Cryptographic hash functions and MACs

Myrto Arapinis
School of Informatics
University of Edinburgh

October 11, 2016
Introduction

Encryption $\Rightarrow$ confidentiality against eavesdropping

What about authenticity and integrity against an active attacker?

$\rightarrow$ cryptographic hash functions and Message authentication codes

$\rightarrow$ this lecture
Encryption $\Rightarrow$ confidentiality against eavesdropping

What about authenticity and integrity against an active attacker?
$\rightarrow$ cryptographic hash functions and Message authentication codes
$\rightarrow$ this lecture
One-way functions (OWFs)

A OWF is a function that is easy to compute but hard to invert:

**Definition (One-way)**

A function $f$ is a one-way function if for all $x$ there is no efficient algorithm which given $f(x)$ can compute $x$
One-way functions (OWFs)

A OWF is a function that is easy to compute but hard to invert:

**Definition (One-way)**

A function $f$ is a one-way function if for all $x$ there is no efficient algorithm which given $f(x)$ can compute $x$.

Constant functions ARE OWF:

for any function $f(x) = c$ ($c$ a constant) it is impossible to retrieve $n$ from $f(n)$.
One-way functions (OWFs)

A OWF is a function that is easy to compute but hard to invert:

Definition (One-way)
A function $f$ is a one-way function if for all $x$ there is no efficient algorithm which given $f(x)$ can compute $x$

Constant functions ARE OWF:
for any function $f(x) = c$ ($c$ a constant) it is impossible to retrieve $n$ from $f(n)$

The successor function in $\mathbb{N}$ IS NOT a OWF
given $\text{succ}(n)$ it is easy to retrieve $n = \text{succ}(n) - 1$
One-way functions (OWFs)

A OWF is a function that is easy to compute but hard to invert:

**Definition (One-way)**

A function $f$ is a one-way function if for all $x$ there is no efficient algorithm which given $f(x)$ can compute $x$

Constant functions ARE OWF:

for any function $f(x) = c$ ($c$ a constant) it is impossible to retrieve $n$ from $f(n)$

The successor function in $\mathbb{N}$ IS NOT a OWF

given $\text{succ}(n)$ it is easy to retrieve $n = \text{succ}(n) - 1$

Multiplication of large primes IS a OWF:

integer factorization is a hard problem - given $p \times q$ (where $p$ and $q$ are primes) it is hard to retrieve $p$ and $q$
Collision-resistant functions (CRFs)

A function is a CRF if it is hard to find two messages that get mapped to the same value threw this function.

Definition (Collision resistance)

A function $f$ is collision resistant if there is no efficient algorithm that can find two messages $m_1$ and $m_2$ such that $f(m_1) = f(m_2)$.

Constant functions ARE NOT CRFs.

The successor function in $\mathbb{N}$ IS a CRF: the predecessor of a positive integer is unique.

Multiplication of large primes IS a CRF: every positive integer has a unique prime factorization.
Collision-resistant functions (CRFs)

A function is a CRF if it is hard to find two messages that get mapped to the same value threw this function

**Definition (Collision resistance)**

A function $f$ is collision resistant if there is no efficient algorithm that can find two messages $m_1$ and $m_2$ such that $f(m_1) = f(m_2)$

Constant functions ARE NOT CRFs
for all $m_1$ and $m_2$, $f(m_1) = f(m_2)$
Collision-resistant functions (CRFs)

A function is a CRF if it is hard to find two messages that get mapped to the same value though this function

**Definition (Collision resistance)**

A function $f$ is collision resistant if there is no efficient algorithm that can find two messages $m_1$ and $m_2$ such that $f(m_1) = f(m_2)$

Constant functions ARE NOT CRFs

for all $m_1$ and $m_2$, $f(m_1) = f(m_2)$

The successor function in $\mathbb{N}$ IS a CRF

the predecessor of a positive integer is unique
Collision-resistant functions (CRFs)

A function is a CRF if it is hard to find two messages that get mapped to the same value threw this function

Definition (Collision resistance)
A function $f$ is collision resistant if there is no efficient algorithm that can find two messages $m_1$ and $m_2$ such that $f(m_1) = f(m_2)$

Constant functions ARE NOT CRFs
for all $m_1$ and $m_2$, $f(m_1) = f(m_2)$

The successor function in $\mathbb{N}$ IS a CRF
the predecessor of a positive integer is unique

Multiplication of large primes IS a CRF:
every positive integer has a unique prime factorization
Cryptographic hash functions

A cryptographic hash function takes messages of arbitrary length end returns a fixed-size bit string such that any change to the data will (with very high probability) change the corresponding hash value.

Definition (Cryptographic hash function)

A cryptographic hash function $H : \mathcal{M} \rightarrow \mathcal{T}$ is a function that satisfies the following 4 properties:

- $|\mathcal{M}| >> |\mathcal{T}|$
- it is easy to compute the hash value for any given message
- it is hard to retrieve a message from its hashed value (OWF)
- it is hard to find two different messages with the same hash value (CRF)

Examples: MD4, MD5, SHA-1, SHA-256, Whirlpool, ...
Cryptographic hash functions: applications

- **Commitments** - Allow a participant to commit to a value $v$ by publishing the hash $H(v)$ of this value, but revealing $v$ only later. Ex: electronic voting protocols, digital signatures, . . .

