CPSC 418/MATH 318 Introduction to Cryptography

Hash Functions and Message Authentication Codes

Renate Scheidler

Department of Mathematics & Statistics
Department of Computer Science
University of Calgary

Week 7

Question:	What do security analysts call a set of identical twins?
Answer:	A hash collision.

Outline

Hash Functions

Often referred to as the "work horse" of cryptography — they are ubiquitous in crypto.

Definition 1 (Hash function)

A function $H : \{0,1\}^* \rightarrow \{0,1\}^m$ ($m \in \mathbb{N}$) that is easy to compute. An image $x = H(M)$ is referred to as a *message digest* or a *digital fingerprint* or a *checksum* or simply a *hash*.

Hash functions thus satisfy two properties:

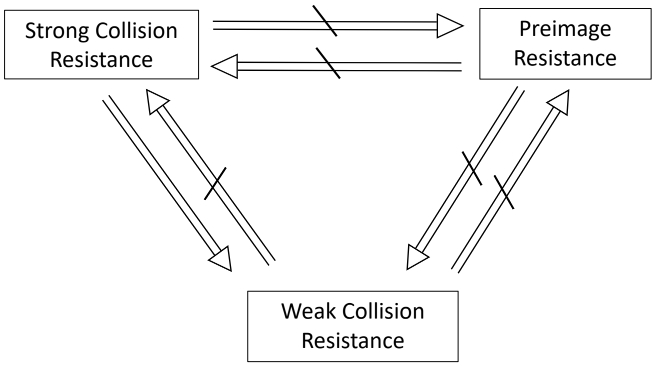
- *Compression*: H maps an input M of arbitrary bit length to an output of fixed bit length.
- *Ease of computation*: for any input M, $H(M)$ is easy to compute.

Cryptographic Requirements

Desirable properties for hash functions in the context of cryptography:

- *Pre-image resistance*: given any hash value x, it is computationally infeasible to find a *pre-image* of x, i.e. *any* input M for which $H(M) = x$.
- *Collision resistance* or *strong collision resistance*: it is computationally infeasible to find a *strong collision*, i.e. two distinct inputs M and M' such that $H(M) = H(M')$.
- *Second pre-image resistance* or *weak collision resistance*: given any M, it is computationally infeasible to find a *weak collision*, i.e. an input $M' \neq M$ with $H(M) = H(M')$.

Relationships



Strong collision resistance implies weak collision resistance because every weak collision is also a strong collision.

Uses of Cryptographically Secure Hash Functions

Definition 2

A hash function is *cryptographic(ally secure)* if it is pre-image resistant and collision resistant.

Some example applications:

- In digital signatures (including the post-quantum scheme SPHINCS+) to prevent impersonation (sign $H(M)$ instead of M — later)
- Data integrity without secrecy (*e.g.* downloading large files, compare checksum before and after download)
- Data integrity with secrecy (see below)
- Key derivation (*e.g.* use hash of a Diffie-Hellman secret as your key)
- Commitment (can verify $H(M)$ to see if M was committed to)
- Randomness (*e.g.* `dev/random`, one-time passwords, OAEP — later)

Application: Data Integrity with Secrecy

Using hashing plus encryption:

- Sender sends $C = E_K(M \| x)$ with $x = H(M)$
- Receiver decrypts C to obtain M', x' and checks that $H(M') = x'$.

Idea:

- Adversary cannot manipulate ciphertext blocks in such a way that $H(M') = x'$.
- May be possible if H is not cryptographically secure (eg. WEP: combination of stream cipher and checksum).

Iterated Hash Function Design

Iterated hash functions are composed of rounds (like DES or AES)

- Repeated use of *compression function* f — takes m-bit input from the previous step (chaining variable) and an r-bit block from M; produces m-bit output.
- Input to H : message M consisting of r-bit blocks $P_1, \ldots, P_L$ (padded, if necessary, so the total length is a multiple of r).

$$\begin{aligned} H_0 &= IV \quad \text{(initial m-bit value, e.g. all zeros)} \\ H_i &= f(H_{i-1}, P_i), \quad 1 \le i \le L \\ H(M) &= H_L \end{aligned}$$

Iterated hash functions can be set up in such a way so that if f is collision-resistant, so is H (Merkle 1989 and Damgard 1989).

