T-110.5211 Cryptosystems

Symmetric algorithms

2.10.2008

Kaufman et al: Chapters 3.3 / 3.5 / 4.4 / 5.5-5.7

Stallings: Chapters 3.3-3.4 / 5.1-5.2 / 6.1 / 6.5 / 12.1-12.2 / 12.4

Outline

- Symmetric ciphers
  - Data Encryption Standard (DES) [block]
  - Advanced Encryption Standard (AES) [block]
  - RC4 [stream]
- Hash functions and MACs
  - MD5 [hash]
  - SHA1 + improved versions SHA-{224,256,384,512} [hash]
  - HMAC [MAC]
  - Poly1305-AES MAC [MAC]
- Some performance figures
- Assignments
  - 3DES (given DES & CBC) [crypto.implementations.TripleDesEcb]
  - HMAC (given MD5 and SHA1) [crypto.implementations.Hmac]

About notation

- We'll use the following symbols for bit operations
  - Bitwise OR: | **or** **OR**
  - Bitwise AND: **&** **and** **AND**
  - Bitwise XOR: **^** **xor** **XOR**
  - Bitwise NOT: **~** **not** **NOT**
  - Concatenation of bit strings: ||
  - Bit shift left and right **<<** **>>**
  - Bit rotation left and right **<<<** **>>>**
- Other Java/C notation is used, too

Keying material and algorithm abstraction

- Abstracting symmetric ciphers and MACs (black box)
  - Block ciphers usually differ only in block size (bits) and key size (bits/bytes)
  - MACs usually differ only in key size (bits)
- Abstraction helps APIs and key exchange protocols
  - Uniform interfaces (algorithm + one key)
- Key(ing) material
  - Key exchange protocols usually produce one important output: trusted keying material – usually a fixed size byte array or an (infinite) byte stream
  - This keying material is split up between the algorithms negotiated during key exchange
  - We'll see this in practice with how TLS works
- This week's assignment
  - 3DES consists of three consecutive passes of DES; each pass takes one 56-bit key (expressed as an 8-byte = 64-bit value with ignored parity bits)
  - 3DES thus basically takes three separate keys
  - However, to maintain the usual abstraction, 3DES is actually considered a black box, and it takes a 24-byte key which is **internally** split into separate keys

Data Encryption Standard (DES)

Background

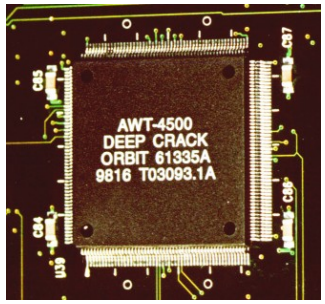
- Data Encryption Standard (DES) history
  - NBS (now NIST) wanted to develop a single encryption algorithm in 1972 => RFPs in 1973 & 1974 Federal Register
  - First good submission: IBM's earlier Lucifer with modifications
  - After rounds of critique, adopted in 1976 as federal standard (*)
- Standards
  - FIPS PUB 46-3: DES + 3DES
  - FIPS PUB 81: DES modes of operation
- DES contributed much to cryptographic research
  - Feistel networks and block ciphers modes of operation
  - Design criteria of DES have been partially revealed, but still partly unknown

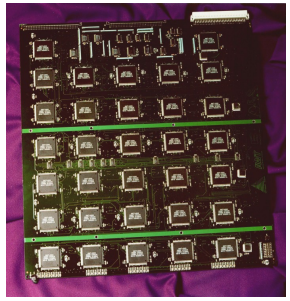
(*) However, FIPS PUB 46 was published in January 1977, a few months later

NIST = National Institute of Standards and Technology
NBS = National Bureau of Standards (now NIST)
FIPS = Federal Information Processing Standard

EFF DES Cracker

- http://www.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/
  - $250'000, DES crack in less than 5 days on average, brute force
  - Tests more than 90 billion keys / second
  - 1998 RSA Labs' DES Challenge II => 56-bit DES key in 56 hours
- => DES is no longer (never was?) secure for serious use



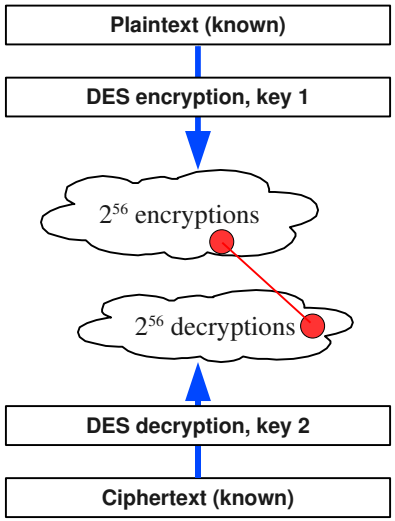


Images: http://www.cryptography.com/resources/whitepapers/DES-photos.html
http://www.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/

