CNS Lecture 6

Block ciphers -- AES (Rijndael)
Stream ciphers
Key management
review

Assignment 5

Assignment 6

In the news
• Microsoft IE ActiveX remote code execution
• Microsoft powerpoint remote code execution (0-day)
• MAX OS X multiple vulnerabilities
• OpenSSL ASN.1 remote buffer overflow
• gzip, firefox, Adobe flash player, ...

You are here ...
Attacks & Defenses
• Risk assessment
• Viruses
• Unix security
• authentication
• Network security
Firewalls, VPN, PEAP, 802.1X

Cryptography
• Random numbers
• Hash functions
MD5, SHA, RIPEMD
• Classical + stego
• Number theory
• Symmetric key
DES, Rijndael, RC5
• Public key
RSA, DSA, D-H, ECC

Applied crypto
• 3DES
• SHA
• S/MIME
• Kerberos
• IPsec

Block ciphers
Feistel
• DES
• Lucifer
• Blowfish
• CAST

Non-Feistel
• IDEA
• RC2, RC5
• AES (Rijndael)
• For non-Feistel need invertible operations

Block cipher modes
• ECB, CBC, CFB, OFB, CTR
• applies to all block ciphers
• Padding, chaining and IVs
• hide repeated plaintext
• different error/attack properties

encryption does not guarantee message integrity!

Padding, IV's, and key generation in OpenSSL
• Encryption will pad to block size of cipher (DES: 8 bytes, AES 16)
  – e.g., 5 bytes in  ➔ 9 encrypted bytes out, 21 in  ➔ 24 out
  – May want to pre-pad with random “salt” to obscure same message
• OpenSSL standard API encryption pads with bytes of IV
  – EVP methods pads with byte count (PKCS 5)
  – pre-penda 8-byte magic “Salted__” and 8 byte random salt
• Cipher may be attackable if attacker knows IV
  – Best practice: derive IV from shared secret (assign 7) + nonce?
  – Need different IV when restarting encryption
  – OpenSSL EVP optionally derives IV from MD5 of key

• Converting a password to a key
  – Assignment 7 uses MD5
  – OpenSSL EVP_EncryptKey() generates key and IV from password using MD5

Encryption does not guarantee message integrity!
AES
Advanced Encryption Standard
- replace DES
- ‘97 call for algorithm
  - royalty-free, publicly disclosed
  - 128-bit block symmetric key cipher
  - key-sizes: 128, 192, 256

evaluation criteria
- security
- cost/performance (memory, computational efficiency)
- architecture -- simplicity, flexibility
- hardware/software suitability
- ‘98 15 candidates
- ‘99 5 finalists (MARS, RC6, Rijndael, Serpent, Twofish)

Mars (IBM)
- Feistel (1/4, 3/4 rather than even split)
- 8 (unkeyed) pre/post-whitening rounds
  - addition, XOR, and 512x32 S-box
  - addition, XOR, and S-box, multiplication, data-dependent rotations, key addition
- S-box pseudo-random and tested
- key schedule: linear transform and S-box, with pattern matching to eliminate weak subkeys

Serpent
- substitution-linear transform network
- non-Feistel - need inverse S-box and inverse transform
- 256-bit key, 128-bit blocks, 32-bit work units
- bit-slice mode
- 8 4x4 DES-S-box S-boxes
  - pseudo-random with testing
- 32 rounds (most secure, slow)
  - XOR, S-boxes, linear transform
  - avalanche after 3 rounds
- key schedule: affine recurrence with S-boxes

Rijndael
- Klein-doll (or rain doll)
- son of Square cipher
- substitution-linear transform network
- non-Feistel (need inverse)
- plaintext block (16 bytes) treated as 4x4 array (state array)
  - 10/12/14 rounds (128/192/256 key size)
  - round constants eliminate symmetry
  - key schedule: linear transform and S-box, with pattern matching to eliminate weak subkeys

Rijndael setup -- S box setup
- S-box (and inverse S-box) (256-byte) can be calculated or pre-built (time vs memory)
  - S-box creation:
    1. initialize to 00 … FF
    2. calculate multiplicative inverse of each element over GF(2^8)
    3. transform bits of byte with XOR with constant
  - S-box (and inverse S-box) (256-byte) can be calculated or pre-built (time vs memory)

