io`).

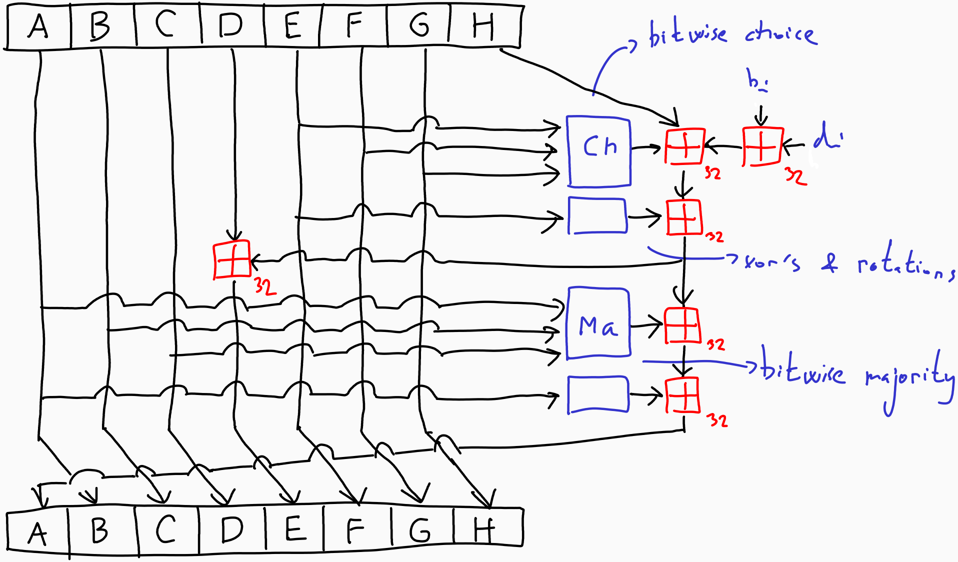


SHA-2

A family of hash functions with different hash lengths:

- most prominent are SHA-256 and SHA-512
- use modified Davies-Meyer, where the $\oplus$ of the output and the previous state is replaced by wordwise addition modulo 2^{32} (see also ChaCha) $\boxplus_{32}$
- SHA-256: uses 256-bit states and 512-bit message blocks
- the block cipher SHACAL-2 of SHA-2 consists of 64 rounds of the operation specified on the next slide (80 rounds for SHA-512)
- considered secure as of today
- used in various blockchain protocols (incl. Bitcoin)

SHA-2 main operation



Known attacks: MD5 and SHA-2 comparison

Hash function	Hash size	Property	Baseline	Best known attack
MD5	128	collision resistance	2^{64}	2^{18}
MD5	128	1st preimage resistance	2^{128}	$2^{123.4}$
SHA-2	256	collision resistance	2^{128}	31 out of 64 rounds in $2^{49.8}$ 28 out of 64 rounds in practice
SHA-2	256	1st preimage resistance	2^{256}	43 out of 64 rounds in $2^{255.5}$
SHA-2	512	collision resistance	2^{256}	31 out of 80 rounds in $2^{115.6}$ 27 out of 80 rounds in practice
SHA-2	512	1st preimage resistance	2^{512}	50 out of 80 rounds in $2^{511.5}$

Sponge construction

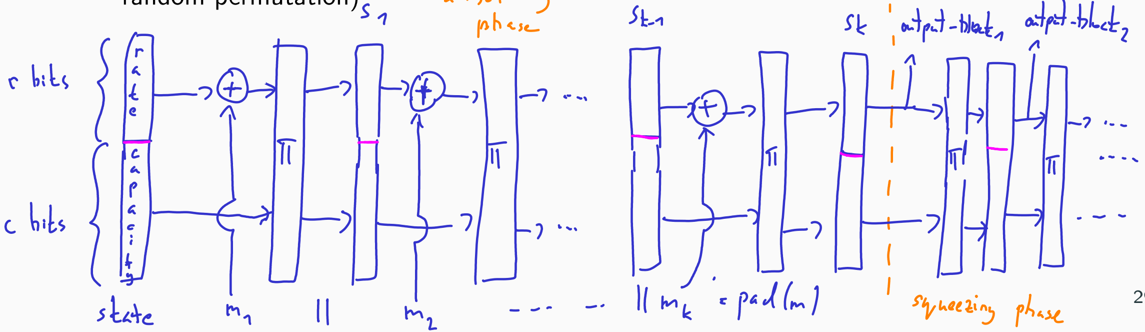
While SHA-2 is considered secure, there were concerns about all the standardized hashing functions being based on the Merkle-Damgård paradigm. In late 2000's, NIST announced a competition for hash functions based on alternative paradigms.