SHA-1

Secure Hash Algorithm 1 (SHA-1)

Developed by NIST in 1993 (FIPS 180 and FIPS 180-1).

- Iterated round hash function with hash length 160 bits.
- Can now find SHA-1 collisions in 2^{57} attempts.
- Longer versions (up to 512 bits) still certified for use under SHA-2 — more on that later.

SHA-1: Overview

Messages (padded suitably) are processed in 512-bit blocks, divided into 16 words of bit length 32 each.

Hash function operates on 160-bit *buffers*, divided into 5 words of bit length 32 each:

- Current message block is processed with current buffer via four rounds of 20 steps each.
- Next buffer is produced by adding wordwise (modulo 2^{32}) the current buffer to the output of the fourth round.
- Hash value is the final buffer value.

For details, consult the SHA-1 handout on the "handouts" page.

Attacks on SHA-1

Finding collisions:

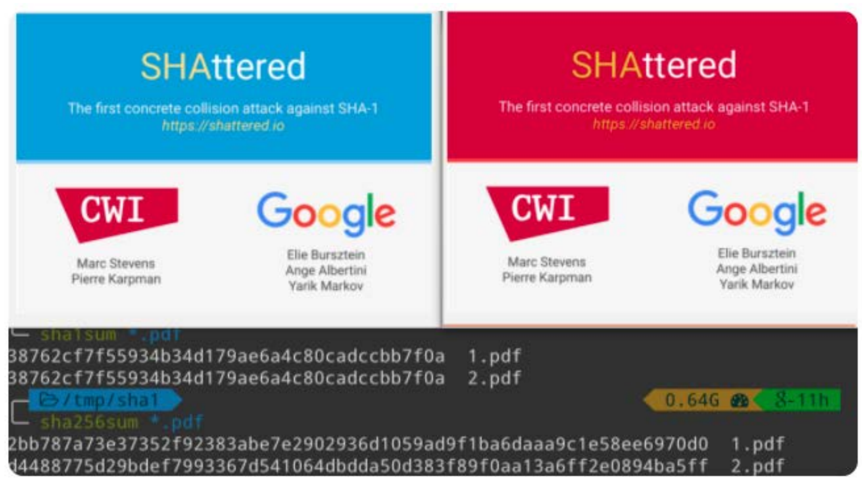
- Wang, Yin, Yu (Feb. 2005) — 2^{69} hash ops
- Wang, Yao, Yao (Aug. 2005) — 2^{63} hash ops
- Stevens (2013) — 2^{60} hash ops
- Stevens, Karpman, Peyrin (2015) — $2^{57.5}$ hash ops
- Practical implementations in 2017 (CWI Amsterdam-Google team including Stevens & Karpman, `https://SHAttered.io/`) and 2020 (Leuren-Peyrin, `https://SHA-mbles.github.io`)

Significantly less than theoretical maximum (2^{80}) — therefore, considered vulnerable.

Replaced by SHA-2 and SHA-3 in August 2015. See the hash function page at `https://csrc.nist.gov/projects/hash-functions` under NIST's Cryptographic Standards and Guidelines website for more.

The SHAttered Attack

Two different files with the same SHA-1 tag:



(Taken from `https://SHAttered.io/`)

Some Other Hash Functions

MD5 — 128-bit hash length, developed by Rivest.

- Essentially broken (Wang et. al., 2004). MD5 collision found on a laptop in 8 hours (Klima, 2005).

Revised hash standard SHA-2 consisting of SHA-224 , SHA-256, SHA-384, SHA-512, SHA-512/224 and SHA-512/256 (see FIPS 180-4):

- modifications of SHA-1 to provide 112, 128, 192, and 256 bits of security for compatibility with 3DES and AES.
- current recommendation: if unable to convert to SHA-3, use one of these in place of SHA-1.

