SCIENCE
PASSION
TECHNOLOGY

# Cryptography 2: Symmetric Authentication

Maria Eichlseder

Information Security – WT 2019/20

› www.iaik.tugraz.at/infosec

# You Are Here

## Crypto 1

### Introduction to InfoSec & Crypto

- Terminology
- Security notions
- Keys, Kerckhoffs' principle

## Crypto 2

### Symmetric Authentication

- ➔ Integrity
- Hash functions
- MACs (Message Authentication)

## Crypto 3

### Symmetric Encryption

- ➔ Confidentiality
- AEAD (Auth. Encryption)
- Symmetric primitives

## Crypto 4

### Asymmetric Cryptography

- ➔ Establishing communication
- Key exchange
- Signatures
- Asymmetric primitives

# Recap of Last Week

- Information security protects assets against adversaries
  - Break the chain:
    Security Property 🔗 Threat 🔗 Vulnerability 🔗 Attack
- Cryptography is the mathematical foundation of secure communication
  - Algorithms to transform data so it can be sent over untrusted channels
  - Creates a new asset: the key

# Administrative Update

- If you’re a 1-person team for the practicals:
  - Try to find a partner right after today’s lecture
  - We may merge teams
- New curricula for CS/ICE (KU InfoSec = IIS+RKN) and SEM (KU InfoSec = IIS):
  - This may be a small disadvantage (SEM 16U) or advantage (SEM 19U)
  - SEM 16U: contact your Dean of Studies (Denis Helic) for options (Freifach)

# Outline

# Cryptographic Authentication

## Introduction

# Authenticity and Integrity

## Message Authentication

- Authenticity: Verify the source of the message
- Integrity: Verify that the message has not been modified while in transit

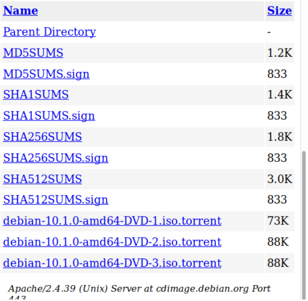
## Entity Authentication

- Verify the identity of a communication endpoint (device, user) based on possession of some cryptographic identifier (password, key, . . . )

# Examples (1): File Checksums

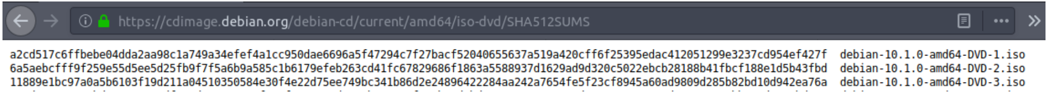
| Name | Size |
|---|---|
| Parent Directory | - |
| MD5SUMS | 1.1K |
| MD5SUMS.sign | 833 |
| SHA1SUMS | 1.3K |
| SHA1SUMS.sign | 833 |
| SHA256SUMS | 1.7K |
| SHA256SUMS.sign | 833 |
| SHA512SUMS | 2.8K |
| SHA512SUMS.sign | 833 |
| debian-10.1.0-amd64-DVD-1.iso | 3.6G |
| debian-10.1.0-amd64-DVD-2.iso | 4.4G |
| debian-10.1.0-amd64-DVD-3.iso | 4.4G |

*Apache/2.4.39 (Unix) Server at cdimage.debian.org Port*



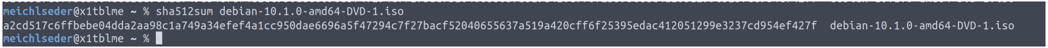
| Name | Size |
|---|---|
| Parent Directory | - |
| MD5SUMS | 1.2K |
| MD5SUMS.sign | 833 |
| SHA1SUMS | 1.4K |
| SHA1SUMS.sign | 833 |
| SHA256SUMS | 1.8K |
| SHA256SUMS.sign | 833 |
| SHA512SUMS | 3.0K |
| SHA512SUMS.sign | 833 |
| debian-10.1.0-amd64-DVD-1.iso.torrent | 73K |
| debian-10.1.0-amd64-DVD-2.iso.torrent | 88K |
| debian-10.1.0-amd64-DVD-3.iso.torrent | 88K |