Security of DES

- DES too weak => Instead, 3DES ("TDEA") is used
  - 3DES EDE = encrypt (K1) – decrypt (K2) – encrypt (K3)
  - EDE allows for 1DES as well, by setting K2=K3
- Why not 2DES ?
  - Meet-in-the-middle attack; given plaintext **P** and ciphertext **C** ...
    - Encrypt plaintext **P** with all keys (1DES), length **n** blocks
    - Decrypt ciphertext **C** with all keys (1DES), length **n** blocks
    - Find corresponding blocks "in the middle", get 2DES keys
    - $\mathbf{n \times 2 \times 2^{56} = n2^{57}}$ DES operations, $\mathbf{n \times 8 \times 2^{57} = n2^{60}}$ bytes memory
    - => With enough memory, 2DES is not more secure than 1DES
  - 3DES effective key length is not 3x56 but ~2x56 (112) bits
- Many cryptographic attacks against DES, e.g.
  - Differential cryptanalysis (public 1989, Eli Biham & Adi Shamir)
  - Linear cryptanalysis (1992, Mitsuru Matsui, $\mathbf{2^{43}}$ known plaintexts)
  - No serious breaks (except short key length)

Meet-in-the-Middle Details – 2DES ~ 1DES



Attacker **encrypts** known plaintext with all 2^{56} possible keys. Result is 2^{56} intermediate values.

Attacker **decrypts** known ciphertext with all 2^{56} possible keys. Result is 2^{56} intermediate values.

If plaintext-ciphertext -pair is correct, there is at least one matching pair (and usually more) in these sets of 2^{56} values (because 2DES was used in encryption). The corresponding key values are a possible 2DES key pair, achieved with maximum of ~2^{57} brute force steps, ~2^{56} on average.

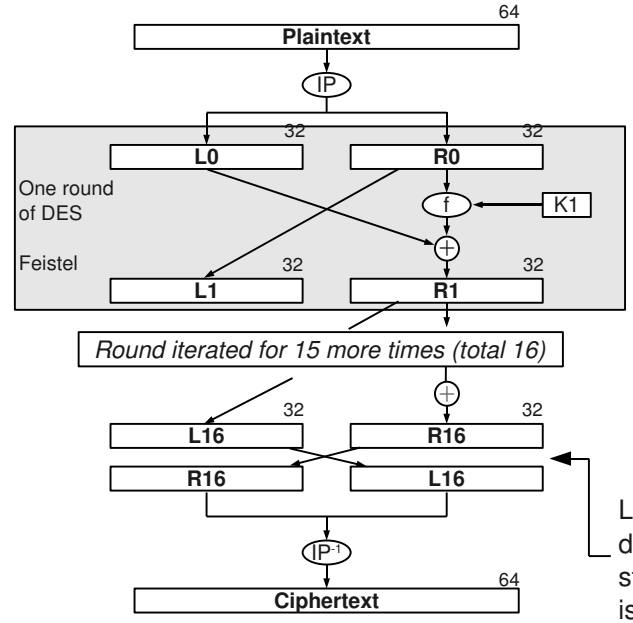
Potential matches can be verified using another plaintext-ciphertext -block, or by using more than one block initially.

Result: 2DES effective key length is ~56 bits
3DES effective key length is ~112 bits

DES parameters

- DES as a mathematical function
  - $C = DES_K(P)$ $P = DES_K^{-1}(C)$
- Key (K) size: 56 bits
  - Typically represented as 64 bits, with 8 parity bits (56 + 8 = 64)
  - Parity bits are, in practice, ignored (parity errors OK)
- Block (P and C) size: 64 bits (8 bytes)
  - DES encrypts and decrypts 64 bits at a time
- Initial permutation + 16 rounds + final permutation
  - Feistel cipher
  - Initial and final permutation are useless: they add no security
- Modes of operation
  - CBC is typically used