Charles, Goren, Lauter (2009) — hash function based on expander graphs

- provable security: finding collisions reduces to computing isogenies between supersingular elliptic curves

SHA-3

After the 2005 attack on SHA-1, NIST initiated a competition for new hash algorithms, similar to the AES competition. It ran 2007-2012 and a SHA-3 standard was adopted on August 5, 2015.

SHA-3 winner: Keccak (pronounced "ketchuk"), invented by

- Guido Bertoni (Italy) of STMicroelectronics,
- Joan Daemen (Belgium) of STMicroelectronics (one of the AES/Rijndahl creators!),
- Michaël Peeters (Belgium) of NXP Semiconductors,
- Gilles Van Assche (Belgium) of STMicroelectronics.

Resources:

- NIST FIPS 202
- `http://keccak.noekeon.org/Keccak-reference-3.0.pdf`
- KECCAK presentation given to NIST by the Keccak inventors on Feb. 6, 2013 (on "handouts" page)

Sponge Construction

Keccak is based on a *sponge* design; see `https://keccak.team/sponge_duplex.html`.

- Hash function: arbitrary input length, fixed output length
- Stream cipher: fixed input length, arbitrary output length
- Sponge function: arbitrary input length, variable user-supplied output length

Sponges can be used to build various cryptographic primitives (stream ciphers, hash functions, message authentication codes)

Sponges – Overview

Ingredients of a sponge function:

- A *width* b (an integer)
- A *bit rate* r (an integer $< b$)
- An input S (a bit string of length b)
- A fixed-length permutation f that operates on S
- A padding rule "*pad*" that pads blocks of length r to blocks of length b.

The *capacity* of the sponge is the padding amount $c = b - r$.

The padding rule for Keccak simply appends the string $1\underbrace{00\cdots0}_{c\text{-2 zeros}}1$ to each r-bit block (called *multi-rate padding*).

Sponge Function – Absorb

The input to the *absorption* phase is the message M — padded so the total length is a multiple of r — consisting of r-bit blocks $P_1, \ldots, P_L$.

The output is a string S of length b.

Absorption Phase — "x-or & permute"

$S \leftarrow 0^b$ (b zeros)
For $i = 1$ to L do
  $S \leftarrow S \oplus pad(P_i)$
  $S \leftarrow f(S)$
end for

Sponge Function – Squeeze

The *squeezing phase* outputs on input S a hash of the message M whose bit length is a user-supplied value m.

Squeezing Phase — "permute & append"

$Z \leftarrow$ first r bits of S
While $length(Z) < m$ do
  $S \leftarrow f(S)$
  append the first r bits of S to Z
end while
$H(M) \leftarrow$ first m bits of Z

SHA-3 Specification

SHA-3/Keccak specifies

- hash lengths $m = 224, 256, 384, 512$ (just like SHA-2)
- capacities $c = 2m$
- widths $b = 25$, 50, 100, 200, 400, 800, 1600 (default is 1600)

The internal state to the Keccak permutation f , denoted A, is a 3-dimensional bit-array of dimensions $5 \times 5 \times 2^\ell$ where $0 \le \ell \le 6$, yielding the above widths (default is $\ell = 6$, with a state of dimensions $5 \times 5 \times 64$).

The Keccak permutation f iterates over multiple rounds. In SHA-3, the number of rounds N_r is $12 + 2\ell$. (E.g. $N_r = 24$ for for $b = 1600$.) Each round of f operates on the state A and is the composition of 5 functions:

$$\iota \circ \chi \circ \pi \circ \rho \circ \theta$$

where θ, ρ, π and χ are identical for each round, and ι incorporates *round constants* that vary by round.