*Apache/2.4.39 (Unix) Server at cdimage.debian.org Port*



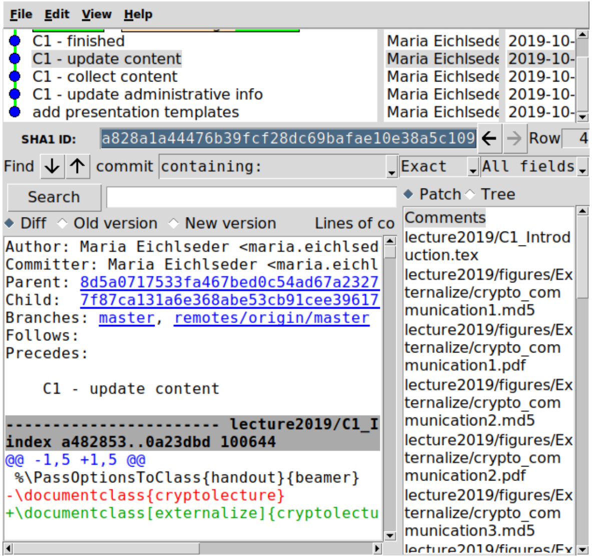
https://cdimage.debian.org/debian-cd/current/amd64/iso-dvd/SHA512SUMS

```
a2cd517c6ffbebe04dda2aa98c1a749a34efef4a1cc950dae6696a5f47294c7f27bacf52040655637a519a420cff6f25395edac412051299e3237cd954ef427f  debian-10.1.0-amd64-DVD-1.iso
6a5aebcfff9f259e55d5ee5d25fb9f7f5a6b9a585c1b6179efeb263cd41fc67829686f1863a5588937d1629ad9d320c5022ebcb28188b41fbcf188e1d5b43fbd  debian-10.1.0-amd64-DVD-2.iso
11889e1bc97a0a5b6103f19d211a04510350584e30f4e22d75ee749bc341b86d2e24896422284aa242a7654fe5f23cf8945a60ad9809d285b82bd10d942ea76a  debian-10.1.0-amd64-DVD-3.iso
```



```
meichlseder@x1tblme ~ % sha512sum debian-10.1.0-amd64-DVD-1.iso
a2cd517c6ffbebe04dda2aa98c1a749a34efef4a1cc950dae6696a5f47294c7f27bacf52040655637a519a420cff6f25395edac412051299e3237cd954ef427f  debian-10.1.0-amd64-DVD-1.iso
meichlseder@x1tblme ~ % █
```

# Examples (2): Commit IDs and File Versions



```
C1 - finished                          Maria Eichlsede 2019-10-
C1 - update content                    Maria Eichlsede 2019-10-
C1 - collect content                   Maria Eichlsede 2019-10-
C1 - update administrative info        Maria Eichlsede 2019-10-
add presentation templates             Maria Eichlsede 2019-10-
```

SHA1 ID: a828a1a44476b39fcf28dc69bafae10e38a5c109

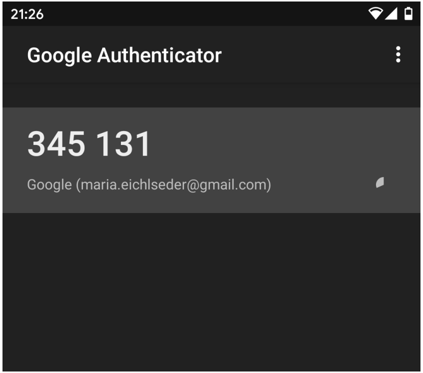
```
Author: Maria Eichlseder <maria.eichlsed
Committer: Maria Eichlseder <maria.eichl
Parent: 8d5a0717533fa467bed0c54ad67a2327
Child:  7f87ca131a6e368abe53cb91cee39617
Branches: master, remotes/origin/master
Follows:
Precedes:

    C1 - update content

----------------------- lecture2019/C1_I
index a482853..0a23dbd 100644
@@ -1,5 +1,5 @@
 %\PassOptionsToClass{handout}{beamer}
-\documentclass{cryptolecture}
+\documentclass[externalize]{cryptolectu
```

lecture2019/C1_Introduction.tex
lecture2019/figures/Externalize/crypto_communication1.md5
lecture2019/figures/Externalize/crypto_communication1.pdf
lecture2019/figures/Externalize/crypto_communication2.md5
lecture2019/figures/Externalize/crypto_communication2.pdf
lecture2019/figures/Externalize/crypto_communication3.md5
lecture2019/figures/Ex

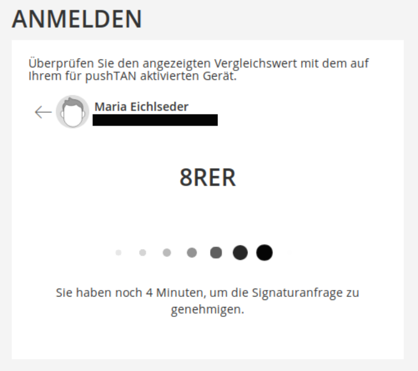
# Examples (3): Mobile TANs, 2-Factor-Authentication



**Google Authenticator**

345 131

Google (maria.eichlseder@gmail.com)



ANMELDEN

Überprüfen Sie den angezeigten Vergleichswert mit dem auf Ihrem für pushTAN aktivierten Gerät.

**Maria Eichlseder**
[redacted]

8RER

Sie haben noch 4 Minuten, um die Signaturanfrage zu genehmigen.

# Cryptographic Schemes for Message Authentication

Cryptographic schemes for message authentication compute a short, fixed-length Tag $T$ from the Message $M$ and (in some cases) a Key $K$.