Rijndael setup -- key expansion
- Key expansion (108 key → 1768 subkeys)
  - 4 byte rotate, S-box, XOR with const.
  - Subkey (4-4 byte words) for each round plus one more subkey
  - For AES-128, 11 subkeys (10 rounds + 1)
- Rationale
  - Resistant to attacks
  - Round constants eliminate symmetry
  - Knowledge of some bits of key or subkey does not help calculating other subkeys
  - Diffusion
  - Speed
  - Can generate subkeys on the fly (time/space)

Rijndael setup – key expansion
- Key expansion (108 key → 1768 subkeys)
  - 4 byte rotate, S-box, XOR with const.
  - Subkey (4-4 byte words) for each round plus one more subkey
  - For AES-128, 11 subkeys (10 rounds + 1)
- Rationale
  - Resistant to attacks
  - Round constants eliminate symmetry
  - Knowledge of some bits of key or subkey does not help calculating other subkeys
  - Diffusion
  - Speed
  - Can generate subkeys on the fly (time/space)
Encryption is not identical to decryption. Decryption is same sequence of transformations, but using inverse transforms. Change of subkeys needed. Need rounds+1 subkeys. Note last round has no mixcolumn. First/last step is add round key. Why?

Rijndael stages
- Byte substitution -- S-box (nonlinear, strength)
- Shift rows -- permutation (diffusion)
- Mix columns -- substitution using arithmetic over GF(2^8)
  - multiplication is polynomial multiplication mod x^4 + 1 over GF(2^8)
  - easy in hardware and fast
  - table-lookup in software (256-entry multiplication table)
  - each element of a column is function of all the elements of the column (mixing)
- Add round key -- XOR with round key (simple Vernam/XOR cipher)
  - on 32-bit processor, round transformation can be done in set of table lookups
  - table lookups contribute to speed of Rijndael and prevent timing/power attacks
  - Potential for parallelism and each stage is reversible → decryption works.

Rijndael data structures
- 128-bit (16 byte) data block treated at 4x4 byte array

Rijndael round

Row and column transformations
- Row transform
  - Row 1 -- unchanged
  - Row 2 -- rotate left 1 byte
  - Row 3 -- rotate right 2
  - Row 4 -- rotate right 1
  - The row transform shifts column values, the 4 bytes of one column are spread out to four different columns.

- Mix column
  - each byte of a column is mapped into a new value that is a function of all four bytes in the column.
  - matrix multiply is over GF(2^8)

Substitute byte and add round key
Rijndael decryption

• Each stage is reversible
  alter direction of shift rows
  Invert Sbox
  Invert mixcolumn (mod x^4 + 1)
  \[ a(x) = 03x^3 + 01x^2 + 01x + 02 \]
  \[ a^{-1}(x) = 0Bx^3 + 0Dx^2 + 09x + 0E \]
  • Modified round key
    pre-calculate when making subkeys

So decryption differs slightly from encryption.

Finite field of dreams

• What we’d like is arithmetic over a finite field
  – Computers do better with finite (discrete) arithmetic
  – Field is associative, commutative, etc, with additive inverse, multiplicative inverse
  – Works for arithmetic mod a prime, e.g (5/4) mod 7 = 3
  – But computer “words” are usually powers of 2, (5/4) mod 8 =

• Stay tuned … corn fields, wheat fields, Galois fields 😊

Rijndael and polynomial arithmetic

• Rijndael utilizes polynomial arithmetic in two ways (really one)
  – Invertible arithmetic (finite field) over 8-bit numbers (+ x )
    invertible S-box
    linear transforms (mixcolumn)
  – 4-byte arithmetic (constant poly is relatively prime to x^4 + 1)
    invert mixcolumn mod (x^4 + 1)
    \[ a(x) = 03x^3 + 01x^2 + 01x + 02 \]

Polynomial arithmetic used by CRC’s, Rijndael, LFSR’s, ECC

Polynomial arithmetic

• addition and multiplication
  \[ (3x^4 + x^2) + (x^3 + 6x^2) = 3x^4 + x^3 + 11x^2 \]
  \[ (3x^4 + 2x^2)(x+1) = 3x^5 + 5x^4 + 2x^2 \]