DES block structure

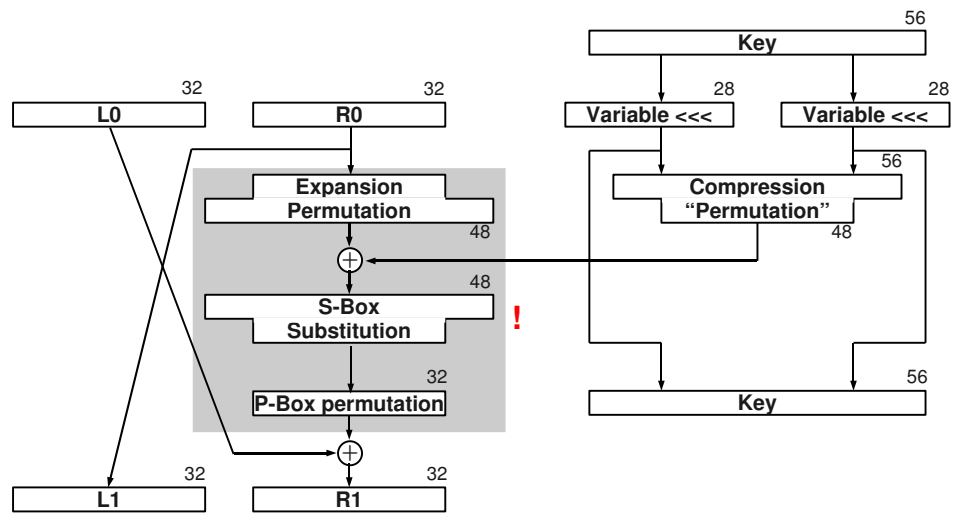


Each round:
- $L_i = R_{i-1}$
- $R_i = L_{i-1}$ XOR $f(R_{i-1}, K_i)$

Round keys (subkeys)
- $K_1, K_2, ..., K_{16}$
- Derived from key K

Last round "twist" allows decryption using same structure if key schedule is reversed.

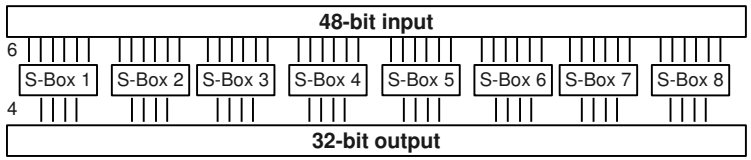
One round of DES in more detail



The round itself (Feistel structure) **Key schedule (round key computation)**

S-boxes up close

- S-boxes (substitution boxes) are critical to DES security
  - They are the only **non-linear** component of DES
  - Extensively studied
- There are eight distinct S-boxes, all 6-to-4-bit functions
  - The same eight S-boxes are used on each round
  - The (supposed) criteria for DES S-box design have been released
- In implementation, S-boxes can be thought of as arrays:
  - byte sbox1[64] = { ... }; // 64 indexes, 4-bit output values



Advanced Encryption Standard (AES)

Background

- Advanced Encryption Standard (AES) history
  - 1997: Federal Register requests for candidates to be AES
  - 1998-1999 => final five: MARS, RC6, Rijndael, Serpent, Twofish
  - Oct 2000: Rijndael selected as AES
  - Nov 2001: AES specification released
- Standards
  - FIPS PUB 197
- Basic parameters
  - Key size: 128, 192, or 256 bits
  - Block size: 128 bits (16 bytes)
  - Number of rounds: 10, 12, or 14 (depends on key length)
  - SP-network design, earlier design called Square (we'll see why)

Criteria for AES

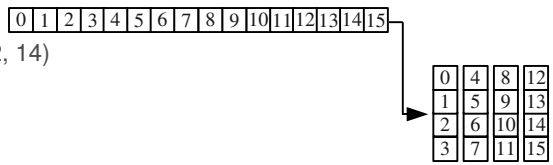
- The following criteria were set for AES
  - AES shall be publicly defined
  - AES shall be a symmetric block cipher
  - AES shall be designed so that the key length may be increased as needed
  - AES shall be implementable in both hardware and software
  - AES shall either be a) freely available or b) available under terms consistent with the American National Standards Institute (ANSI) patent policy

Security of AES

- Withstand linear and differential cryptanalysis
  - "Wide Trail Strategy"
  - S-boxes optimized for these criteria
- Algebraic cryptanalysis a threat
  - Known plaintext + ciphertext pairs => form system of algebraic equations and solve them
  - AES has an interesting algebraic structure, but no algebraic attack exists (yet)
- Side channel attacks against software AES implementation
  - Software implementation => difficult to get constant time ops
  - Allows timing attacks, e.g. D.J. Bernstein against OpenSSL AES
    - http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
    - Can be used to recover AES key by only inspecting timing of a network protocol encrypted using AES; relevance limited
  - Also cache-based (not direct timing) attacks
  - We'll cover timing attacks in more detail later in the course