**Hash Function $\mathcal{H}$**

Unkeyed

- Anyone can compute $T$
- Anyone can verify $T$

**MAC $\mathcal{H}_{K_{AB}}$**

Symmetric Key $K_{AB}$

- $A, B$ can compute $T$
- $A, B$ can verify $T$

**Signature $\mathcal{S}_{K_A}$**

Asymmetric Key $K_A$

- $A$ can compute $T$
- Anyone can verify $T$

# Application Examples (1)

Hash functions:

- File download with checksum
- Identifier for files and commits
- Identification of identical files (for deduplication, detecting changes)
- Linking blockchain blocks + proof-of-work for timestamping
- Storing login passwords securely (requires special password hash function!)
- Announcing commitment to something you only reveal later
  (no, this has nothing to do with hashtags)

# Application Examples (2)

MACs:

- Challenge-response in multifactor authentication (mobile TANs)
- Message integrity in secure communication protocols (TLS, SSH, …)

Signatures (in two weeks):

- Electronic signature of documents, Handysignatur
- Signing emails with PGP
- Entity authentication and X509 certificates in secure communication protocols (TLS, SSH, …)

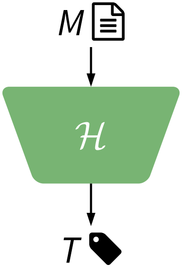
# Hash Functions

## Keyless Authentication

# Hash Functions – Definition

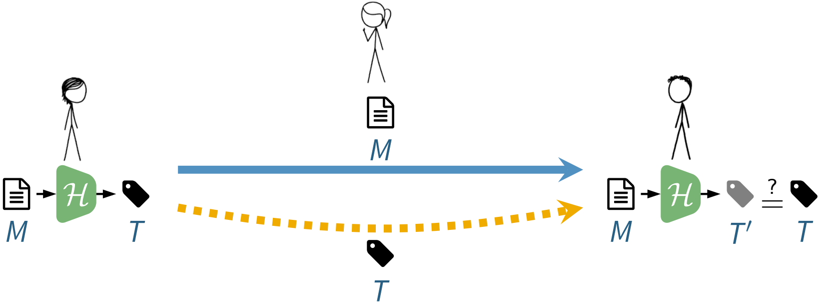
A cryptographic hash function $\mathcal{H}$ maps a message $M$ (a bitstring) of arbitrary bitlength to a $t$-bit tag $T$ that serves as fingerprint/checksum for $M$:

$$\mathcal{H} : \mathbb{F}_2^* \to \mathbb{F}_2^t, \qquad \mathcal{H}(M) = T$$



The challenge of protecting the authenticity of $M$ is transformed into protecting $T$.

# Hash Functions – Application



1. Alice computes $T = \mathcal{H}(M)$
2. Alice transmits $M$ to Bob (over an insecure channel controlled by Eve)
3. Alice separately transmits $T$ to Bob (over a secure channel).
4. Bob re-computes $T' = \mathcal{H}(M)$ and verifies that $T' = T$.

# Not to be Confused with…

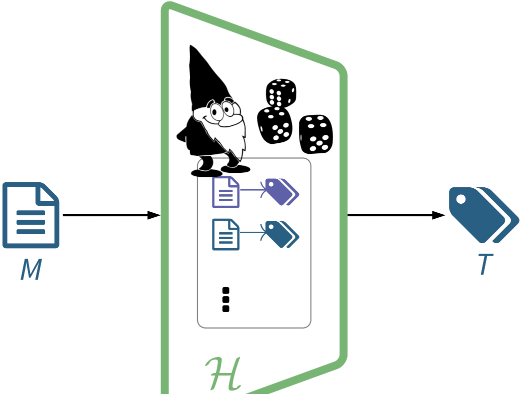
(Cryptographic) hash functions are not to be confused with…

- Password Hash Functions or Key Derivation Functions, which map a password to a password hash or key and have stronger requirements.
- Non-Cryptographic Hash Functions, which map values to reasonably uniformly distributed values (e.g., index for hash tables). They have different, weaker requirements and no attacker.
- Error-Detecting/Correcting Codes and Checksums like CRC32 to correct accidental transmission errors (no attacker). They are usually shorter and only guarantee detection of specific modifications like single bitflips.

# Security Notion – Random Oracles

Idealized model of a hash function: The (truncated) Random Oracle

- Returns a random bitstring for every possible query
- Same input query → same output



# 3 Security Properties of Hash Functions

## Preimage resistance:

Given a tag $T$, it must be infeasible for an attacker to find any message $M$ such that $T = \mathcal{H}(M)$.

Generic complexity: about $2^t$ trials

## Second preimage resistance:

Given a message $M$, it must be infeasible for an attacker to find any second message $M' \neq M$ such that $\mathcal{H}(M') = \mathcal{H}(M)$.

Generic complexity: about $2^t$ trials

## Collision resistance:

It must be infeasible for an attacker to find any two different messages $M, M'$ such that $\mathcal{H}(M') = \mathcal{H}(M)$.