• division, 3x^7 + x^5 + x^2 - 2 divided by x^4 - 1
  equals 3x^3 + x with remainder 3x^3 + x^2 + x = 2

• Coefficients can be integers or /mod p (Z_p)
  \[ (3x^4 + 5x^2) + (x^3 + 6x^2) \mod 7 = (3x^4 + x^3 + 1x^2) \mod 7 = 3x^4 + x^2 + 4x^2 \]

see CRC reading or Rijndael spec or Sect Ch 4 and Ch. 5 appendix

Polynomial arithmetic over GF(2)

• hardware influence: use coefficients mod 2
  – Think of each int as a coefficient
  – addition/subtraction in XOR, multiply is AND
• polynomial: \( x^0 + x + 1 \) is 1011
  \( (x^0 + x) + x^4 + x^2 + x = x + x \)

• Hardware/software: fast, XOR, and table lookups

Galois fields: finite fields of order \( p^n \), written GF(p^n)
Modular polynomial arithmetic over GF(2^n)

- Do polynomial multiplication mod an irreducible polynomial
- Notion of prime/irreducible polynomials
  \( f(x) \) is irreducible if it cannot be expressed as product of two polynomials
- GF(2^8) has 30 irreducible polynomials, Rijndael uses the first one
  \( x^8 + x^4 + x^3 + x + 1 \)
- Primitive polynomials (a subset of irreducible polynomials) generate all elements of an
  extension field from a base field (used in LFSR)
  Over GF(2^n) there are \( \phi(2^n-1)/n \) primitive polynomials
- \( \phi(n) \) number of elements relatively prime to \( n \) (Euler's totient)

<table>
<thead>
<tr>
<th><em>primitive polynomials</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Extended Euclid algorithm can be used to find the multiplicative inverse
of a polynomial (exists if mod irreducible polynomial)

\[ (x^7 + x + 1)^{-1} = x^7 \]

or \( 10000011 \times 1000000 = 00000001 \)

poly arithmetic in GF(2^n)

Finite field GF(2^3) – only two irreducible polynomials: \( x^3 + x^2 + 1 \) \( x^3 + x + 1 \)

Multiplication by \( x \) (010): shift left, if bit shifted out is 1 XOR in 011

\( \oplus \)

Addition is just XOR

\[ (3 \oplus 2 \oplus 1 \oplus 0) \mod (3 \oplus 2 \oplus 1 \oplus 0) = 0 \]

\[ (3 \oplus 2 \oplus 1 \oplus 0) \mod (3 \oplus 2 \oplus 1 \oplus 0) = 0 \]

\[ (54)_{16} \mod 8 \oplus (5 \times 4^{-1}) \mod 8 \]

\[ \oplus \]

no solution

Every element has a multiplicative inverse

Now 5/4 has a solution \( \odot = 2 \)

Rijndael poly arithmetic

- Coefficient of polynomial in GF(2^n) represented by 8-bit number (byte)
- Addition of two polynomials is just XOR of the two 8-bit numbers
  \( (x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2 \)
  \( 01010111 \oplus 10000011 = 11010100 \)
- Multiplication is more complicated (need modular reduction), but still is just
  shifts and XORs
  - Rijndael prime poly: \( x^8 + x^4 + x^3 + x + 1 \)
  - For multiplication by \( x \), a shift and XOR in \( 00011011 \)
  - Modular poly multiply mod \( (x^4 + 1) \) (e.g. 4 bytes, each in
    GF(2^4))
    \[ (02)x^4 + (03)x^3 + (02)x + (02) \mod (x^4 + 1) \]
  - But it is implemented as multiply and add on GF(2^4)

CRC's and polynomial arithmetic -- sidebar

- CRC is remainder in dividing message by polynomial
  - Think of message as long string of bits (polynomial)
  - Can be implemented with XOR's, shifts, and a table lookup
  - Fast in hardware
  - \( \oplus \) implementation: \( r = r \oplus m \oplus t \)
  - \( m \oplus t \) is a single bit error
  - All 2-bit errors
  - No 3-bit errors
  - \( \oplus \) errors correct

- Worry about what errors are not detected?

