Cryptography 1: Symmetric Authentication

Maria Eichlseder

Information Security – WT 2023/24

You Are Here

Crypto 0	Crypto 1	Crypto 2	Crypto 3
Introduction to InfoSec & Crypto	Symmetric Authentication	Symmetric Encryption	Asymmetric Cryptography
Terminology	**Authenticity**	**Confidentiality**	**Establishing communication**
Security notions	Secrets	AEAD (Auth. Encryption)	Key exchange
Keys, Kerckhoffs' principle	Hash functions	Symmetric primitives	Signatures
	MACs (Message Authentication)		Asymmetric primitives

Recap of Last Week

- **Information security** protects assets against adversaries
  - 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
  - Kerckhoffs' principle: Consider the algorithm public

Outline

Entity Authentication Protocols

- Weak Authentication (Passwords)
- Strong Authentication (Challenge-Response)

Hash Functions

- Definition and Security
- Generic Attacks
- Construction

Message Authentication Codes

- Definition and Security

Cryptographic Authentication

Introduction

Authenticity and Integrity

Entity Authentication

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

Message Authentication

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

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 443

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

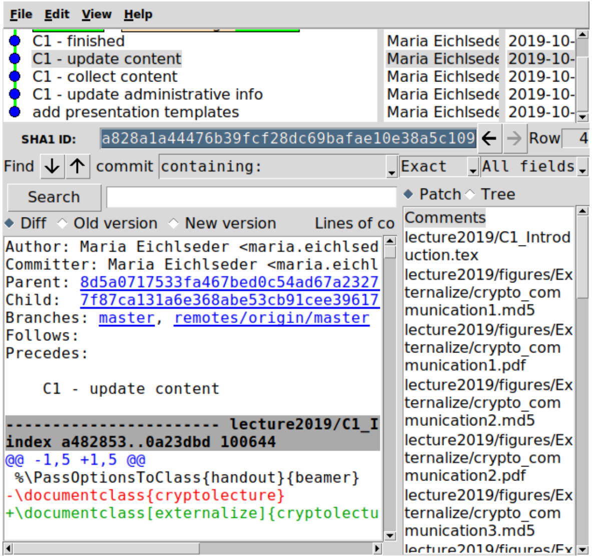
Apache/2.4.39 (Unix) Server at cdimage.debian.org Port 443