Generic complexity: about $2^{t/2}$ trials (!)

# The Birthday Paradox

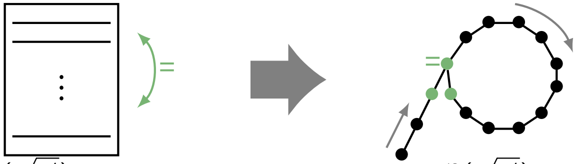
**The Birthday Paradox**

In a class of only 23 people, there is a good chance (about 50 %) that 2 of them have the same birthday.

Application to the collision resistance of $\mathcal{H}$:

- The attacker collects a list of tags for about $\sqrt{2^t} = 2^{t/2}$ different messages.
- Now they have $\binom{2^{t/2}}{2} \approx \frac{(2^{t/2})^2}{2} = \frac{1}{2} \cdot 2^t$ candidate message pairs.
- The probability of a collision for one pair is $\frac{1}{2^t}$.
- So it is quite likely that there is at least one collision in the list.

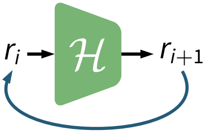
# Rho-Method: Memoryless Collision-Finding



$\mathcal{O}(\sqrt{2^t})$ time
$\mathcal{O}(\sqrt{2^t})$ space

$\mathcal{O}(\sqrt{2^t})$ time
$\mathcal{O}(1)$ space

- Define a sequence $r_{i+1} = \mathcal{H}(r_i)$ for $i \geq 0$ by starting from some value $r_0$ and iteratively applying the function $\mathcal{H}$
- If $r_j = r_k$, then $r_{j+1} = \mathcal{H}(r_j) = \mathcal{H}(r_k) = r_{k+1}$
- After an initial tail, the sequence turns cyclic (“$\rho$”)



# Rho-Method: Memoryless Collision-Finding

- How to find the collision $r_j = r_k$?
  Cycle finding algorithms such as Floyd's algorithm ("tortoise and hare"):

**Find cycle length $\lambda$:**

```
$r_i \leftarrow r_0, \quad r_{2i} \leftarrow r_0$
repeat
    $r_i \leftarrow \mathcal{H}(r_i)$
    $r_{2i} \leftarrow \mathcal{H}(\mathcal{H}(r_{2i}))$
until $r_i = r_{2i}$
$\lambda \leftarrow 2i - i = i$
```

**Find prefix length $\mu$:**

```
$r_j \leftarrow r_0, \quad r_{j+\lambda} \leftarrow r_i$
repeat
    $r_j \leftarrow \mathcal{H}(r_j)$
    $r_{j+\lambda} \leftarrow \mathcal{H}(r_{j+\lambda})$
until $r_j = r_{j+\lambda}$
$\mu \leftarrow j + \lambda - \lambda = j$
```

- Runtime depends on cycle length $\lambda$ and prefix length $\mu$.
  The expected value of both is about $\sqrt{2^t}$ (times a small constant).
  → Complexity: $\mathcal{O}(\sqrt{2^t})$ time and $\mathcal{O}(1)$ memory

# How much computation time, memory, data is practically “feasible”?

| | **Time [cipher calls]** | **Memory [cipher states]** | **Data [queries]** |
|---|---|---|---|
| $2^{32}$ | trivial | easy | practical |
| $2^{48}$ | easy[1] | practical | practical |
| $2^{64}$ | practical[2] | unpractical | unpractical |
| $2^{80}$ | unpractical[3] | infeasible | infeasible |
| $2^{128}$ | infeasible[4] | | |
| $2^{256}$ | infeasible | | |

[1] easy: you can do this.

[2] practical: *you* probably can’t do it, but a powerful attacker possibly can.

[3] unpractical: probably no-one can currently do this, but better not to rely on it.

[4] infeasible: no-one can do this.

# Security Levels

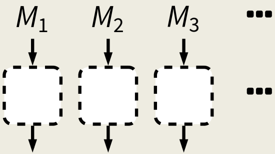
$n$-bit Security means that an attacker would need about $2^n$ computation time (measured in “number of cipher evaluations”) to have a good success probability of breaking the scheme.

- 128-bit Security is widely seen as a good choice for most applications.
  ➔ Hash output size should be $2 \times 128 = 256$ bits (birthday paradox).
- 256-bit Security may be preferable for special applications and for higher post-quantum security

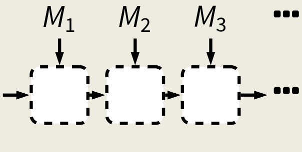
You sometimes see $\mathcal{O}$-notation for security claims. This is usually not a meaningful security claim – the constants hidden in the $\mathcal{O}$-notation can make a big difference!