The Keccak Permutation f

Input: bit string S of length b

Output: bit string S of length b

1. Convert S to a $5 \times 5 \times 2^\ell$ state A (where $b = 5 \cdot 5 \cdot 2^\ell$)
2. For $i = 0$ to $N_r - 1$ do
$$A \leftarrow \iota(\chi(\pi(\rho(\theta(A)))), i)$$
3. Convert A to a string S of length b
4. Output S

The mathematical description of each of the 5 maps θ, ρ, π, χ and ι can be found on page 8 of *Keccak-reference-3.0.pdf*. They can all be implemented using only bitwise XOR, AND, NOT, but no table look-ups, arithmetic or data-dependent rotations (very fast).

Geography of Keccak States

State entries are denoted $A[x, y, z]$ where

$$0 \le x \le 4 \ , \quad 0 \le y \le 4 \ , \quad 0 \le z \le 2^\ell - 1 \ .$$

E.g. for $b = 1600$ ($\ell = 6$), we have $0 \le x \le 4$, $0 \le y \le 4$, $0 \le z \le 63$.

Navigating States:

Rows: $A[0, y, z]$ $A[1, y, z]$ $A[2, y, z]$ $A[3, y, z]$ $A[4, y, z]$
Columns: $A[x, 0, z]$ $A[x, 1, z]$ $A[x, 2, z]$ $A[x, 3, z]$ $A[x, 4, z]$
Lanes: $A[x, y, 0]$ $A[x, y, 1]$ $A[x, y, 2]$ $\cdots$ $A[x, y, 2^\ell - 1]$

Converting Bit Strings to States

Suppose the input string consists of bits

$$s_0, s_1, \ldots, s_{b-1} \ .$$

Then

$$A[x, y, z] = s_{2^\ell(5y+x)+z} \ .$$

So A is populated lane-wise, "floor" by "floor":

- starting with the bottom row of lanes (ground floor)
- followed by the row of lanes second from the bottom (second floor)
- followed by the middle, then the second from the top, then the top row of lanes

Converting Bit Strings to States (cont'd)

We assign the bits s_i ($0 \le i \le b - 1$) to A in the following order:

$y = 0$	$x = 0$ $x = 1$ $\vdots$ $x = 4$	$z = 0, 1, \ldots 2^\ell - 1$ $z = 0, 1, \ldots 2^\ell - 1$ $\vdots$ $z = 0, 1, \ldots 2^\ell - 1$
$y = 1$	$x = 0$ $x = 1$ $\vdots$ $x = 4$	$z = 0, 1, \ldots 2^\ell - 1$ $z = 0, 1, \ldots 2^\ell - 1$ $\vdots$ $z = 0, 1, \ldots 2^\ell - 1$
$\vdots$	$\vdots$	$\vdots$
$y = 4$	$x = 0$ $\vdots$ $x = 4$	$z = 0, 1, \ldots 2^\ell - 1$ $\vdots$ $z = 0, 1, \ldots 2^\ell - 1$

Converting States to Bit Strings

Conversion from the final state A to the bit string S is done in by reversing this process (order lane–row–column):

$$\begin{aligned}
S = {} & A[0,0,0] \ A[0,0,1] \ \ldots \ A[0,0,2^\ell-1] \\
& A[1,0,0] \ A[1,0,1] \ \ldots \ A[1,0,2^\ell-1] \\
& A[2,0,0] \ A[2,0,1] \ \ldots \ A[2,0,2^\ell-1] \\
& A[3,0,0] \ A[3,0,1] \ \ldots \ A[3,0,2^\ell-1] \\
& A[4,0,0] \ A[4,0,1] \ \ldots \ A[4,0,2^\ell-1] \\
& A[0,1,0] \ A[0,1,1] \ \ldots \ A[0,1,2^\ell-1] \\
& \qquad \cdots \\
& A[4,1,0] \ A[4,1,1] \ \ldots \ A[4,1,2^\ell-1] \\
& \qquad \cdots \\
& A[0,4,0] \ A[0,4,1] \ \ldots \ A[0,4,2^\ell-1] \\
& \qquad \cdots \\
& A[4,4,0] \ A[4,4,1] \ \ldots \ A[4,4,2^\ell-1]
\end{aligned}$$

The Map θ

θ adds to each bit $A[x, y, z]$ the bitwise x-or of the parities of the two columns $A[x-1, *, z]$ and $A[x+1, *, z-1]$, where the x-index is taken modulo 5 and the z-index modulo 2^ℓ.