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

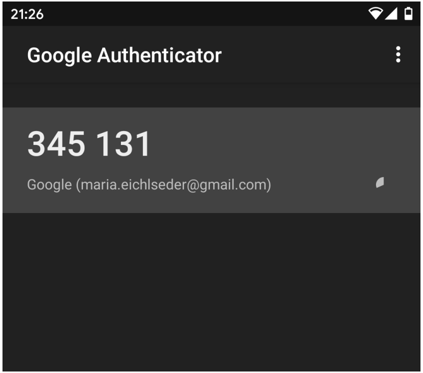
```
a2cd517c6ffbebe04dda2aa98c1a749a34efef4a1cc950dae6696a5f47294c7f27bacf52040655637a519a420cff6f25395edac412051299e3237cd954ef427f  debian-10.1.0-amd64-DVD-1.iso
6a5aebcfff9f259e55d5ee5d25fb9f7f5a6b9a585c1b6179efeb263cd41fc67829686f1863a5588937d1629ad9d320c5022ebcb28188b41fbcf188e1d5b43fbd  debian-10.1.0-amd64-DVD-2.iso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

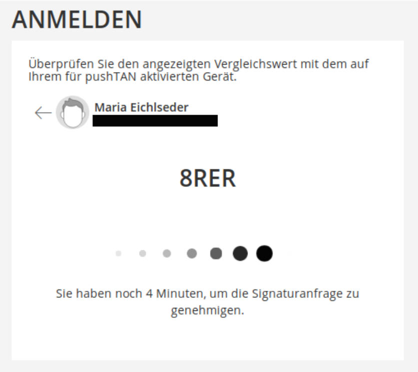
```
meichlseder@x1tblme ~ % sha512sum debian-10.1.0-amd64-DVD-1.iso
a2cd517c6ffbebe04dda2aa98c1a749a34efef4a1cc950dae6696a5f47294c7f27bacf52040655637a519a420cff6f25395edac412051299e3237cd954ef427f  debian-10.1.0-amd64-DVD-1.iso
meichlseder@x1tblme ~ %
```

Examples (2): Commit IDs and File Versions



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





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 (1)

Naive password protocol

Setup: Prover A chooses password K_A, verifier B stores (A, K_A)

Identification:

Prover A		Verifier B
	$\xrightarrow{A,\ K_A}$	accept if (A, K_A) stored

Storage: B's stored table of passwords vulnerable

Transport: Attacker C can eavesdrop K_A (replay attack)

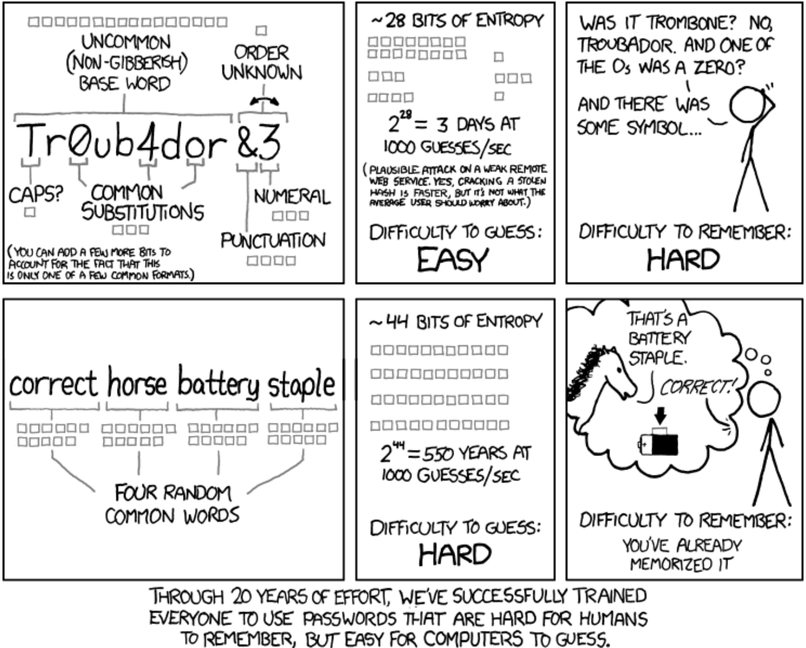
How strong is K_A?

Entropy $\mathrm{H}(X)$

Entropy is a measure for the "amount of randomness" of a random variable X.
It does not measure the quality of a particular value (that's actually impossible),
but of a **selection process** or **distribution** $p(x) := \mathbb{P}[X = x]$ of values $x \in \mathcal{X}$:

$$\mathrm{H}(X) = -\sum_{x\in\mathcal{X}} p(x) \log_2 p(x) = \mathbb{E}[-\log_2 p(X)]$$

- A 128-bit string where each bit is independently and uniformly randomly selected (= 2^{128} equally likely values) has an entropy of 128 bits.
- If some 128-bit values are *more likely than others*, then the entropy is *less than* 128 bits.
- A 128-bit string that is selected to be either $00\ldots0$ or $11\ldots1$ has an entropy of 1 bit.
- A password chosen uniformly at random from a list of 10 000 words has an entropy of $\log_2(10\,000) \approx 13.29$ bits.


UNCOMMON (NON-GIBBERISH) BASE WORD
ORDER UNKNOWN
Tr0ub4dor &3
CAPS?
COMMON SUBSTITUTIONS
NUMERAL
PUNCTUATION
(YOU CAN ADD A FEW MORE BITS TO ACCOUNT FOR THE FACT THAT THIS IS ONLY ONE OF A FEW COMMON FORMATS.)
~28 BITS OF ENTROPY
2^{28} = 3 DAYS AT 1000 GUESSES/SEC
(PLAUSIBLE ATTACK ON A WEAK REMOTE WEB SERVICE. YES, CRACKING A STOLEN HASH IS FASTER, BUT IT'S NOT WHAT THE AVERAGE USER SHOULD WORRY ABOUT.)
DIFFICULTY TO GUESS:
EASY
WAS IT TROMBONE? NO, TROUBADOR. AND ONE OF THE Os WAS A ZERO?
AND THERE WAS SOME SYMBOL...
DIFFICULTY TO REMEMBER:
HARD
correct horse battery staple
FOUR RANDOM COMMON WORDS
~44 BITS OF ENTROPY
2^{44} = 550 YEARS AT 1000 GUESSES/SEC
DIFFICULTY TO GUESS:
HARD
THAT'S A BATTERY STAPLE.
CORRECT!
DIFFICULTY TO REMEMBER:
YOU'VE ALREADY MEMORIZED IT
THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

Passwords (2)

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

- **Transport** still vulnerable, needs a secure channel
- Storage: Now less vulnerable
- If stored table leaks: still allows mass dictionary attack

Passwords (3)

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

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: Strong Authentication with TOTP

Time-based One-Time Password (**TOTP**)

Prover A $\xrightarrow{\mathcal{H}_K(t_A)}$ Verifier B

- t_A: timestamp in 30-second steps (synchronized clock!)
- K: pre-shared secret key between app A and server B
- $\mathcal{H}_K$: message authentication code (MAC) function

Usually as one of two factors in 2-Factor Authentication (2FA):

