Crypto Engineering 2024
Hash functions

Léo COLISSON PALAIS

leo.colisson-palais@univ-grenoble-alpes.fr

https://leo.colisson.me/teaching.html

Hash functions

What is a hash function?

Hash function

A hash function is a function $h : \{0, 1\}^* \to \{0, 1\}^n$.

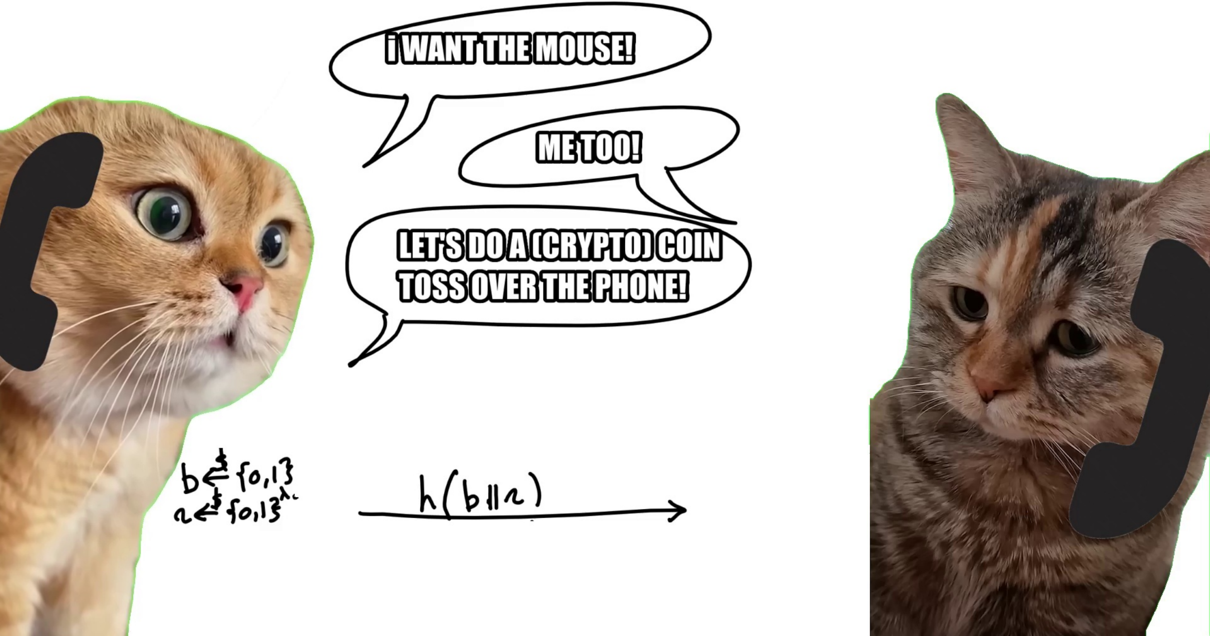
As it not really helpful, but based on applications, hash functions may have **multiple properties**. Informally:

- **Collision-resistance**: hard to find a collision, i.e. $x \neq x'$ such that $h(x) = h(x')$
- **First-preimage resistance** (one-way): given y, hard to find x such that $h(x) = y$
- **Second-preimage resistance**: given a random x, hard to find $x' \neq x$ such that $h(x) = h(x')$
- **Hiding**: given $h(r \| x)$ for a long enough random r, hard to find x
- **Universality**: weaker assumption about the distribution of the output