Sponge construction: A generic construction mapping arbitrary-length input bitstreams to arbitrary length output bitstreams: variability of use (hash functions, PRGs...)

Sponge construction II

A sponge construction determined by:

- choice of state bitlength n, capacity c and rate r; we have $n = c + r$
- choice of padding function *pad* that pads to a length that is multiple of r (any injective padding suffices)
- choice of round permutation $\pi\colon \{0,1\}^n \to \{0,1\}^n$ (should be indistinguishable from a random permutation)



Sponge construction: pseudocode

Algorithm 4: Hashing via sponge based function h with rate r, capacity c, permutation $\pi\colon \{0,1\}^{r+c} \to \{0,1\}^{r+c}$ and padding scheme *pad*.

Input: $m \in \mathcal{M}$, k - number of required output blocks

Output: $h(m) \in \mathcal{H} = \{0,1\}^{r \cdot k}$

$m \leftarrow pad(m)$; $cap \leftarrow 0^c$; $rate \leftarrow 0^r$;

repeat

- $rate \leftarrow rate \oplus$ the first r-bit block of m;
- $x \leftarrow \pi(rate \,||\, cap)$;
- $rate \leftarrow$ first r bits of x; $cap \leftarrow$ last c bits of x;
- $m \leftarrow m$ with the first r bits removed

until *m is an empty string*;

absorbing phase

for $i \in \{1, \ldots, k\}$ **do**

- **print** *rate*;
- $x \leftarrow \pi(rate \,||\, cap)$;
- $rate \leftarrow$ first r bits of x; $cap \leftarrow$ last c bits of x;

squeezing phase

Keccak (aka SHA-3)

- Bertoni, Daemen, Peeters, Van Assche (around 2009)
- based on an earlier RadioGatún hash function
- padding simply appends $1 \,||\, 0^* \,||\, 1$
- uses rather complex permutation π consisting of variable number of rounds (24 in the original submission) of simpler operations (including rotations, xor's, sub-word permutations, non-linear operation including bitwise $\wedge$, and round constants derived from a fixed linear feedback shift register sequence): see `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf`
- uses $r + c = 1600$ with variable capacities; see next slide (higher capacity = higher security, lower performance)

SHA-3 variants

Let $kec[c, \ell]$ be a sponge-based hash function operating as follows:

- the input message m is padded with $1 \,||\, 0^* \,||\, 1$ so that its length is a multiple of $r = 1600 - c$
- a sponge derived from the Keccak permutation is used to produce an r-bit output block
- the first ℓ bits of the output block are returned as the hash of m

Then the SHA-3 standard defines the following variants:

SHAKE

$$
\begin{aligned}
\textit{SHA-3-224}(m) &= kec[448, 224](m \,||\, 01) \\
\textit{SHA-3-256}(m) &= kec[512, 256](m \,||\, 01) \\
\textit{SHA-3-384}(m) &= kec[768, 384](m \,||\, 01) \\
\textit{SHA-3-512}(m) &= kec[1024, 512](m \,||\, 01)
\end{aligned}
$$