Some preliminaries

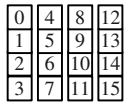
- Intermediate **State** represented as 4x4 byte array (128 bits)
  - Input block (128 bits) is converted to a square array (below)
  - Round operations operate on the array (rows, columns, or bytes)
- Operations on bytes and columns (words) are algebraic
  - Bytes: **GF(2^8)** (a finite field)
  - Columns: Four-term polynomials, with coefficients in **GF(2^8)**
  - We won't go into detail here, as it's not critical to understanding how AES works
- Basic definitions
  - 1 word = 1 column = 32 bits
  - **Nr** = number of rounds (10, 12, 14)
  - **Nb** = words in block (4)
  - **Nk** = words in key (4, 6, 8)



Representation of 128 bits (16 bytes) as a square matrix, each element is 1 byte (8 bits)

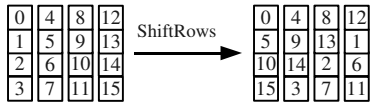
Overview of encryption

- Key expansion
  - Convert **Nk**-word key into **Nr+1** round keys (4x4 matrices)
- Encrypting a block
  - Convert input block into a 4x4 matrix of elements => state
  - AddRoundKey(state, roundkey[0])
  - For i = 1 to Nr
    - SubBytes(state)
    - ShiftRows(state)
    - if (not last round) MixColumns(state)
    - AddRoundKey(state, roundkey[i])
  - Convert state into a 128-bit block => output

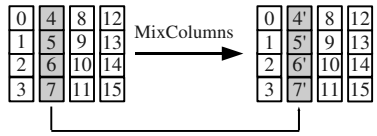


AES round operations

- SubBytes(state)
  - Substitute each byte of state using a fixed 256-entry lookup-table
  - The lookup-table is computed algebraically
- ShiftRows(state)
  - Shift rows (0,1,2,3) times left



- MixColumns(state)
  - Interpret each column as a 32-bit word and permute
  - The permutation is computed algebraically; each byte potentially affects all output bytes



- AddRoundKey(state, roundkey)
  - XOR the round key (4x4 matrix) into the state (also a 4x4 matrix)

Key expansion

- Definitions
  - Treat cipher key as an array of 32-bit words, key[0], ... key[Nk-1]
  - Define array w[0], ..., w[4*(Nr+1)-1]
- Initial entries in w[] copied directly from key
  - w[i] = k[i], for i = 0...Nk-1
- The rest are determined using the following algorithm
  - For i = Nk...4*(Nr+1)-1
    - temp = w[i-1]
    - If ((i % Nk) == 0) /* key “wraps” */
      - temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
    - Else
      - temp = SubWord(temp)
    - w[i] = w[i – Nk] xor temp
- The round key matrices are four-word parts of w[]

Key expansion functions

- Round constants, Rcon
  - Computed algebraically, basically pseudo-random values
  - One constant for each “reuse” of the original key
- SubWord
  - Apply the S-box (the same as in the actual cipher) to each byte in word
- RotWord
  - Rotate bytes in word once to the left
  - **[a, b, c, d]** => **[b, c, d, a]**

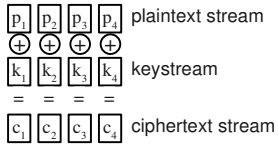
(Alleged) RC4 or “ARCFOUR”

Background

- RC4™ history (Alleged RC4)
  - Developed in 1987 by Ron Rivest for RSA Data Security, Inc.
  - Available under NDA; leaked to public in 1994 (Cypherpunks list)
    - “Interoperability” confirmed by NDA-bound people
  - Legal status
    - RC4 is still a trademark of RSA, so the cipher is often called ARCFOUR (or ARC4)
    - Compatible implementations seem to be legal, but many vendors still seem to license from RSA
- Specifications
  - Unofficial specifications
- RC4 used e.g. in 802.11 WEP (Wired Equivalent Privacy)
  - Infamous for its weakness – but not really RC4's fault

RC4 basics

- RC4 is a stream cipher
  - Operates 8 bits at a time
  - Keystream generator works independently of plaintext stream
  - Ciphertext stream = plaintext stream XOR keystream
- Two major components
  - Key Setup Algorithm (KSA)
  - Pseudo-Random Generation Algorithm (PRGA)
- RC4 uses a 256-byte array, S[i]
  - Initialization using KSA
  - Keystream generation using PRGA, updates S on-the-fly
  - Initial keystream elements often discarded to improve security (e.g. 256 first bytes)