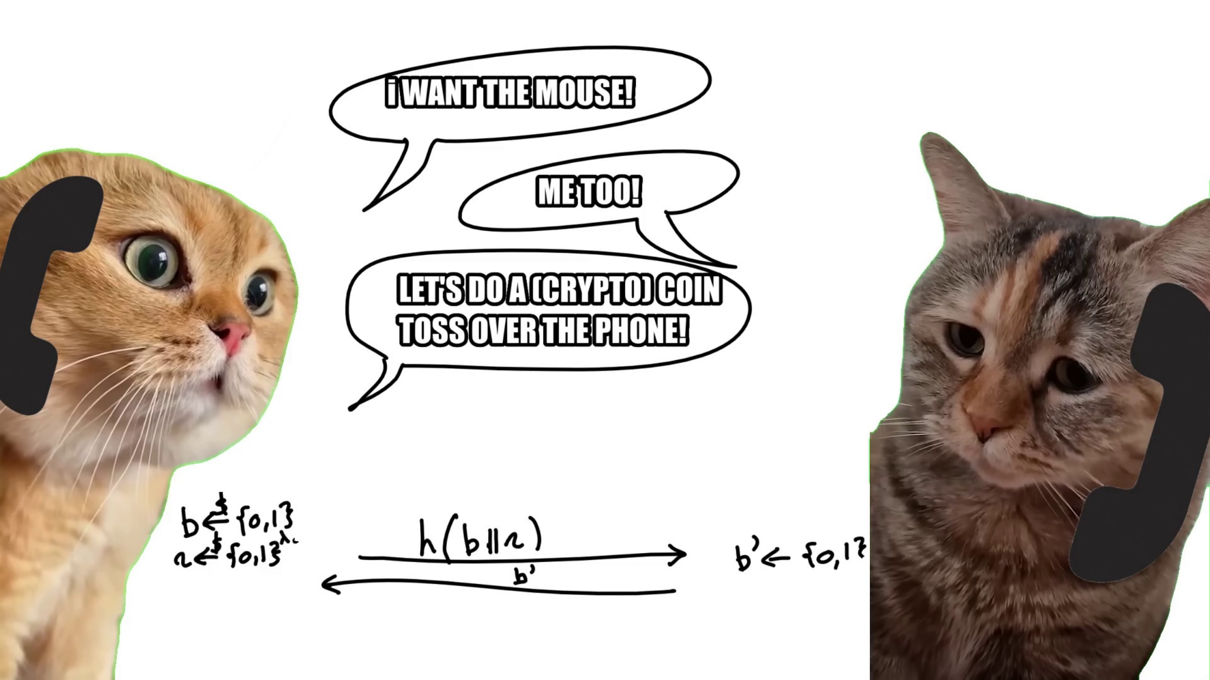

¡WANT THE MOUSE!

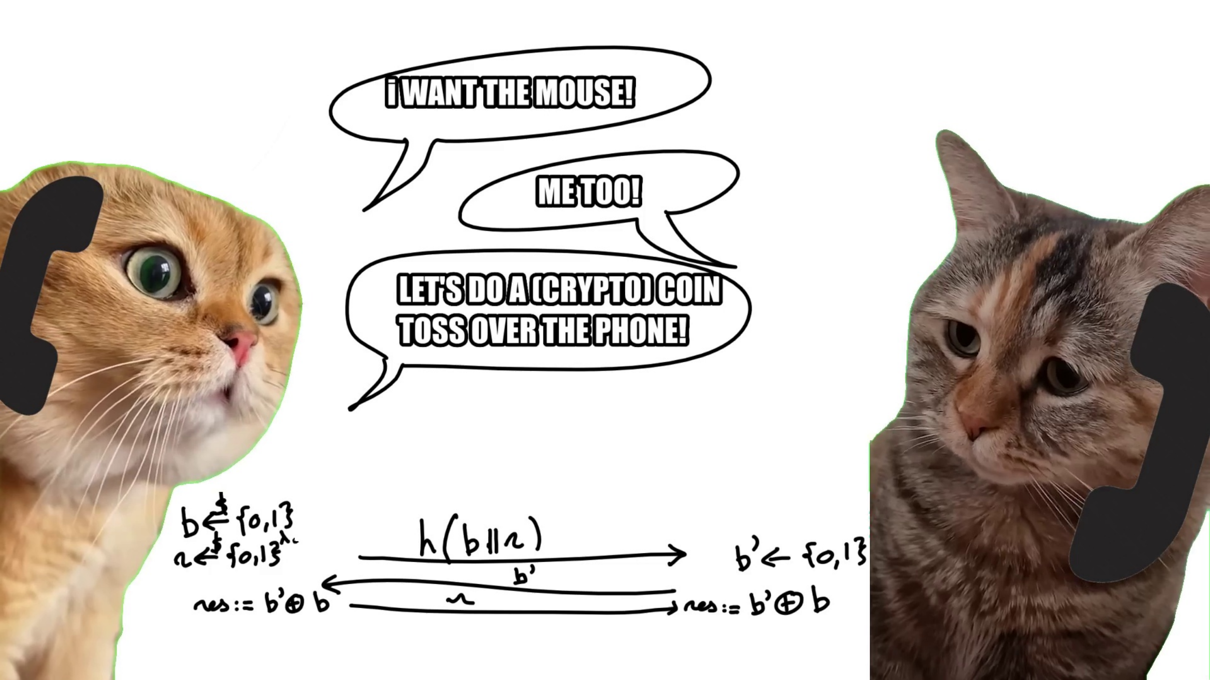

i WANT THE MOUSE!
ME TOO!

i WANT THE MOUSE!
ME TOO!
LET'S DO A (CRYPTO) COIN TOSS OVER THE PHONE!