Merkle trees

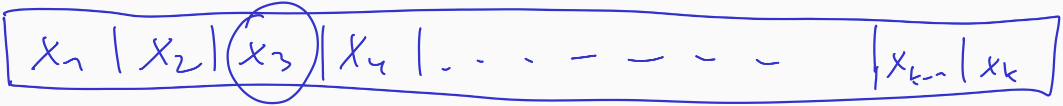
Consider the following scenario: We are presented with a list $List = (x_1, x_2, \ldots, x_k)$, where k is a power of 2. We want to come up with a hash-based scheme with these features:

- checking the integrity of *List*

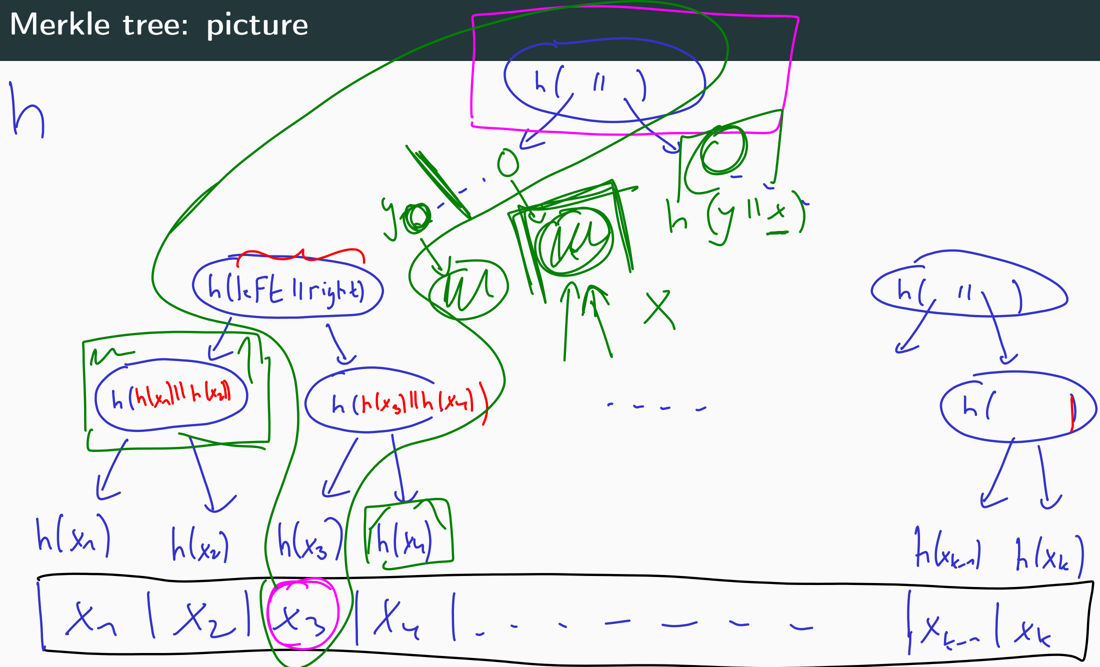
Merkle trees

Consider the following scenario: We are presented with a list $List = (x_1, x_2, \ldots, x_k)$, where k is a power of 2. We want to come up with a hash-based scheme with these features:

- checking the integrity of *List*
- given an index $1 \leq i \leq k$ and an item x, proving that the i-th item of *List* is x (i.e. that $x_i = x$) without revealing the contents of the remainder of the list



Merkle tree: picture



Notes on Merkle trees (I)

- leaves = items of *List* (i-th leaf from the left contains x_i)
- next-to-last level: hashes of *List*'s items (i-th node from the left $= h(x_i)$)
- all other internal nodes contain h(left_child,right_child)

Notes on Merkle trees (II)

- For the scheme to work, the Merkle root must be public and integrity preserved (e.g. in Bitcoin: each block header contains a Merkle root of its list of transactions, as well as a hash of the previous block's header).
- To demonstrate that $x_i = x$, the prover presents a proof consisting of hashes stored in siblings of all internal nodes on the path from the i-th leaf to the root.
- Given such a proof (and x), the verifier can compute all hashes on the path from the i-th leaf to the root, which he does and checks that the computed Merkle root matches the publicly known root.