- **File integrity** - Hashes are sometimes posted along with files on "read-only" spaces to allow verification of integrity of the files. Ex: SHA-256 is used to authenticate Debian GNU/Linux software packages.

- **Password verification** - Instead of storing passwords in cleartext, only the hash digest of each password is stored. To authenticate a user, the password presented by the user is hashed and compared with the stored hash.

- **Key derivation** - Derive new keys or passwords from a single, secure key or password.

- **Building block of other crypto primitives** - Used to build MACs, block ciphers, PRG, . . .
Cryptographic hash functions: applications

- **Commitments** - Allow a participant to commit to a value \( v \) by publishing the hash \( H(v) \) of this value, but revealing \( v \) only later. Ex: electronic voting protocols, digital signatures, . . .

- **File integrity** - Hashes are sometimes posted along with files on “read-only” spaces to allow verification of integrity of the files. Ex: SHA-256 is used to authenticate Debian GNU/Linux software packages
Cryptographic hash functions: applications

- **Commitments** - Allow a participant to commit to a value $v$ by publishing the hash $H(v)$ of this value, but revealing $v$ only later. Ex: electronic voting protocols, digital signatures, . . .

- **File integrity** - Hashes are sometimes posted along with files on “read-only” spaces to allow verification of integrity of the files. Ex: SHA-256 is used to authenticate Debian GNU/Linux software packages

- **Password verification** - Instead of storing passwords in cleartext, only the hash digest of each password is stored. To authenticate a user, the password presented by the user is hashed and compared with the stored hash.
Cryptographic hash functions: applications

- **Commitments** - Allow a participant to commit to a value $v$ by publishing the hash $H(v)$ of this value, but revealing $v$ only later. Ex: electronic voting protocols, digital signatures, ...  

- **File integrity** - Hashes are sometimes posted along with files on “read-only” spaces to allow verification of integrity of the files. Ex: SHA-256 is used to authenticate Debian GNU/Linux software packages

- **Password verification** - Instead of storing passwords in cleartext, only the hash digest of each password is stored. To authenticate a user, the password presented by the user is hashed and compared with the stored hash.

- **Key derivation** - Derive new keys or passwords from a single, secure key or password.
Cryptographic hash functions: applications

- **Commitments** - Allow a participant to commit to a value $v$ by publishing the hash $H(v)$ of this value, but revealing $v$ only later. Ex: electronic voting protocols, digital signatures, . . .

- **File integrity** - Hashes are sometimes posted along with files on “read-only” spaces to allow verification of integrity of the files. Ex: SHA-256 is used to authenticate Debian GNU/Linux software packages

- **Password verification** - Instead of storing passwords in cleartext, only the hash digest of each password is stored. To authenticate a user, the password presented by the user is hashed and compared with the stored hash.

- **Key derivation** - Derive new keys or passwords from a single, secure key or password.

- **Building block of other crypto primitives** - Used to build MACs, block ciphers, PRG, . . .
Collision resistance and the birthday attack

**Theorem**

Let $H : \mathcal{M} \rightarrow \{0, 1\}^n$ be a cryptographic hash function ($|\mathcal{M}| \gg 2^n$)

Generic algorithm to find a collision in time $O(2^{n/2})$ hashes:

1. Choose $2^{n/2}$ random messages in $\mathcal{M}$: $m_1, \ldots, m_{2^{n/2}}$
2. For $i = 1, \ldots, 2^{n/2}$ compute $t_i = H(m_i)$
3. If there exists a collision ($\exists i, j. \ t_i \neq t_j$) then return $(t_i, t_j)$ else go back to 1

**Birthday paradox** Let $r_1, \ldots, r_n \in \{1, \ldots, N\}$ be independent variables. For $n = 1.2 \times \sqrt{N}$, $Pr(\exists i \neq j. \ r_i = r_j) \geq \frac{1}{2}$

$\Rightarrow$ the expected number of iteration is 2
$\Rightarrow$ running time $O(2^{n/2})$

$\Rightarrow$ Cryptographic function used in new projects should have an output size $n \geq 256!$
The Merkle-Damgard construction

![Diagram showing the Merkle-Damgard construction]

- Compression function: $h : \mathcal{T} \times \mathcal{X} \to \mathcal{T}$
- PB: 1000...0∥mes-len (add extra block if needed)

**Theorem**

*Let $H$ be built using the MD construction to the compression function $h$. If $H$ admits a collision, so does $h.*

**Example of MD constructions**: MD5, SHA-1, SHA-2, ...
Compression functions from block ciphers

Let $E: \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ be a block cipher
Let $E : \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ be a block cipher
Let $E : \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ be a block cipher.
Example of cryptographic hash function: SHA-256

- Structure: Merkle-Damgard
- Compression function: Davies-Meyer
- Block cipher: SHACAL-2
Message Authentication Codes (MACs)

Myrto Arapinis
School of Informatics
University of Edinburgh

October 11, 2016
Goal: message integrity

Generate tag:
$t ← \text{MDC}(m)$

Verify tag:
$V(m, t) = \text{“yes”}$?
Goal: message integrity

Alice

Generate tag

\[ t \leftarrow S(k, m) \]

Verify tag

\[ V(k, m, t) = \text{“yes”} \]?