¡WANT THE MOUSE!
ME TOO!
LET'S DO A (CRYPTO) COIN TOSS OVER THE PHONE!
$b \xleftarrow{\$} \{0,1\}$
$r \xleftarrow{\$} \{0,1\}^{\lambda}$
$h(b \| r)$


¡WANT THE MOUSE!
ME TOO!
LET'S DO A (CRYPTO) COIN TOSS OVER THE PHONE!
$b \xleftarrow{\$} \{0,1\}$
$r \xleftarrow{\$} \{0,1\}^{\lambda}$
$h(b \| r)$
b'
$b' \leftarrow \{0,1\}$



¡WANT THE MOUSE!
ME TOO!
LET'S DO A (CRYPTO) COIN TOSS OVER THE PHONE!
$b \xleftarrow{\$} \{0,1\}$
$r \xleftarrow{\$} \{0,1\}^\lambda$
$h(b \| r)$
$b' \leftarrow \{0,1\}$
b'
r
$res := b' \oplus b$
$res := b' \oplus b$

What security property is needed?

?

Do we need collision resistance here?

- (A) No
- (B) Yes, to protect against malicious left cat
- (C) Yes, to protect against malicious right cat

What security property is needed?

?

Do we need collision resistance here?

Ⓐ No ✗

Ⓑ Yes, to protect against malicious left cat ✓ If left cat knowns r, r' s.t. $h(0\|r) = h(1\|r')$ and wants outcome 0: reveal r if $b' = 0$, else r' (res $= b' \oplus b = 0$)

Ⓒ Yes, to protect against malicious right cat ✗

What security property is needed?

?

Do we need hiding here?

- (A) No
- (B) Yes, to protect against malicious left cat
- (C) Yes, to protect against malicious right cat

What security property is needed?

?

Do we need hiding here?

- (A) No ✗
- (B) Yes, to protect against malicious left cat ✗
- (C) Yes, to protect against malicious right cat ✓ If right cat can recover b from $h(b\|r)$, and wants outcome 0, just send $b' := b$ (res $= b' \oplus b = 0$).

Applications hash functions

Hash function = many **applications**:

- Efficiently check integrity of file (fingerprint)
- Authentication (HMAC, NMAC, Envelope MAC. . .)
- **IND-CCA** constructions
- Secure password storage
- Organize, retrieve and/or cache data efficiently and/or securely (git, nix, . . .)
- Blockchain (proof of work)
- Commitments
- Coin tossing
- Zero-knowledge proofs
- Multi-party computing
- . . .

Formal definition

How to formally define collision-resistance?

First attempt: what about h is collision-resistant iff:

$\mathcal{L}_0$
TEST(x, x'):
if $x \neq x'$ and $h(x) = h(x')$: return `true`
else: return `false`

$\approx$

$\mathcal{L}_1$
TEST(x, x'):
return `false`

?

Formal definition

?

Is this a reasonable definition?

1. Yes
2. No, because basically all functions would be collision-resistant
3. No, because basically no function would be collision-resistant

Formal definition

?

Is this a reasonable definition?

1. Yes
2. No, because basically all functions would be collision-resistant
3. No, because basically no function would be collision-resistant

✓ Very **subtle** issue in order of quantifiers. This definition says: h collision-resistant iff $\forall \mathcal{A}$, $|\Pr[\mathcal{A} \diamond \mathcal{L}_0 = 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_1 = 1]| \leq$ negl(λ). Since $\mathcal{A}$ appears **after** h, $\mathcal{A}$ can depend arbitrarily on h. So $\mathcal{A}$ could just happen to **hardcode** a collision (x, x'), like:

$\mathcal{A}$
return TEST(x, x')

It is really hard to find the code of $\mathcal{A}$, but $\mathcal{A}$ still runs in polynomial time!
$\Rightarrow$ We would like to fix h **after** $\mathcal{A}$: public **salt**

Formal definition

Salt = random publicly known value sampled to "customize" the function h.

Collision-resistance (flavor 1)

A hash function $h\colon \{\texttt{0},\texttt{1}\}^* \times \{\texttt{0},\texttt{1}\}^* \to \{\texttt{0},\texttt{1}\}^*$ is collision resistant if:

$\mathcal{L}^h_{\text{cr-real}}$
$s \xleftarrow{\$} \{0,1\}^\lambda$
<u>GETSALT$()$:</u>
return s
<u>TEST(x,x'):</u>
if $x \neq x'$ and $h(s,x) = h(s,x')$: return `true`
else: return `false`

$\approx$

$\mathcal{L}^h_{\text{cr-fake}}$
$s \xleftarrow{\$} \{0,1\}^\lambda$
<u>GETSALT$()$:</u>
return s
<u>TEST(x,x'):</u>
Return `false`

Often: $h(s,x) := h(s\|x)$.