- Some popular CRC polynomials:
  - 16 bits:
    - (16,12,5,0) \( x^{16} + x^{12} + x^5 + 1 \) (X25 standard)
    - (16,15,2,0) \( "CRC-16" \)
  - 32 bits:
    - (32,26,23,16,12,11,10,8,7,5,4,2,1,0) \( "CRC-32" \)
    - Ethernet

1. Shift the register left by one byte, reading in a new message byte.
2. XOR the top byte just rotated out of the register with the next message byte to yield an index into the table \( (0:255) \).
3. XOR the table value into the register.
4. Goto 1 if more message bytes.
Rijndael in C

```c
/* BC byte count rk round key S sbox*/
/* plaintext in a */
KeyAddition(a,rk[0],BC);
/* ROUNDS-1 ordinary rounds */
for(r = 1; r < ROUNDS; r++) {
    Substitution(a,S,BC);
    ShiftRow(a,0,BC);
    MixColumn(a,BC);
    KeyAddition(a,rk[r],BC);
}
/* Last round is special: there is no MixColumn */
Substitution(a,S,BC);
ShiftRow(a,0,BC);
KeyAddition(a,rk[ROUNDS],BC);
```

Rijndael/AES in OpenSSL

```c
/* Command line (uses EVP mode, prepend magic and salt, pad with byte count)*/
evp -aes-128-cbc -in plain.txt -out enc.dat -pass pass:boo
```

```c
/* API*/
#define AES_BLOCK_SIZE 16
int AES_set_encrypt_key(const unsigned char *userKey, const int bits,
                        AES_KEY *key);
int AES_set_decrypt_key(const unsigned char *userKey, const int bits,
                        AES_KEY *key);
void AES_cbc_encrypt(const unsigned char *in, unsigned char *out,
                     const unsigned long length, const AES_KEY *key,
                     unsigned char *iv, const int enc);
```

AES and java

```java
// Get the KeyGenerator
KeyGenerator kgen = KeyGenerator.getInstance("AES");
bgen.init(128); // 192 and 256 bits may not be available
SecretKey skey = kgen.generateKey();
byte[] raw = skey.getEncoded();
SecretKeySpec skeySpec = new SecretKeySpec(raw, "AES");
// Instantiate the cipher
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, skeySpec);
byte[] encrypted =
    cipher.doFinal((args.length == 0 ?
                   "This is just an example" : args[0]).getBytes());
System.out.println("encrypted string: " + asHex(encrypted));
```

Rijndael in hardware

- 8-bit processor (smartcard)
  - adjkroundkey is a bytewise XOR
  - shiftrows is byte rotates
  - subbyte is table lookup (256 byte table)
  - mixcolumn is XORs and a table lookup (256 byte table)

- 32-bit processor
  - Operate on 32-bit words rather than bytes
  - 4 table lookups and four XORs per column per round (fast)
  - Need 4 256-word (1024 byte) tables

  - Can trade off memory space for computation time

AES selection criteria

- security
- software implementations C/java, 8/32/64-bit processors
- ROM/RAM requirements
- hardware implementations ASIC/FPGA
- instruction-level parallelism
- speed
- susceptibility to timing/power attacks
- encryption vs decryption
- key agility (switch keys quickly)
- versatility (block/key/round sizes)
AES assessment

- High security: mars, serpent, twofish
- Software: Rijndael good 8-64, rc6 good, serpent slow
- Key schedule: fast, twofish slow
- Space: Rijndael/serpent, mars, not.
- Hardware: Rijndael/serpent, good, mars average.
- Attacks: serpent/Rijndael good, twofish ok, rc6/mars bad
- Enc/dec: twofish, mars, rc6 good, serpent last.
- Key agility: twofish/serpent, good, Rijndael ok, rc6 last.
- Parallelism: rijndael best.

Votes:
rijndael(86), serpent(59), twofish(31), rc6(23), mars(13)

Choosing AES
(Take from Twofish Paper)

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Speed</th>
<th>Speed Factor</th>
<th>Simplicity (code size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serpent</td>
<td>62</td>
<td>69</td>
<td>3.56 541K</td>
</tr>
<tr>
<td>MARS</td>
<td>23</td>
<td>34</td>
<td>1.90 85K</td>
</tr>
<tr>
<td>RC6</td>
<td>15</td>
<td>43</td>
<td>1.15 48K</td>
</tr>
<tr>
<td>Rijndael</td>
<td>18</td>
<td>20</td>
<td>1.11 98K</td>
</tr>
<tr>
<td>Twofish</td>
<td>16</td>
<td>18</td>
<td>2.67 104K</td>
</tr>
</tbody>
</table>

(Rijndael style encrypt)

Rijndael AES evaluation

General Security:
Rijndael has a known security attack. Rijndael uses 5 round ones in multiple components.
Rijndael appears to have an adequate security margin, though several other ciphers
are believed to have a more secure margin.