# Processing Long Messages by Iterating a Primitive

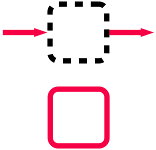
## Translate Data Blocks
(e.g., encryption)



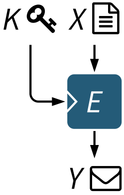
## Accumulate Data
(e.g., authentication, hashing)



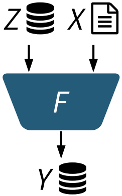
- Today: the mode
- Next week: the primitive (and more modes)



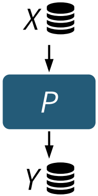
# Symmetric Primitives



Block Cipher
(BC)



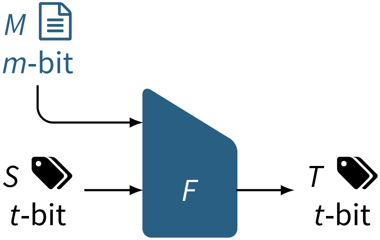
Compression
Function



Permutation

…

# Compression Functions

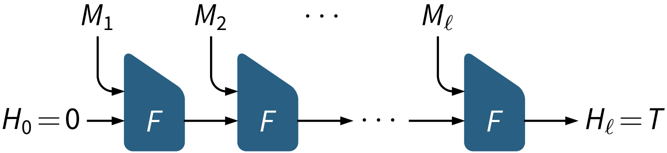


Compress

- One fixed mapping
- $2^{t+m}$ possible inputs
- $2^t$ possible outputs
  - $t$ bounds the security level
  - Small $t$: Danger of collisions
  - Large $t$: Higher transmission cost

# Merkle–Damgård Hashing (MD)

Hashing by iterating a compression function $F$:



1. Split message $M$ into $m$-bit blocks $M_1, M_2, \ldots, M_\ell$
2. Start iteration with fixed initial value $H_0$
3. For $i = 1, \ldots, \ell$: Compress old state $H_{i-1}$ and message block $M_i$ to new state $H_i$
4. Return the final state (chaining value) $H_\ell$ as the tag $T$

# Merkle–Damgård Hashing (MD) – Padding and Security

❓ What if the length of $M$ is not a multiple of the block size of $m$ bits?

➔ Requires injective padding to produce a multiple of the block length $m$:

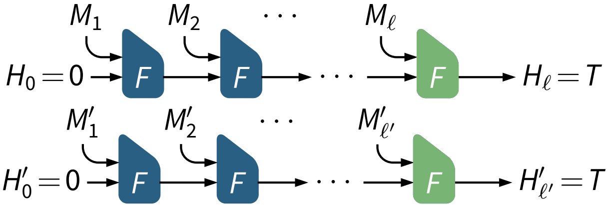
| | | | | | |
|---|---|---|---|---|---|
| | 80 | 00 | $\dots$ | 00 | bit-length of $M$ as a 64-bit integer |

- This padding is specified as part of the mode of operation
- It is always applied, not only if the last block is a partial block!

➔ **Theorem**: If $F$ is collision resistant, then $\mathcal{H}$ is collision resistant (why?)

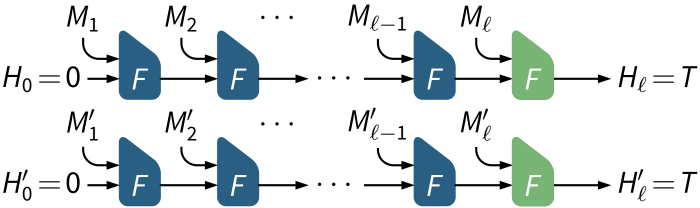
# MD Theorem: If $F$ is collision resistant, then $\mathcal{H}$ is collision resistant (1)

- **Proof by contraposition**: We show that if Eve finds a hash collision for $\mathcal{H}$ ("$\mathcal{H}$-collision"), she also knows a compression collision for $F$ ("$F$-collision").
- So assume that Eve knows two messages $M \neq M'$ such that $\mathcal{H}(M) = \mathcal{H}(M')$. Let $M_1 \| M_2 \| \dots \| M_\ell$ be the blocks of padded $M$ and $M'_1 \| M'_2 \| \dots \| M'_{\ell'}$ those of $M'$.



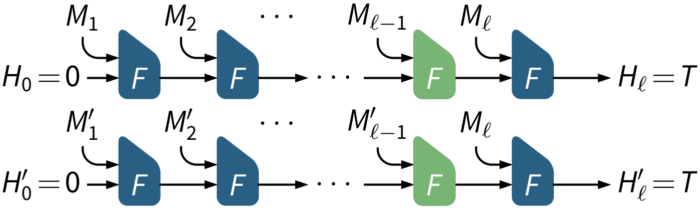
- First consider the case that $M$ and $M'$ have different bitlength $|M| \neq |M'|$. The length is encoded in the last 64 bits of the last block, so $M_\ell \neq M'_{\ell'}$. Thus, Eve has an $F$-collision: $(M_\ell, H_{\ell-1}) \neq (M'_{\ell'}, H'_{\ell'-1})$, but both compress to $T$.