Formal definition

Problem: this definition is **rarely useful as it** since we never explicitly check if there is a collision: we just assume there is none.

Rather **used in reductions**: if $\mathcal{A}$ can distinguish $\mathcal{L}_0$ from $\mathcal{L}_1$, then we can build $\mathcal{A}'$ (calling $\mathcal{A}$ internally) that finds a collision against h (then, trivial to distinguish $\mathcal{L}^h_{\text{cr-real}}$ from $\mathcal{L}^h_{\text{cr-fake}}$). Hence this equivalent definition might be easier to use:

Collision-resistance (flavor 2)

A hash function h is collision resistant if for any polynomially bounded $\mathcal{A}$:

$$\Pr_{\substack{s \leftarrow \{0,1\}^\lambda \\ (x,x') \leftarrow \mathcal{A}(s)}} \left[h(s,x) = h(s,x') \right] \leq \mathsf{negl}(\lambda)$$

Specificity of password hashing

Hashing password

?

Alice is creating a website, and, to provide extra-security, she decides to store the user's passwords by encrypting them with AES in CTR mode. Is this a good idea, why?

- (A) Yes
- (B) No

Hashing password

Alice is creating a website, and, to provide extra-security, she decides to store the user's passwords by encrypting them with AES in CTR mode. Is this a good idea, why?

Ⓐ Yes ✗

Ⓑ No ✓ To check the passwords, she needs the decryption key, and this key will stay on the server. If the server is corrupted (database stolen. . .) the key will also likely be stolen, revealing the passwords.

You should always hash the passwords you store in a database!

Hashing passwords

If we can't "decrypt" the password $(s_{\text{Alice}}, h_{\text{Alice}})$ (hash function), how can we check if the password p is correct?

Hashing passwords

If we can't "decrypt" the password $(s_{\text{Alice}}, h_{\text{Alice}})$ (hash function), how can we check if the password p is correct?

$\Rightarrow$ Check if $h(s_{\text{Alice}}, p) = h_{\text{Alice}}$!

Salt

Salt: useful in theory… But **salt also useful in practice**! (change hash for every password) Otherwise

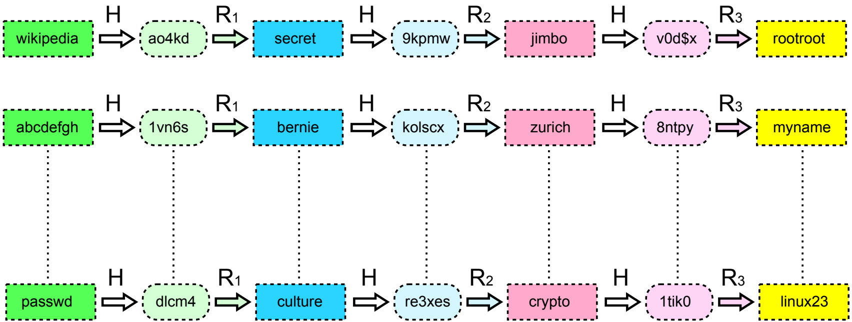
- Easy to see if two users have identical passwords
- Limit **pre-computation** attacks

Rainbow tables

How to (try to) recover a hashed password with no salt?

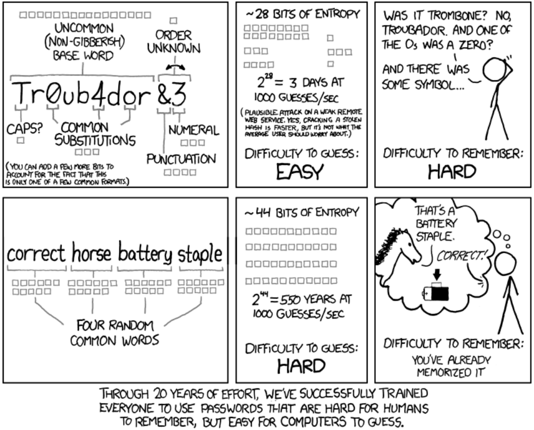
- **Method 1**: brute force, restart from scratch for any new password
  $\Rightarrow$ inefficient in time $O(\#_{\text{passwords}})$, efficient in space $O(1)$
- **Method 2**: brute force & store for re-use next time
  $\Rightarrow$ efficient in time once the table is generated $O(\log \#_{\text{passwords}})$, but needs HUGE storage $O(\#_{\text{passwords}})$
- **Method 3**: **rainbow tables** = time/space tradeoff
  $\Rightarrow$ e.g. moderate time $O(\sqrt{\#_{\text{passwords}}})$, moderate storage $O(\sqrt{\#_{\text{passwords}}})$