Software Implementation:
Rijndael performance experiments and description tests across a variety of platforms,
including 32-bit and 64-bit platforms, and DSP. However, there is a difference in performance
with the same key size because of the increased number of rounds that are performed
in Rijndael's implementation. Performance figures reflect the current state of performance
and are based on results from the National Security Agency.

Hardware Implementation:
In general, Rijndael is very well suited for commercial environments where other
encryption or decryption is implemented for hardware. It has very low area and clock
requirements. A disadvantage is that Rijndael's encryption requires both encryption
and decryption to be implemented simultaneously. Although it performs reasonably well,
the implementation can be expensive for these environments. The key schedule for encryption is
more expensive than for decryption.

AES vs DES

- Can you match the DES steps with the Rijndael steps?

<table>
<thead>
<tr>
<th>DES steps</th>
<th>Rijndael</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate subkeys</td>
<td>Shiftrows</td>
</tr>
<tr>
<td>Permutations</td>
<td>Mix column</td>
</tr>
<tr>
<td>Xor sub-key with right half</td>
<td>Byte substitution</td>
</tr>
<tr>
<td>Xor (S-box) with right half</td>
<td>Add round key</td>
</tr>
<tr>
<td>Swap left and right</td>
<td>Generate round keys</td>
</tr>
</tbody>
</table>

DES: substitution & permutation (tuned only for hardware)
AES (Rijndael): substitution & linear transform

Rijndael AES evaluation cont.

Attention to Implementation:
The encryption and decryption in Rijndael use the same round function and hence
are not easily distinguishable from each other. However, the key schedule
for encryption is more expensive than for decryption.

Encrypted vs. Description:
The encryption and decryption functions in Rijndael differ. DES makes use of
encryption and decryption functions. Although they are very similar, they
are not identical. The implementation of encryption and decryption in
Rijndael is not significantly different from those functions, although the key
schedule in encryption is more expensive than in decryption.

Key Agility:
Rijndael supports a key-dialing mechanism for encryption. Rijndael requires a new-time
key in each encryption. In addition, the key schedule in Rijndael is more
secure than in DES. This makes Rijndael an attractive choice for systems
where key agility is important.

Other Concerns and Optimization:
Rijndael has been optimized for maximum area/throughput ratio in standard
implementations because of the additional number of rounds. For fully
optimized implementations, the performance can be significantly better than
the default implementation.

Whirlpool (hash function)

- Uses AES-like encryption function (W) to mix bits
  - based on polynomial arithmetic but fast (shifts and XORs)
  - added to OpenSSL 0.9.9
- 512-bit hash (faster than SHA-512)
- More secure? ... test of time
AES and whirlpool hash

<table>
<thead>
<tr>
<th>W</th>
<th>AES</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>128</td>
</tr>
<tr>
<td>Key size bits</td>
<td>128</td>
</tr>
<tr>
<td>Mappings</td>
<td>input is mapped one-to-one</td>
</tr>
<tr>
<td>Number of rounds</td>
<td>10</td>
</tr>
<tr>
<td>W (round function)</td>
<td>10, 12, or 14</td>
</tr>
<tr>
<td>AES-256</td>
<td>10</td>
</tr>
<tr>
<td>AES-192</td>
<td>10</td>
</tr>
<tr>
<td>AES-128</td>
<td>10</td>
</tr>
</tbody>
</table>

| AES-256 | 10 |
| AES-192 | 10 |
| AES-128 | 10 |

| AES-256 | 10 |
| AES-192 | 10 |
| AES-128 | 10 |

Block cipher advances

- Variable key length
- Mixed operators (non-linear) (Bent functions)
- Key/Data-dependent rotations (RC5)
- Key-dependent S boxes (blowfish)
- Round-dependent functions
- Whitening (xor key material before first round and after last round)
- Complex sub-key generation (blowfish)
- Variable block lengths and rounds and substitution
- Operate on both halves (blowfish/RC5)
- Mitigate linear/differential cryptanalysis
- Optimized for hardware/software