RC4 / Key Setup Algorithm (KSA)

- RC4 state consists of:
  - The S-array (S[0], S[1], ..., S[255]), a permutation of {0, ..., 255}
  - Counters i and j
- Initialize state linearly: S[t] = t, for t=0..255
- Initialize key array: $K[t] = k_{t \% keylen}$, for t=0..255
  - Key repeats, modulo keylen
  - E.g. K[] = { k_0, k_1, k_2, k_0, k_1, k_2, k_0, ... }
- Then:
  - For i = 0 to 255:
    - j = (j + S[i] + K[i]) % 256
    - Swap S[i] and S[j]
- i and j are then reset to zero for PRGA

RC4 / Pseudo-Random Generation Algorithm (PRGA)

- After KSA, we have S[] set up, and i=j=0
- To generate one byte of keystream, K, do:
  - i = (i + 1) % 256
  - j = (j + S[i]) % 256
  - Swap S[i] and S[j]
  - t = (S[i] + S[j]) % 256
  - Return S[t]
- Encryption of one byte of plaintext is simply then:
  - E(p) = p XOR K()
- Keystream must not be reused
  - E.g. encrypting packets, per-packet-key = key + per-packet data
- RC4 performance is excellent
  - About 5 times faster than DES

A simple RC4 implementation in Java

```
public class rc4 {
    int[] S = new int[256];  // to avoid Java's signed semantics
    int i, j;

    void _swap(int i, int j) {
        int t = S[i]; S[i] = S[j]; S[j] = t;
    }

    public rc4(byte[] key, int initialDiscard) {  // Key Setup Algorithm (KSA)
        int[] K = new int[256];
        for(int t = 0; t < 256; t++) { S[t] = t; }
        for(int t = 0; t < 256; t++) { K[t] = ((int) key[t % key.length]) & 0xff; }
        for(i = 0; i < 256; i++) {
            j = (j + S[i] + K[i]) % 256;
            _swap(i, j);
       }
       i = j = 0;
       for(int t = 0; t < initialDiscard; t++) { _generate(); }
    }

    byte _generate() {  // Pseudo-Random Generation Algorithm (PRGA)
        i = (i + 1) % 256;
        j = (j + S[i]) % 256;
        _swap(i, j);
        return (byte) S[(S[i] + S[j]) % 256];
    }

    public byte[] encrypt(byte b[]) {
        byte p[] = (byte[]) b.clone();
        for(int t = 0; t < p.length; t++) { p[i] ^= _generate(); }
        return p;
    }
}
```

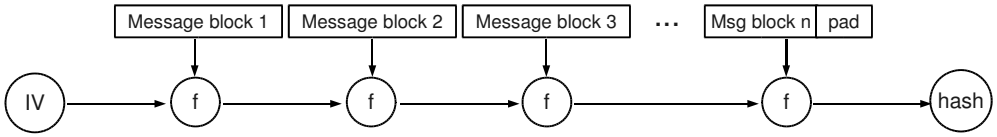
Message Digest 5 (MD5)

Background

- History
  - Ron Rivest (MIT): MD4, improved security in MD5 (1991)
  - Widespread use in cryptography and other applications
- Specification
  - RFC 1321 (freely available)
- Merkle-Damgård structure
  - Length extension attacks work
- MD5 collisions found in 2004 (pub. August 2004)
  - Wang et al: Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD
  - MD5 security is quite uncertain
  - Even so, MD5 will be in widespread use for some time

From previous lecture – hash function design

- Merkle-Damgård structure
  - Message processed (compressed) in fixed size blocks
    - Requires padding of message
  - Compression function **f** combines state + block => new state
  - Initialization vector (fixed) is initial state – hash is last state
- Security
  - If compression function **f** is collision resistant, overall construction is collision resistant (Merkle & Damgård)
- Commonly used
  - MD5 and SHA1; block size 512 bits



MD5 steps

- Data types
  - 512-bit block at a time, as 16 x 32-bit words (little-endian)
- Steps 1 & 2: Append padding and length
  - **[m_0 ... m_{b-1}]** => **[m_0 ... m_b 1 0 0 ... 0 <64-bit field: length in bits>]**
  - Total message size is a multiple of 512 bits, 1...512 bits padding
- Step 3: Initialize MD buffer ("IV")
  - **A=0x01234567**, **B=0x89abcdef**, **C=0xfedcba98**, **D=0x76543210**
- Step 4: Process message in 16-word blocks (512 bits)
  - Actual work done here; mix input word & produce new (**A**,**B**,**C**,**D**)
  - This is described in more detail on following slides
- Step 5: Output
  - MD5 is **A, B, C, D** concatenated into final 128 bit quantity