Rainbow tables



Different reduction function for each column = avoid long chains of collision

Salting is necessary (cost attack n passwords = $n\times$ cost of attacking 1 password), but not enough: low password entropy = few used passwordsu t:



Password hashing: recommendations

Mitigation: limit bruteforce attack with **slow hash functions**
(ideally on **any** hardware = memory-hard functions good candidate: assume memory is equaly costly/fast everywhere)

OWASP recommends

- Argon2 (ideally Argon2id)
- Scrypt (if Argon2 not available)
- Bcrypt (legacy systems)
- PBKDF2 if FIPS-140 compliance required

Don't use!

- NTLM (Windows: too quick)
- MD5 (broken)
- SHA1 (broken)
- SHA256 (too quick)

+ good to add **pepper** (HMAC of hash, with a key stored outside the database in case of SQL injection/backup access)

Password hashing: recommendations

Mitigation: limit bruteforce attack with **slow hash functions**
(ideally on **any** hardware = memory-hard functions good candidate: assume memory is equaly costly/fast everywhere)

Many great guides: more details in Password Storage Cheat Sheet, testing guide...

OWASP recommends

Argon2 (ideally Argon2id)

Scrypt (if Argon2 not available)

Bcrypt (legacy systems)

PBKDF2 if FIPS-140 compliance required

Don't use!

NTLM (Windows: too quick)

MD5 (broken)

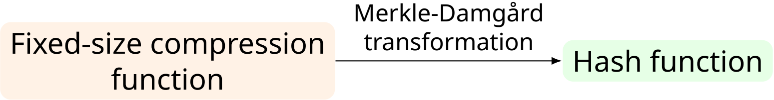
SHA1 (broken)

SHA256 (too quick)

+ good to add **pepper** (HMAC of hash, with a key stored outside the database in case of SQL injection/backup access)

Building hash functions

Merkle-Damgård



Merkle-Damgård used in:

- MD5 (broken)
- SHA-1 (broken)
- SHA-2 (still safe)

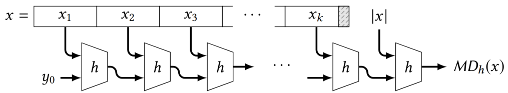
Merkle-Damgård

Merkle-Damgård construction

Let $h\colon \{0,1\}^{n+t} \to \{0,1\}^n$ be a compression function. Then the Merkle-Damgård transformation of h is $\mathrm{MD}_h\colon \{0,1\}^* \to \{0,1\}^n$ where:

$\underline{\textsc{mdpad}_t(x)}$
- $\ell := |x|$, as length-t binary number
- while $|x|$ not a multiple of t:
  - $x := x \| 0$
- return $x \| \ell$

$\underline{\mathrm{MD}_h(x)}$:
- $x_1 \| \cdots \| x_{k+1} := \textsc{mdpad}_t(x)$
- // *each x_i is t bits*
- $y_0 := 0^n$
- for $i = 1$ to $k+1$:
  - $y_i := h(y_{i-1} \| x_i)$
- output y_{k+1}



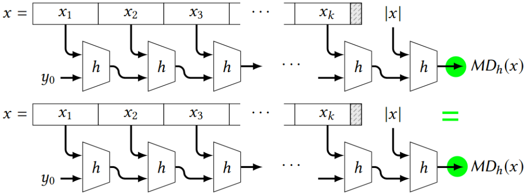
(actually, $h(x)$ is defined only if $x < 2^t$ here, but we can improve the padding part)

Merkle-Damgård

Theorem (Merkle-Damgård is collision resistant)

Let $h\colon \{0,1\}^{n+t} \to \{0,1\}^n$ be a collision-resistant compression function. Then the Merkle-Damgård transformation MD_h is collision resistant.

Proof. By contradiction, assume we found a collision $x \neq x'$ against MD_h, we want to find a collision against h:

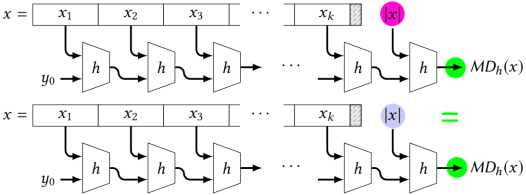


Merkle-Damgård

Theorem (Merkle-Damgård is collision resistant)

Let $h\colon \{0,1\}^{n+t} \to \{0,1\}^n$ be a collision-resistant compression function. Then the Merkle-Damgård transformation MD_h is collision resistant.