1. For all pairs (x, z) with $0 \le x \le 4$ and $0 \le z \le 2^{\ell-1}$ do
   // x-or all columns $A[x, *, z]$ to compute parities
   $C[x, z] \leftarrow A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z]$
2. For all pairs (x, z) with $0 \le x \le 4$ and $0 \le z \le 2^{\ell-1}$ do
   $D[x, z] \leftarrow C[(x-1) \bmod 5, z] \oplus C[(x+1) \bmod 5, (z-1) \bmod 2^\ell]$
3. For all triples (x, y, z) with $0 \le x \le 4$, $0 \le y \le 4$, $0 \le z \le 2^{\ell-1}$ do
   $A[x, y, z] \leftarrow A[x, y, z] \oplus D[x, z]$

θ provides a high level of diffusion.

The Map ρ

ρ rotates the bits of each lane by adding to the z-coordinate an *offset* modulo 2^ℓ (circular shift along the lane) as given in the following table:

	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	153	231	3	10	171
$y=1$	55	276	36	300	6
$y=0$	28	91	0	1	190
$y=4$	120	78	210	66	253
$y=3$	21	136	105	45	15

Consult pages 12-13 of FIPS 202 or page 8 of *Keccak-reference-3.0.pdf* to see how these offsets are calculated.

ρ disperses *slices* $A[x, y, *]$ for more diffusion.

The Map π

π rearranges all the lanes, moving lane

$$A[x, y, *]$$

to lane

$$A[(x+3y) \bmod 5, x, *] \ .$$

This lane dispersion provides yet more diffusion.

The Map χ

χ x-or's each bit $A[x, y, z]$ with the non-linear function of two bits in the same row given by

$$\overline{A}[(x+1) \bmod 5, y, z] \ \wedge \ A[(x+2) \bmod 5, y, z]$$

where $\overline{A}$ denotes the bit complement of A and $\wedge$ denotes logical "and" (multiplication modulo 2).

χ is the only non-linear map within Keccak.

The Map ι

ι x-or's the ℓ bits $A[0,0,2^j-1]$ $(0 \le j \le \ell)$ with *round constants* $rc(j+7i)$ where i is the round number.

Here, $rc[t]$ is the constant coefficient of x^t modulo $x^8+x^6+x^5+x^4+1$ which can be obtained via some simple bit x-ors and truncations as the output of a *linear feedback shift register* (LSFR) (see page 16 of FIPS 202).

ι disrupts symmetry.

ι acts only on a few bits in lane $A[0,0,*]$, but the lane rearrangement π and the slice dispersion ρ ensure that this action affects every lane of A.

Concluding Remarks on SHA-3 and Keccak

Keccak is secure against all known attacks.

In addition to the four hash functions SHA3-m that produce hashes of lengths $m = 224$, 256, 384, 512 using capacities $c = 2m$, the SHA-3 standard supports two other Keccak-based *extendable output* functions SHAKE128 and SHAKE256 (supporting variable length outputs) that produce hashes of the same four lengths m using respective fixed capacities 256 and 512. (Not approved yet, guidelines for use promised in the future.)

Four other SHA-3 derived functions (called cSHAKE, KMAC, TupleHash and ParallelHash) are described in NIST SP 800-185.

See `https://csrc.nist.gov/projects/hash-functions` for details.

Attacks on Hash Functions

Objectives of adversaries attacking hash functions:

- Find a pre-image: given any hash, create a corresponding message with that hash.
- Find a weak collision: given a message, modify it to another message with the same hash.
- Find a collision: find two messages with the same hash.