MD5 step for one 512-bit block of data

- Steps
  - 1. Copy state **(A, B, C, D)** => **(AA, BB, CC, DD)**
  - 2. Perform four passes on the data, mixing data and **(A, B, C, D)**
  - 3. Add into state: **A += AA**, **B += BB**, **C += CC**, **D += DD**
- Helper functions operating on 32-bit words (bitwise)
  - **F(X,Y,Z) = (X&Y) | (~X&Z)**
  - **G(X,Y,Z) = (X&Z) | (Y&~Z)**
  - **H(X,Y,Z) = X ^ Y ^ Z**
  - **I(X,Y,Z) = Y ^ (X | ~Z)**
- Lookup table T[1..64], where T[i] = int(2^{32} abs(sin(i)))
  - Produces "pseudorandom" constants, no other meaning
- Current data block, X[1..16]
  - Sixteen 32-bit words = 512 bits

MD5 rounds

- Four rounds operating on **(A, B, C, D)** and **X[1..16]**
  - 4x16 = 64 operations on 16 words of data
  - Each round is similar, but a helper function varies
- The following notation is used:
  - R1: **[abcd k s i] => a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s)**
  - R2: **[abcd k s i] => a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s)**
  - R3: **[abcd k s i] => a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s)**
  - R4: **[abcd k s i] => a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s)**

MD5 rounds

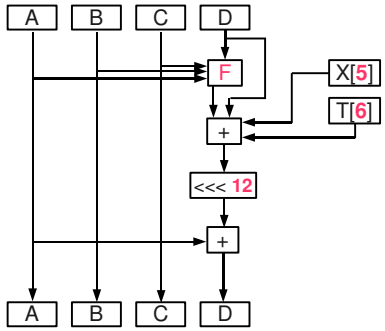
Round 1	Round 2	Round 3	Round 4
[ABCD 0 7 1]	[ABCD 1 5 17]	[ABCD 5 4 33]	[ABCD 0 6 49]
[DABC 1 12 2]	[DABC 6 9 18]	[DABC 8 11 34]	[DABC 7 10 50]
[CDAB 2 17 3]	[CDAB 11 14 19]	[CDAB 11 16 35]	[CDAB 14 15 51]
[BCDA 3 22 4]	[BCDA 0 20 20]	[BCDA 14 23 36]	[BCDA 5 21 52]
[ABCD 4 7 5]	[ABCD 5 5 21]	[ABCD 1 4 37]	[ABCD 12 6 53]
[DABC 5 12 6]	[DABC 10 9 22]	[DABC 4 11 38]	[DABC 3 10 54]
[CDAB 6 17 7]	[CDAB 15 14 23]	[CDAB 7 16 39]	[CDAB 10 15 55]
[BCDA 7 22 8]	[BCDA 4 20 24]	[BCDA 10 23 40]	[BCDA 1 21 56]
[ABCD 8 7 9]	[ABCD 9 5 25]	[ABCD 13 4 41]	[ABCD 8 6 57]
[DABC 9 12 10]	[DABC 14 9 26]	[DABC 0 11 42]	[DABC 15 10 58]
[CDAB 10 17 11]	[CDAB 3 14 27]	[CDAB 3 16 43]	[CDAB 6 15 59]
[BCDA 11 22 12]	[BCDA 8 20 28]	[BCDA 6 23 44]	[BCDA 13 21 60]
[ABCD 12 7 13]	[ABCD 13 5 29]	[ABCD 9 4 45]	[ABCD 4 6 61]
[DABC 13 12 14]	[DABC 2 9 30]	[DABC 12 11 46]	[DABC 11 10 62]
[CDAB 14 17 15]	[CDAB 7 14 31]	[CDAB 15 16 47]	[CDAB 2 15 63]
[BCDA 15 22 16]	[BCDA 12 20 32]	[BCDA 2 23 48]	[BCDA 9 21 64]

```
Round 1 notation      [abcd k s i] => a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s)
Round 2 notation      [abcd k s i] => a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s)
Round 3 notation      [abcd k s i] => a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s)
Round 4 notation      [abcd k s i] => a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s)
```


Example operation (round 1)