Proof. By contradiction, assume we found a collision $x \neq x'$ against MD_h, we want to find a collision against h:

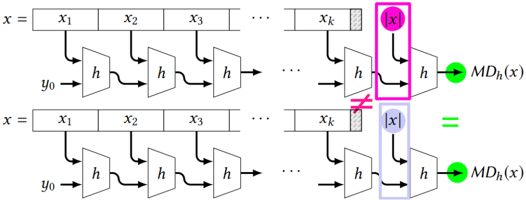


Merkle-Damgård

Theorem (Merkle-Damgård is collision resistant)

Let $h\colon \{\mathtt{0},\mathtt{1}\}^{n+t} \to \{\mathtt{0},\mathtt{1}\}^n$ be a collision-resistant compression function. Then the Merkle-Damgård transformation MD_h is collision resistant.

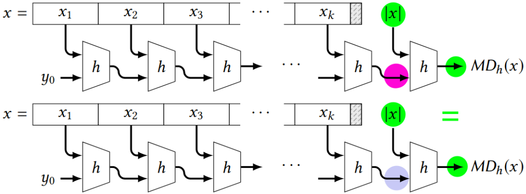
Proof. By contradiction, assume we found a collision $x \neq x'$ against MD_h, we want to find a collision against h:



Theorem (Merkle-Damgård is collision resistant)

Let $h\colon \{0,1\}^{n+t} \to \{0,1\}^n$ be a collision-resistant compression function. Then the Merkle-Damgård transformation MD_h is collision resistant.

Proof. By contradiction, assume we found a collision $x \neq x'$ against MD_h, we want to find a collision against h:

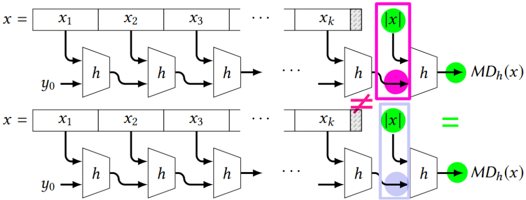


Merkle-Damgård

Theorem (Merkle-Damgård is collision resistant)

Let $h\colon \{0,1\}^{n+t} \to \{0,1\}^n$ be a collision-resistant compression function. Then the Merkle-Damgård transformation MD_h is collision resistant.

Proof. By contradiction, assume we found a collision $x \neq x'$ against MD_h, we want to find a collision against h:

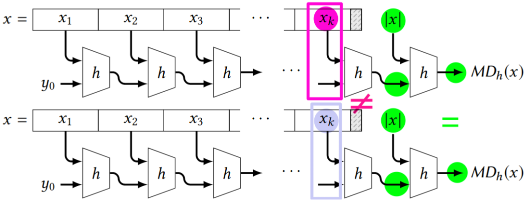


Merkle-Damgård

Theorem (Merkle-Damgård is collision resistant)

Let $h\colon \{0,1\}^{n+t} \to \{0,1\}^n$ be a collision-resistant compression function. Then the Merkle-Damgård transformation MD_h is collision resistant.

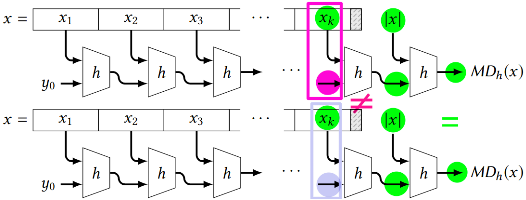
Proof. By contradiction, assume we found a collision $x \neq x'$ against MD_h, we want to find a collision against h:



Theorem (Merkle-Damgård is collision resistant)

Let $h\colon \{0,1\}^{n+t} \to \{0,1\}^n$ be a collision-resistant compression function. Then the Merkle-Damgård transformation MD_h is collision resistant.

Proof. By contradiction, assume we found a collision $x \neq x'$ against MD_h, we want to find a collision against h:

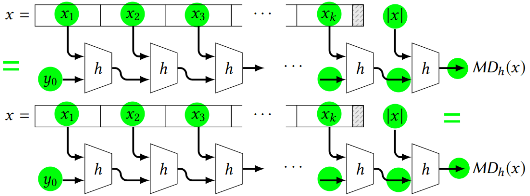


Merkle-Damgård

Theorem (Merkle-Damgård is collision resistant)

Let $h\colon \{0,1\}^{n+t} \to \{0,1\}^n$ be a collision-resistant compression function. Then the Merkle-Damgård transformation MD_h is collision resistant.