# MD Theorem: If $F$ is collision resistant, then $\mathcal{H}$ is collision resistant (2)



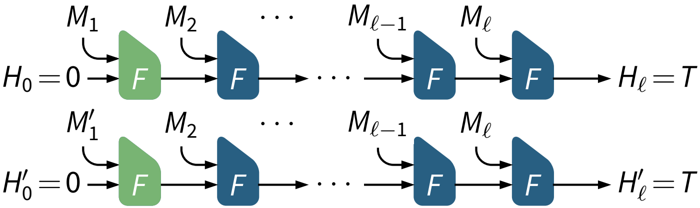
- Next, consider the case that $M$ and $M'$ have the same bitlength $|M| = |M'|$.
- In case their last blocks $M_\ell, M'_\ell$ *or* the previous chaining blocks $H_{\ell-1}, H'_{\ell-1}$ are still different, Eve again knows an $F$-collision with the same reasoning.
- So we consider the case $M_\ell = M'_\ell$ and $H_{\ell-1} = H'_{\ell-1}$. Either there is a difference in the *previous* inputs $(M_{\ell-1}, H_{\ell-2}) \neq (M'_{\ell-1}, H'_{\ell-2})$ and Eve knows an $F$-collision with output $H_{\ell-1} = H'_{\ell-1}$, or there is no difference and we can repeat the argument for the block before.

# MD Theorem: If $F$ is collision resistant, then $\mathcal{H}$ is collision resistant (2)



- Next, consider the case that $M$ and $M'$ have the same bitlength $|M| = |M'|$.
- In case their last blocks $M_\ell, M'_\ell$ *or* the previous chaining blocks $H_{\ell-1}, H'_{\ell-1}$ are still different, Eve again knows an $F$-collision with the same reasoning.
- So we consider the case $M_\ell = M'_\ell$ and $H_{\ell-1} = H'_{\ell-1}$. Either there is a difference in the *previous* inputs $(M_{\ell-1}, H_{\ell-2}) \neq (M'_{\ell-1}, H'_{\ell-2})$ and Eve knows an $F$-collision with output $H_{\ell-1} = H'_{\ell-1}$, or there is no difference and we can repeat the argument for the block before.

# MD Theorem: If $F$ is collision resistant, then $\mathcal{H}$ is collision resistant (3)



- We can repeat this argument backwards until we have either found an $F$-collision $(M_i, H_{i-1}) \neq (M'_{i'}, H'_{i'-1})$ with $H_i = H'_{i'}$ or we reach the first block $M_1$.
- If we reach the first blocks $M_1, M'_1$, then these cannot be identical: That would mean that the entire messages $M = M'$ are identical, which contradicts our initial assumption that Eve has a $\mathcal{H}$-collision.
  Thus, Eve has an $F$-collision $(M_1, 0) \neq (M'_1, 0)$, which both compress to $H_1 = H'_1$.
- In summary, Eve always finds an $F$-collision $(M_i, H_{i-1}) \neq (M'_{i'}, H'_{i'-1})$ with $H_i = H'_{i'}$.

# Standardized Hash Functions and TLS 1.3

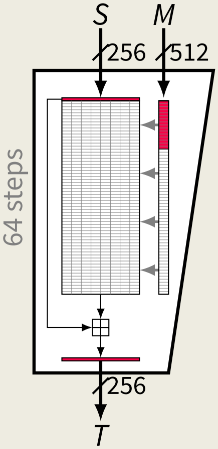
In TLS, hash functions are used for signing and to build MACs. They are standardized by NIST (`SHA` = Secure Hash Algorithm) and follow the MD design.

| Family | Hash size | Security | TLS 1.2 | TLS 1.3 |
|---|---|---|---|---|
| `MD5` | 128 bits | broken | ✓ | ✗ |
| `SHA-1` | 160 bits | broken | ✓ | ✓ |
| `SHA-2` | 224 bits | 112 bits | ✓ | ✗ |
| | 256 bits | 128 bits | ✓ | ✓ |
| | 384 bits | 192 bits | ✓ | ✓ |
| | 512 bits | 256 bits | ✓ | ✓ |
| `SHA-3` | * | * | not yet | not yet |

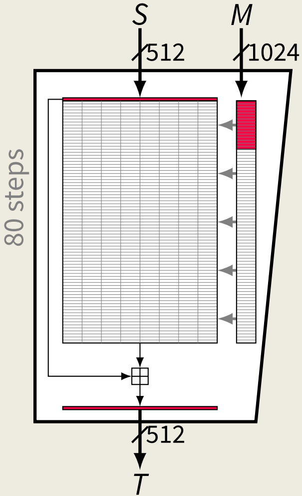
✓ supported ✓ legacy certificates only ✗ not supported