Round 1

```
[ABCD  0  7  1]
[DABC  1 12  2]
[CDAB  2 17  3]
[BCDA  3 22  4]
[ABCD  4  7  5]
[DABC  5 12  6]
[CDAB  6 17  7]
[BCDA  7 22  8]
[ABCD  8  7  9]
[DABC  9 12 10]
[CDAB 10 17 11]
[BCDA 11 22 12]
[ABCD 12  7 13]
[DABC 13 12 14]
[CDAB 14 17 15]
[BCDA 15 22 16]
```




```
Round 1 notation      [abcd k s i] => a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s)
Round 2 notation      [abcd k s i] => a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s)
Round 3 notation      [abcd k s i] => a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s)
Round 4 notation      [abcd k s i] => a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s)
```

Secure Hash Algorithm 1 (SHA1)

and

SHA-224, SHA-256, SHA-384, SHA-512

Background

- History
  - Principles based on MD4
  - Approved and published in 1995 by NIST
- Specifications
  - FIPS 180-2 (older specification FIPS 180-1)
  - RFC 3174 (republished for IETF use)
- SHA1 is now considered to be untrustworthy
  - Xiaoyun Wang – collisions for SHA1, effort 2^{63}
    - Rough birthday paradox collision attack is 2^{80} (around 50%)
    - Room for improvement of the 2^{63} attack
    - http://www.schneier.com/blog/archives/2005/08/new_cryptanalyt.html
  - => Stronger hashes should be used if possible
    - However, may not always be possible in the real world
    - MD5 and SHA1 are entrenched in protocols

SHA1 steps

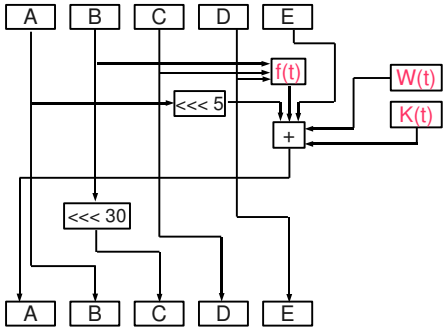
- Data types
  - Same as MD5
- Steps 1 & 2: Append padding and length
  - Same as MD5
- Step 3: Initialize state
  - **H0 = 0x67452301**, **H1 = 0xEFCDAB89**, **H2 = 0x98BADCFE**, **H3 = 0x10325476**, **H4 = 0xC3D2E1F0**
- Step 4: Process message in 16-word blocks (512 bits)
  - Expand the 16-word block into 80 words, **W(0)**, ..., **W(79)**
  - Update **H0**, ..., **H4** by iterating a step function, 80 iterations
  - This is described in more detail on following slides
- Step 5: Output
  - SHA1 is **H0, H1, H2, H3, H4** concatenated into 160-bit quantity

SHA1 step for one 512-bit block of data

- Function **f(t; B, C, D)**
  - **f(t;B,C,D) = (B & C) | (~B & D)** **(0 <= t <= 19)**
  - **f(t;B,C,D) = B ^ C ^ D** **(20 <= t <= 39)**
  - **f(t;B,C,D) = (B & C) | (B & D) | (C & D)** **(40 <= t <= 59)**
  - **f(t;B,C,D) = B ^ C ^ D** **(60 <= t <= 79)**
- Constants K(0), ..., K(79)
  - **K(t) = 0x5A827999** **(0 <= t <= 19)**
  - **K(t) = 0x6ED9EBA1** **(20 <= t <= 39)**
  - **K(t) = 0x8F1BBCDC** **(40 <= t <= 59)**
  - **K(t) = 0xCA62C1D6** **(60 <= t <= 79)**
- Expanded input data words W(0), ..., W(79)
  - **W(0)**, ..., **W(15)** directly from block
  - **W(t) = (W(t-3) ^ W(t-8) ^ W(t-14) ^ W(t-16)) <<< 1**, **t=16,...,79**

SHA1 step for one 512-bit block of data

- Steps for processing one block
  - 1. Expand block into **W(0), ..., W(79)** (previous slide)
  - 2. Set **A = H0**, **B = H1**, **C = H2**, **D = H3**, **E = H4**
  - 3. For **t=0...79** do
    - **TEMP = (A <<< 5) + f(t;B,C,D) + E + W(t) + K(t)**
    - **(E, D, C, B, A) = (D, C, B <<< 30, A, TEMP)**
  - 4. Set **H0 += A**, **H1 += B**, **H2 += C**, **H3 += D**, **H4 += E**



SHA-{224,256,384,512} summary

- History
  - Improved hash functions, followers of SHA1
  - Hash function strong enough to be used with AES
  - 128-bit security (brute force birthday collision attack)
- Specifications
  - FIPS PUB 180-2 “Secure Hash Standard”
  - Contains SHA1, SHA-224, SHA-256, SHA-384, SHA-512
- Design
  - Improved SHA1, changed constants, block sizes, etc.