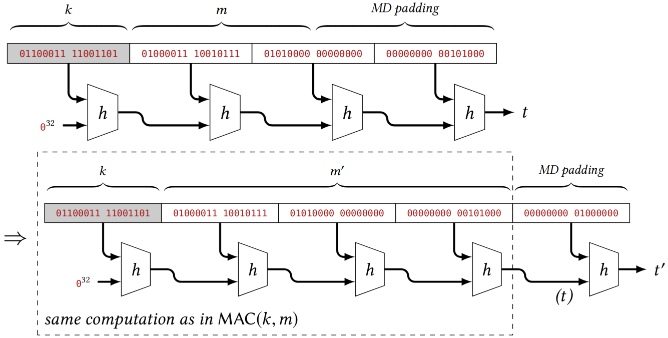
Proof. By contradiction, assume we found a collision $x \neq x'$ against MD_h, we want to find a collision against h:



$\square$

Length-extension attack

Goal: Obtain Message Authentication Code (MAC, ="signature", see next course) from hash functions via $h(k\|m)$.
Issue: **Length-extension attack**: With the MD construction, possible to get $h(k\|m')$ from $h(k\|m)$ (= sign a different message without knowing k).
How? Observation: "Knowing $H(x)$, allows to predict the hash of any string that begins with $\text{MDPAD}(x)$":

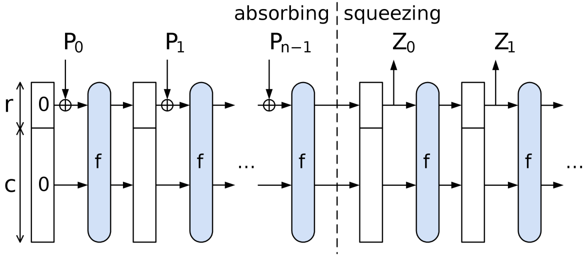


Mitigate length-extension attack

Length-extension attack possible because the hash **contains the full internal state**

Solutions:

- **Wide-pipe construction**: Output only, e.g., half of the final hash. Used in SHA-512/224 and SHA-512/256 (SHA-2 family), while SHA-512 and SHA-256 are vulnerable to this attack.
- **Sponge construction**: two phases absorb & squeeze, used in SHA-3



Building à compression function

How to obtain compression functions?

Building a compression function $h\colon \{0,1\}^{n+t} \to \{0,1\}^n$:

- From scratch
- From a block cipher (e.g. AES) $\mathcal{E}$, choose what defines the key/message, add feedforward (otherwise invertible):
  - Davies–Meyer: $h(x\|k) := \mathcal{E}_k(x) \oplus x$ (e.g. used in SHA-2 with a custom cipher)
  - Matyas–Meyer–Oseas: $h(x\|x') := \mathcal{E}_{g(x)}(x') \oplus x'$
  - Miyaguchi–Preneel: $h(x\|x') := \mathcal{E}_{g(x)}(x') \oplus x' \oplus x$
  - Hirose

Proofs typically done in the **Ideal Cipher Model** to model $\mathcal{E}$

Security models

Ideal models

In Ideal Cipher Model (resp. Random Oracle Model), we assume a function behaves like a randomly uniformly sampled permutation (resp. function), that the parties (including the attacker) can only access in a black-box way (for ideal ciphers, the parties can also ask for the inverse of the function). But:

- In practice, me must instantiate it with an actual permutation (resp. function), e.g. AES (resp. SHA-3):
  $\Rightarrow$ we have then **heuristic security** (no reduction)
- There exists (pathologic) schemes secure in the ROM [Bellare, Boldyreva, Palacio 03] but impossible to instantiate
- Yet, no non-pathological construction is known to be secure in the ROM but insecure in practice

Idealized model $\neq$ standard model

Microsoft needed a hash function for ROM integrity check for the XBOX:

- They used Tiny Encryption Algorithm (TEA, block-cipher) as a basic cipher with Davies-Meyer [1]
- Issue: for any k, easy to find k' such that $\text{TEA}_k(m) = \text{TEA}_{k'}(m)$ (like flip a bit of k), and $\Rightarrow$ Trivial to get a collision:

$$\text{DM} - \text{TEA}(x \| k') = \text{TEA}_{k'}(x) = \text{TEA}_k(x) = \text{DM} - \text{TEA}(x \| k)$$

- Yet, TEA is still a good PRP (once we sample a random k)!