# The Compression Function of `SHA-2` (2 Sizes)

**`SHA-256`**



**`SHA-512`**

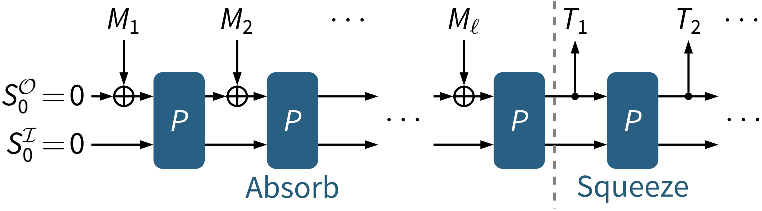


# The SHA-3 Competition (2007–2012)

- SHA-3 – Secure Hash Algorithm
- Goals: A hash function to complement `SHA-2`
  - `SHA-2` is secure, but also similar to the broken `SHA-1`, `MD5`
  - New design should look very different
- Organized by NIST (US Institute of Standards and Technology)
- Announced in 2007, 64 submissions from 200 cryptographers
- Winner: `Keccak/SHA-3` by Bertoni, Daemen, Peeters, Van Assche, Van Keer

  Other Finalists: `BLAKE`, `Grøstl` 🟥, `JH`, `Skein`

# Hashing with Permutations: The Sponge Construction



- Large state with two parts:
  - $r$-bit outer part $S^{\mathcal{O}}$ (“rate” $r$) → message/tag block size
  - $c$-bit inner part $S^{\mathcal{I}}$ (“capacity” $c$) → security level up to $2^{c/2}$
- State update with unkeyed $(r + c)$-bit permutation $P$ (`SHA-3`: $r + c = 1600$)

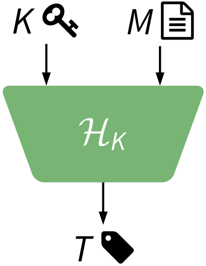
# Message Authentication Codes

## Symmetric-Key Authentication

# Message Authentication Codes (MAC) – Definition

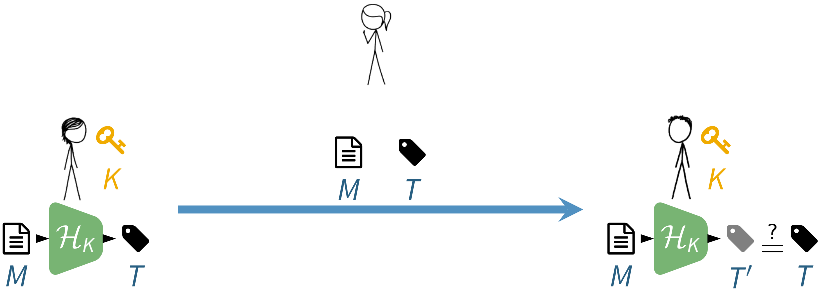
A Message Authentication Code is a keyed hash function $\mathcal{H}_K$ that maps a $k$-bit key $K$ and a message $M$ of arbitrary length to a $t$-bit tag $T$ to protect the integrity and authenticity of $M$:

$$\mathcal{H}_K : \mathbb{F}_2^k \times \mathbb{F}_2^* \to \mathbb{F}_2^t, \qquad \mathcal{H}_K(M) = T$$



The challenge of protecting the authenticity of $M$ is transformed into protecting $K$.

# Message Authentication Codes (MAC) – Application



1. Alice and Bob share a secret key $K$.
2. Alice computes $T = \mathcal{H}.(K, M) = \mathcal{H}_K(M)$.
3. Alice transmits $M$ and $T$ to Bob (over an insecure channel controlled by Eve).
4. Bob re-computes $T' = \mathcal{H}_K(M)$ and verifies that $T' = T$.

# Security Notion for Authenticity – Unforgeability

**Unforgeability**

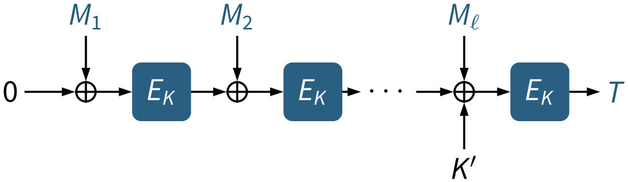
It is infeasible for an attacker to produce (forge) any new, valid message-tag pair $(M, T)$ even if they can query tags for any other messages of their choice.