Brute-force Attacks

Like block ciphers, brute force should be the best attack.

For an m-bit hash function:

- Pre-images and weak collisions: 2^m attempts on average ($\approx 0.69 \cdot 2^m$ attempts for a 50% chance of finding a weak collision; see Problem 3 on Assignment 1)
- Strong collisions: $2^{m/2}$ attempts on average due to the *birthday paradox*: expect that $\approx 1.177 \cdot \sqrt{2^m}$ trials yield a 50% chance of finding a collision (see Problem 4 on Assignment 1 or page 145 of Paterson-Stinson)

Recommended sizes: $m = 224$, 256, 394, 512 (provide 112, 128, 192, and 256 bits of security)

Weak Versus Strong Collision Resistance

Recall that every strongly collision resistant hash function is also weakly collision resistant (because every weak collision is also a strong collision).

What about a weakly collision resistant hash function that is *not* strongly collision resistant?

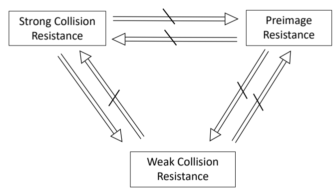
Let m be of a size where

- searching a space of size 2^m is computationally infeasible,
- searching a space of size $2^{m/2}$ is computationally feasible.

(E.g. $m = 112$, like in 3DES versus DES.)

Then we expect an m-bit hash function to be

- pre-image resistant
- weakly collision resistant
- *not* strongly collision resistant



Birthday Attack on Digital Signatures

Birthday attack on signature schemes with hash functions (more later):

- Attacker generates $2^{m/2}$ variations of a valid message (easy to do by adding/removing white space, replacing synonyms, etc...).
- Attacker generates $2^{m/2}$ variations of a desired fraudulent message.
- The two sets of messages are compared to find a pair with the same hash.
- Attacker has the victim sign the hash of the valid message — the signature will also be valid for the fraudulent message.

Cryptanalytic Attacks

Recall that iterated hash functions H (like SHA-1 and MD5) are composed of rounds that iteratively use a *compression function* f.

Iterated hash functions can be set up in such a way so that if f is collision-resistant, so is H (Merkle 1989 and Damgard 1989).

An attack approach is to exploit the structure of the hash function (similar to block ciphers):

- Analytically attack the rounds of a hash function
- Focus on collisions in function f.
- Many hash functions have succumbed to this type of attack (due to Wang et al).

Message Authentication Codes

A small, fixed-size, key-dependent block that is appended to a message to check data integrity — AKA *keyed hash function* or *tag*.

Definition 3 (Message authentication code (MAC))

A single-parameter family $\{MAC_K\}_{K \in \mathcal{K}}$ of many-to-one functions $MAC_K : \mathcal{M} \to \{0,1\}^n$ ($n \in \mathbb{N}$) satisfying:

- *Ease of computation with knowledge of K:* For any $M \in \mathcal{M}$ and $K \in \mathcal{K}$, $MAC_K(M)$ is easy to compute.
- *Computation resistance*: for any $K \in \mathcal{K}$, given zero or more message/MAC pairs $(M_i, MAC_K(M_i))$, it is computationally infeasible to compute any new message/MAC pair $(M, MAC_K(M))$, $M \neq M_i$ for all i, without knowledge of K.

More on Computation Resistance

Common error: confusing computation resistance with collision resistance or pre-image resistance!

The two are very different:

- Unlike collision resistance, computation resistance does **not** ask for two different messages with the same MAC.
- Unlike pre-image resistance, computation resistance does **not** ask for a pre-image of a given MAC tag.
- Rather, computation resistance asks for a valid message/MAC pair where the message is **new**.
- The idea is that despite observing a collection of message/MAC pairs $(M_i, MAC_K(M_i))$ exchanged between Alice and Bob, Eve cannot authenticate a message that was not already sent.
  In practice, Alice and Bob should time-stamp their messages to avoid *replays* by Eve.