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

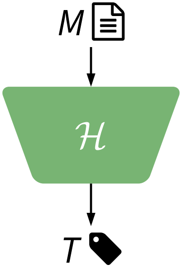
Hash Functions

Keyless Authentication

Hash Functions – Definition

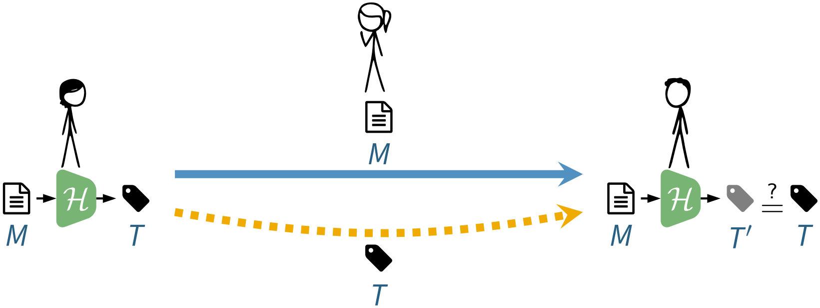
A **cryptographic hash function** $\mathcal{H}$ maps a message M (a bitstring) of arbitrary bitlength to a t-bit tag T that serves as fingerprint/checksum for M:

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



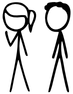
The challenge of protecting the authenticity of M is transformed into protecting T.

Hash Functions – Application

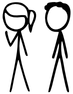


1. Alice computes $T = \mathcal{H}(M)$
2. Alice transmits M to Bob (over an insecure channel controlled by Eve)
3. Alice separately transmits T to Bob (over a secure channel).
4. Bob re-computes $T' = \mathcal{H}(M)$ and verifies that $T' = T$.

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.

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: maybe no-one can currently do this, but better not to rely on it.
4 infeasible: no-one can do this.

Security Levels

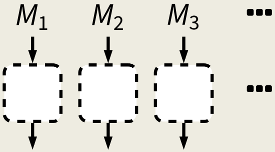
***n*-bit Security** means that an attacker would need about 2^n computation time (measured in "number of cipher evaluations") to have a good success probability of breaking the scheme.

- **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

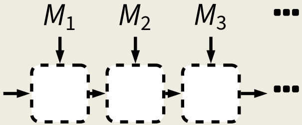
You sometimes see $\mathcal{O}$-notation for security claims. This is usually not a meaningful security claim – the constants hidden in the $\mathcal{O}$-notation can make a big difference!

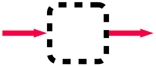
Processing Long Messages by Iterating a Primitive

Translate Data Blocks
(e.g., encryption)

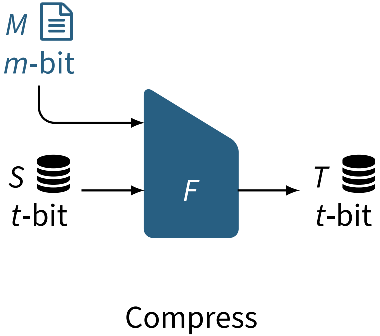


Accumulate Data
(e.g., authentication, hashing)



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

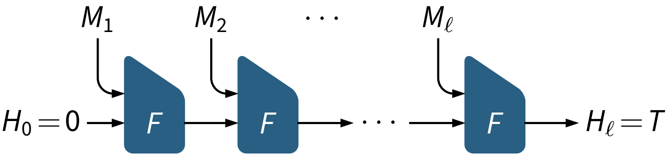
A Useful Primitive: Compression Functions



- F is a function with …
- 2^{t+m} possible inputs (M, S)
- 2^t possible outputs T
  - Smaller t: Lower security (collisions!)
  - Larger t: Lower performance

Merkle–Damgård Hashing (MD)

Hashing an arbitrarily long message M 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_ℓ 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	...	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

Application Examples for 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)

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

Not to be Confused with…

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

- Password Hash Functions or Key Derivation Functions like `PBKDF2`, 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.

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:

- ✅ `Argon2`
- ✅ `scrypt`
- ✅ `bcrypt` (legacy systems)
- ✅ `PBKDF2`

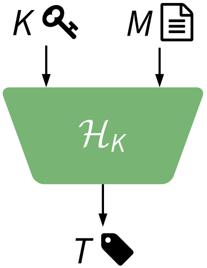
Message Authentication Codes

Symmetric-Key Authentication

Message Authentication Codes (MAC) – Definition

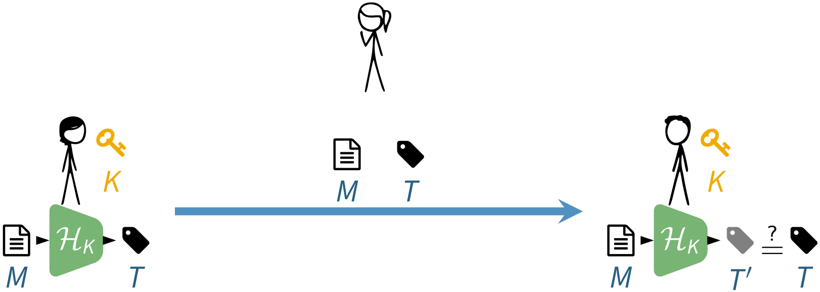
A Message Authentication Code is a keyed hash function $\mathcal{H}_K$ that maps a k-bit key K and a message M of arbitrary length to a t-bit tag T to protect the integrity and authenticity of M:

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



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 $T = \mathcal{H}_K(M)$.
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

Application Examples for MACs

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

Conclusion

Conclusion

Message authentication can be done with

- No key: Hash function
- Symmetric key: MAC; AEAD (coming soon...)
- 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)