Generic attacks on MACs:

- Exhaustive key search – Expected complexity: $2^k$ “offline” trials
- Guess the tag – Expected complexity: $2^t$ “online” verification trials

# How to Construct a MAC?

- From a Hash Function $\mathcal{H}$
  - Feed the key $K$ and message $M$ into the hash $\mathcal{H}$, e.g., $\mathcal{H}_K(M) = \mathcal{H}(M\|K)$
  - Example: `HMAC-SHA2` (`HMAC` = `Hash-based MAC`)
- By using a keyed primive, such as a Block Cipher $E_K(M)$
  - Example: `CMAC` (`C` = `CBC` = `Cipher Block Chaining`)



# Entity Authentication Protocols

# Authentication Protocols

Entity Authentication aka Identification – (*not* message authentication)

- Access control, login
- As part of communication protocols

Entities:

- The Prover claims an identity
- The Verifier wants evidence of the prover’s identity

# Authentication Factors

- What someone **knows**: Password, PIN, …
- What someone **has**: Smartcard, token, mobile, …
- What someone **is**: Fingerprint, face, voice, …

Multi-factor authentication: Smardcard + PIN, Password + mobile TAN, …

- A key can be what someone knows (password) or has (key stored on device)
- In this course, we won't go into details on biometrics.
  It's a separate field of research based on computer vision, biology, etc. and not as "open source" as crypto (proprietary algorithms)

# Passwords

**Naive password protocol**

Setup: Prover $A$ 👤 chooses password $K_A$ 🔑, verifier $B$ 🗄 stores $(A, K_A)$

Identification:

Prover $A$ 👤

Verifier $B$ 🗄

$A$ 👤, $K_A$ 🔑

accept if
$(A, K_A)$ stored

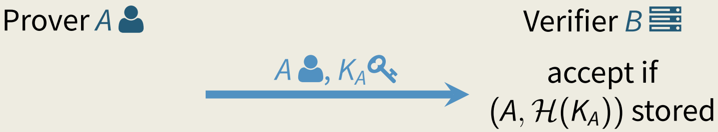
- Attacher $C$ can eavesdrop $K_A$ (replay attack)
- $B$'s stored table of passwords vulnerable
- Entropy of $K_A$?

# Passwords

**Passwords with Hash function $\mathcal{H}$**

Setup: Prover $A$ chooses password $K_A$, verifier $B$ stores $(A, \mathcal{H}(K_A))$

Identification:



Prover $A$

Verifier $B$

$A$, $K_A$

accept if
$(A, \mathcal{H}(K_A))$ stored

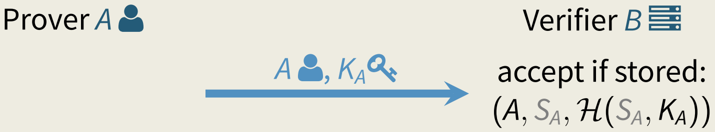
- Advantage: Stored tables less vulnerable
- Still assumes secure transmission
- If table leaks: still allows mass dictionary attack

# Passwords

## Passwords with Hash function $\mathcal{H}()$ and Salt $S_A$

**Setup**: Prover $A$ chooses password $K_A$, verifier $B$ chooses salt $S_A$, stores $(A, S_A, \mathcal{H}(S_A, K_A))$

**Identification**:



Prover $A$

Verifier $B$

$A$, $K_A$ $\longrightarrow$

accept if stored:
$(A, S_A, \mathcal{H}(S_A, K_A))$

- Advantage: No parallel attack on hash function $\mathcal{H} \rightarrow$ target individual users
- Table doesn’t leak users with same password

# Modern password hash functions

Requirements are slightly different from cryptographic hashes:

- Support long passwords and salts
- Not too fast, parameters to adapt speed (“Moore’s law”)
- Should need a lot of memory

Password hashing functions:

- `PBKDF2`
- `bcrypt`
- `scrypt`
- Have a look at the Password Hashing Competition (PHC): https://password-hashing.net/candidates.html

# Strong Authentication (Challenge-Response Protocols)

- Problem of Weak Authentication protocols like passwords:

  User always has to transmit the complete secret.
  This is potentially vulnerable to replay attacks.

- Idea of Strong Authentication protocols (Challenge-Response):

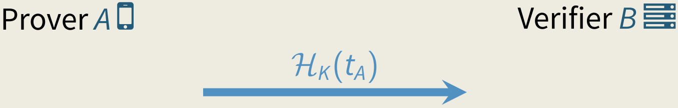
  Proving, not telling: Don't tell the Verifier the complete secret $x$.
  Instead "prove" possession by computing a function of $x$ plus some changing "challenge", such as a timestamp or a value sent by the verifier.

# Example: Time-based One-Time Password (TOTP)

2-step authentication for online services (Google, Github, banking, …):

1. User logs in with password
2. User provides (part of) TOTP from app, token, …

## TOTP



Prover $A$ 📱

Verifier $B$ 🗄

$\mathcal{H}_K(t_A)$

🔑 $K$: pre-shared secret key between app 📱 $A$ and server $B$
🕒 $t_A$: timestamp in 30-second steps (synchronized clock!)
🏷 $\mathcal{H}_K$: $d$ first digits of $\mathtt{HMAC}(\cdot)$

# Conclusion

# Conclusion

Message authentication can be done with

- No key: Hash function
- Symmetric key: Message Authentication Code (MAC)
- Asymmetric key: Signatures (coming soon...)

Entity authentication can be done with

- Weak authentication: Password (with salted password hash function)
- Strong authentication: Challenge-response (e.g., with MAC)

# Questions