Data Integrity using MACs

Computation resistance implies data integrity (without secrecy):

- Sender and receiver share a secret key K.
- Sender computes $T = MAC_K(M)$ and sends (M, T).
- Upon receiving a pair (M', T') from the sender, the receiver checks whether $T' = MAC_K(M')$. If this computation checks out and MAC_K is computation resistant, the integrity of M is preserved, i.e. $M' = M$ (as (M', T') would be a new message/MAC pair otherwise).

Active attack:

- Attacker suppresses (M, T) and instead sends a pair (M'', T'') with $M'' \neq M$ to the receiver.
- Receiver checks if $MAC_K(M'') = T''$. If this holds, the attacker must have defeated computation resistance by generating a new pair (M'', T'') from (M, T).

Sender Authentication using MACs

MACs also provide limited sender authentication in a similar manner to encryption

- only sender or receiver (who both know K) can generate the MAC.

Note: Non-repudiation of data origin *not* provided

- *either* party possessing K can generate the MAC.

In practice, digital signatures should be used, which provide both sender authentication and non-repudiation (more later).

MAC Versus Encryption

Differences between encryption and MACs:

Encryption	*MAC*
Secrecy	No secrecy
Reversible via decryption	Need not be reversible
Injective	Many messages with the same MAC

Note 1

Just like in encryption, MAC should depend equally on all bits of the message. Given a valid message/MAC pair, it should still be hard to find another valid pair even if only one bit of the message is modified.

MACs from Block Ciphers

A secure block cipher (satisfying additional statistical properties) can be used to generate MACs. Two methods are:

1. CBC-MAC:
   - Encrypt the message (IV of all zeros, last block padded with 0s) using CBC mode.
   - The last cipher block (whose bits are dependent on all the key bits and all message bits) is the MAC.
2. CFB-MAC: Same idea as CBC-MAC

A CBC-MAC using DES appears in both FIPS 113 and the ANSI X9.17 standard.

Problem with CBC-MAC

Problem: only secure if messages of *one* fixed length are processed (Bellare, Killian, Rogaway 2000).

Solution (CMAC):

- Use *three* keys, one at each step of the chaining, two for the last block (Black-Rogaway 2000).
- Second two keys may be derived from the encryption key (Iwata, Kurosawa 2003).

Properties of CMAC

Cipher-based Message Authentication Code (CMAC)

- Specified for use with AES and 3DES in NIST Special Pub. 800-38B
- Can be proved secure as long as the underlying block cipher's output is indistinguishable from a random permutation.
- No known weaknesses.

Operation of CMAC

Message M is padded so its length is a multiple of the cipher's block length n (128 for AES, 64 for 3DES) by appending a 1 and as many 0s as necessary, then divided into blocks $M_1, \ldots, M_m$.

Let K be the block cipher key. Two additional keys K_1 and K_2 are computed as follows:

$$\begin{aligned} L &= E_K(0^n) \\ K_1 &= L \cdot x \\ K_2 &= L \cdot x^2 = K_1 \cdot x \end{aligned}$$

where $\cdot$ denotes multiplication of polynomials with bit coefficients modulo $x^{64} + x^4 + x^3 + x + 1$ or $x^{128} + x^7 + x^2 + x + 1$ (i.e., mult. in $GF(2^n)$).

Operation of CMAC, cont.

To compute the MAC of message M, process blocks $M_1, \ldots, M_{m-1}$ using CBC with $IV = 0$:

$$C_0 = 0^n$$
$$C_i = E_K(M_i \oplus C_{i-1}) \quad 1 \leq i \leq m-1 \ .$$

Compute

$$C_m = E_K(M_m \oplus C_{m-1} \oplus K_i)$$

where $i = 1$ if M was not padded and $i = 2$ if M was padded.

MAC is the s leftmost (most significant) bits of C_m (where s is determined by the desired level of security).

Identical to CBC-MAC except for the last round.