Algorithm	Message size (bits)	Block size (bits)	Word size (bits)	Digest size (bits)	Security bits (birthday attack)
SHA-1	$< 2^{64}$	512	32	160	63 (hoped 80)
SHA-224	$< 2^{64}$	512	32	224	112
SHA-256	$< 2^{64}$	512	32	256	128
SHA-384	$< 2^{128}$	1024	64	384	192
SHA-512	$< 2^{128}$	1024	64	512	256

HMAC

HMAC in a nutshell

- Specification
  - RFC 2104
- Definitions
  - **H** is the underlying hash function, e.g. MD5
  - **L** = byte-length of hash result (MD5=16, SHA1=20)
  - **B** = byte-length of hash block (e.g. SHA1, MD5 => 64)
  - **ipad** = the byte 0x36 repeated **B** times: (0x36, 0x36, ..., 0x36)
  - **opad** = the byte 0x5c repeated **B** times: (0x5c, 0x5c, ..., 0x5c)
  - **K** = secret key, pad with zeroes to **B** bytes
    - Corner case: **K** is longer than **B** => replace **K = H(K)** first
- HMAC is then defined as
  - **HMAC(K, text) = H(K ^ opad || H(K ^ ipad || text))**
  - (Where || denotes concatenation, ^ denotes bit string XOR)

Poly1305-AES MAC

Poly1305-AES MAC

- Polynomial MACs
  - Construct a long polynomial with message blocks as coefficients
  - Evaluate polynomial (modulo something) with variable set to key
- Poly1305-AES MAC
  - D.J. Bernstein, Fast Software Encryption (FSE) 2005
  - Computation modulo 2^{130}-5
  - Uses AES internally
- Benefits
  - Provable security (relative to AES)
  - Very high performance
- Poly1305-AES MAC, one of many new proposed designs
  - We use it as an example of a more modern MAC design than MD5 and SHA*

Poly1305-AES MAC computation

- **Poly1305-AES$_{(k,r)}$(n,m) = h_r(m) + AES$_k$(n) (mod 2^{128})**
  - Keys: **k** = 128-bit AES key, **r** = additional key modulo 2^{130}-5
  - Inputs: **n** = unique 128-bit nonce, **m** = message (series of bytes)
- Computing the MAC
  - Message **m** is padded and split into 128-bit chunks, $c_1, ..., c_q$
  - Form polynomial $f(x) = c_1x^q + ... + c_qx^1 \pmod{2^{130}-5}$
    - Polynomial coefficients are 128-bit message chunks!
  - Evaluate polynomial at **r** (set **x** to **r**) => $h_r(m) = f(r)$
    - Why? It happens to work well in a security proof :-)
  - Compute "masking" value **AES$_k$(n)** straightforwardly using AES
    - Nonce never repeats => "mask" never repeats
    - The random "mask" prevents direct attacks on $h_r(m)$
  - Add "masking" value to evaluated polynomial => final MAC
- "Modern" construction – definition based on security proof

Some performance figures

Some performance figures

- Machine: AMD Athlon XP 2000+ @ 1671MHz, Linux kernel 2.6.9
- Figures obtained using **openssl speed** (MB/s = megabytes/second)

Algorithm	Speed MB/s (64 bytes)	Speed MB/s (8192 bytes)
MD2	2.7	4.4
MD4	45.1	336.5
MD5	39.0	290.6
HMAC-MD5	49.5	295.4
SHA1	32.6	173.6
RIPEMD-160	27.9	105.1
RC4	147.9	152.6
DES CBC	41.5	42.3
DES EDE3	14.5	14.6
AES-128	37.8	38.7
AES-192	32.8	33.4
AES-256	29.3	29.7

Summary

- Symmetric ciphers
  - Data Encryption Standard (DES)
  - Advanced Encryption Standard (AES)
  - RC4
- Hash functions and MACs
  - MD5
  - SHA1 and improved versions SHA-{224,256,384,512}
  - HMAC
  - Poly1305-AES MAC
- Some performance figures
- Assignments
  - 3DES (given DES & CBC) [crypto.implementations.TripleDesEcb]
  - HMAC (given MD5 and SHA1) [crypto.implementations.Hmac]