Bob

A MAC is a pair of algorithms \((S, V)\) defined over \((\mathcal{K}, \mathcal{M}, \mathcal{T})\):

\begin{itemize}
  \item \( S : \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{T} \)
  \item \( V : \mathcal{K} \times \mathcal{M} \times \mathcal{T} \rightarrow \{\top, \bot\} \)
  \item Consistency: \( V(k, m, S(k, m)) = T \)
\end{itemize}

and such that

\begin{itemize}
  \item It is hard to compute a valid pair \((m, S(k, m))\) without knowing \(k\)
At installation time

$t_1 = S(k, F_1)$
$t_2 = S(k, F_2)$
$t_n = S(k, F_n)$

$k$ derived from user password

To check for virus file tampering/alteration:

- reboot to clean OS
- supply password
- any file modification will be detected
Block ciphers and message integrity

Let $(E, D)$ be a block cipher. We build a MAC $(S, V)$ using $(E, D)$ as follows:

$S(k, m) = E(k, m)$

$V(k, m, t) = \text{if } m = D(k, t) \text{ then return } \top \text{ else return } \bot$

But: block ciphers can usually process only 128 or 256 bits.
Our goal now: construct MACs for long messages.
Let \((E, D)\) be a block cipher. We build a MAC \((S, V)\) using \((E, D)\) as follows:

- \(S(k, m) = E(k, m)\)
- \(V(k, m, t) = \begin{cases} \top & \text{if } m = D(k, t) \\ \bot & \text{otherwise} \end{cases}\)
Let \((E, D)\) be a block cipher. We build a MAC \((S, V)\) using \((E, D)\) as follows:

\[
\begin{align*}
S(k, m) &= E(k, m) \\
V(k, m, t) &= \text{if } m = D(k, t) \\
&\quad \text{then return } \top \\
&\quad \text{else return } \bot
\end{align*}
\]

**But:** block ciphers can usually process only 128 or 256 bits
Block ciphers and message integrity

Let \((E, D)\) be a block cipher. We build a MAC \((S, V)\) using \((E, D)\) as follows:

\[
\begin{align*}
S(k, m) &= E(k, m) \\
V(k, m, t) &= \text{if } m = D(k, t) \\
&\quad \text{then return } \top \\
&\quad \text{else return } \bot
\end{align*}
\]

**But:** block ciphers can usually process only 128 or 256 bits

**Our goal now:** construct MACs for long messages
ECBC-MAC

$E(K_1, \cdot) + E(K_1, \cdot) + E(K_1, \cdot) + E(K_1, \cdot) + E(K_2, \cdot) \rightarrow t$

$E : \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ a block cipher

$ECBC-MAC : \mathcal{K}^2 \times \{0, 1\}^* \rightarrow \{0, 1\}^n$

→ the last encryption is crucial to avoid forgeries!!

Ex: 802.11i uses AES based ECBC-MAC

(details on the board)
PMAC

$E(K_1, \cdot) + E(K_1, \cdot) + E(K_1, \cdot) + E(K_1, \cdot) \rightarrow t$

- $E: \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ a block cipher
- $P: \mathcal{K} \times \mathbb{N} \rightarrow \{0, 1\}^n$ any easy to compute function
- $PMAC: \mathcal{K}^2 \times \{0, 1\}^* \rightarrow \{0, 1\}^n$
**HMAC**

MAC built from cryptographic hash functions

\[ HMAC(k, m) = H(k \oplus OP || H(k \oplus IP || m)) \]

*IP, OP*: publicly known padding constants

Ex: SSL, IPsec, SSH, ...
Authenticated encryption

Myrto Arapinis
School of Informatics
University of Edinburgh

October 11, 2016
Plain encryption is malleable

Goal
Simultaneously provide data confidentiality, integrity and authenticity
⇝ decryption combined with integrity verification in one step

- The decryption algorithm never fails
- Changing one bit of the $i^{th}$ block of the ciphertext
  - CBC decryption: will affect last blocks after the $i^{th}$ of the plaintext
  - ECB decryption: will only the $i^{th}$ block of the plaintext
  - CTR decryption: will only affect one bit of the $i^{th}$ block of the plaintext

Decryption should fail if a ciphertext was not computed using the key
Encrypt-then-MAC

1. Always compute the MACs on the ciphertext, never on the plaintext
2. Use two different keys, one for encryption and one for the MAC

Encryption

1. \( C \leftarrow E_{AES}(K_1, M) \)
2. \( T \leftarrow HMAC-SHA(K_2, C) \)
3. return \( C \| T \)

Decryption

1. if \( T = HMAC - SHA(K_2, C) \)
2. then return \( D_{AES}(K_1, C) \)
3. else return \( \perp \)

Do not:

- Encrypt-then-MAC
- Encrypt-and-MAC