[1] Details of attack in [Steil, 2005]
https://events.ccc.de/congress/2005/fahrplan/events/559.en.html

Random-Oracle Model

Random-Oracle Model (ROM)

A protocol is said to be defined in the Random-Oracle Model (ROM) if all parties (including honest parties when defining the protocol and adversaries) have oracle access to H defined as:

Random Oracle

$T :=$ empty assoc. array

$\underline{H(x \in \{0,1\}^*):}$

  if $T[x]$ undefined:

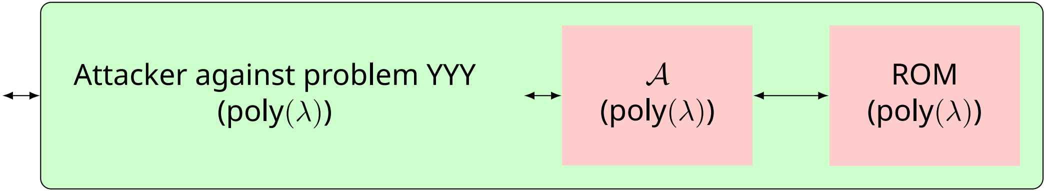
    $T[x] \xleftarrow{\$} \{0,1\}^{\mathsf{out}}$

  return $T[x]$

Random-Oracle Model

Remarks:

- **Lazy sampling** instead of sampling full H = needed in reductions to have polynomial adversary:



- **ROM $\neq$ PRF!!!** In ROM the parties have only oracle access to H, in PRF the parties can also see the "code" of H. This allow new proof techniques!

Ideal-Cipher Model

Ideal-Cipher Model = pick a random permutation for each key

Ideal-Cipher Model (ICM)

A protocol is said to be defined in the Ideal-Cipher Model (ICM) if all parties (including honest parties when defining the protocol and adversaries) have oracle access to F and F^{-1} defined as:

Ideal-Cipher Model

$T :=$ empty assoc. array of assoc. array

$F(k \in \{0,1\}^\lambda, x \in \{0,1\}^{\mathsf{blen}})$:

- if $T[k][x]$ undefined:
  - $T[k][x] \xleftarrow{\$} \{0,1\}^{\mathsf{blen}} \setminus T[k]$.values
- return $T[k][x]$

$F^{-1}(k \in \{0,1\}^\lambda, y \in \{0,1\}^{\mathsf{blen}})$:

- if $\exists x$ s.t. $T[k][x] = y$:
  - return x
- else:
  - $x \xleftarrow{\$} \{0,1\}^{\mathsf{blen}} \setminus T[k]$.keys
  - $T[k][x] := y$
  - return x

Main hash functions

Comparison of the main hash functions

Long story short: use SHA-3, or SHA-2 (but not for MAC). **Never use MD5**, SHA-0, SHA-1

Algorithm and variant		Output size (bits)	Internal state size (bits)	Block size (bits)	Rounds	Operations	Security against collision attacks (bits)	Security against length extension attacks (bits)	Performance on Skylake (median cpb)[61]		First published
									Long messages	8 bytes	
MD5 (as reference)		128	128 (4 × 32)	512	4 (16 operations in each round)	And, Xor, Or, Rot, Add (mod 2^{32})	≤ 18 (collisions found)[62]	0	4.99	55.00	1992
SHA-0		160	160 (5 × 32)	512	80	And, Xor, Or, Rot, Add (mod 2^{32})	< 34 (collisions found)	0	≈ SHA-1	≈ SHA-1	1993
SHA-1							< 63 (collisions found)[63]		3.47	52.00	1995
SHA-2	*SHA-224* *SHA-256*	224 256	256 (8 × 32)	512	64	And, Xor, Or, Rot, Shr, Add (mod 2^{32})	112 128	32 0	7.62 7.63	84.50 85.25	2004 2001
	SHA-384	384	512 (8 × 64)	1024	80	And, Xor, Or, Rot, Shr, Add (mod 2^{64})	192	128	5.12	135.75	2001
	SHA-512	512					256	0[64]	5.06	135.50	2001
	SHA-512/224 *SHA-512/256*	224 256					112 128	288 256	≈ SHA-384	≈ SHA-384	2012
SHA-3	*SHA3-224* *SHA3-256* *SHA3-384* *SHA3-512*	224 256 384 512	1600 (5 × 5 × 64)	1152 1088 832 576	24[65]	And, Xor, Rot, Not	112 128 192 256	448 512 768 1024	8.12 8.59 11.06 15.88	154.25 155.50 164.00 164.00	2015
	SHAKE128 *SHAKE256*	*d* (arbitrary) *d* (arbitrary)		1344 1088			min(*d*/2, 128) min(*d*/2, 256)	256 512	7.08 8.59	155.25 155.50	