Block cipher summary

- 64-bit or 128-bit blocks
- DES
  - small key (use 3DES)
  - last of time
  - widely available
  - S boxes are strength
- IDEA
  - optimized for hardware/software
- Blowfish
  - key-dependent S boxes, fast
- Rijndael
  - S-boxes, optimized for hardware/software
- CAST
  - nonlinear S-boxes, round-dependent functions
- AES
  - S-boxes, optimized for hardware/software

Need key padding, IV, and ECB/CBC/OFB/CFB/CTR

CCM – encryption and authentication

- Use 128-bit encryption (AES) to do both authentication and encryption in one pass using same key (RFC 3610, NIST 800-38C)
- Use AES-CBC to calculate authenticator over message and nonce (e.g., message number)
- Append authenticator to message
- Use AES in counter mode to encrypt message and authenticator
- Two encryptions per message block
- Many parameters to select, prepend mode to message
- Used in 802.16 (wireless MAN, WiMAX)
- Though it is best practice to use different key for authentication and encryption, it’s OK here to use one key because of “shared” nonce
- Remember encryption does not provide authentication

Stream ciphers

- encrypt a byte/bit at a time
- XOR plaintext with keystream
- fast
- exportable
- used by RADIUS/TACACS+
- error properties:
  - change a byte in c_i?
  - lose c_i?

Encryption with a hash function

- (pre) compute a (pseudo) one-time pad (keystream)
- b_i = Hash(key, IV)
- c_i = p_i ⊕ b_i
- p_i = c_i ⊕ b_i
- why IV? why use key each time?
- stream cipher (byte at a time)
- exportable
Stream cipher from a block cipher

- Either OFB or CFB mode or CTR
- Can pre-compute key stream in OFB or CTR

**RC4**

- trademark of RSA
- used in several products (WEP, SSL, WORD)
- fast
- synchronous, 8x8 S box (evolves)

1. Fill S box with 0 to 255 and take key K
2. For i = 0 to 255 // rearrange Sbox according to key
   3. j = (i + K) mod 256
   4. swap Sj and Si

/Stream generation
5. i,j = 0
6. while (true)
   7. i = (i+1) mod 256
   8. j = (j + Si) mod 256
   9. swap Si and Sj // rearrange Sbox
   10. t = (Si + Sj) mod 256
   11. b = St
   12. XOR plain/cipher text byte with b

LFSR keystream

- efficient in hardware (shift XOR)
- where to tap (connection polynomial)
- full period (2^n - 1) polynomial is primitive mod 2
- example, x^5 + x + 1 (L.O.D.1)
- n-bit “key” is initial setting (seed)
- can solve single LFSR, so use several
-shrinking generator – use output of first LFSR to select drop bit of 2nd LFSR
- GSM A5 uses XOR of 3 LFSRs
- US cellular CDMA 3-s 52-bit LFSRs

Combining LFSR’s

- **Alternating step generator**
  - select R2 or R3 based on R1
- **Shrinking generator**
  - select R2 only when R1 outputs a 1

GSM A5

- 3 LFSRs with periods (19, 22, and 23 - 64-bit key)
- XOR S box
- Output “clocked” by majority function from taps at 8, 10, and 10
- Clocked means register is shifted with its new feedback input
- Without clocking, period would be (2^19 - 1)(2^22 - 1)(2^23 - 1), but experiments show really only 4/3 (2^23 - 1)
- Only 70% of seeds produce different keystreams

SNOW

- Version 1 weak, version 2 better
- LFSR (16x32 bits) plus finite state machine (FSM)
- 32-bit operations / output
- 8x8 bit S box
- 128 or 256 bit key
- 128-bit IV
LFSR summary

- Fast/simple in hardware
- Subject to correlation attacks with known plaintext
- Need non-linear combinations
- Use "secret" connection polynomial (c_i)

Polynomial arithmetic over GF(2^n) used in Rijndael, CRC's, ECC, and LFSR's

Stream ciphers

- Byte or bit based
- Efficient in hardware (LFSR) based on XOR: c_i = p_i \oplus r_i
- The ultimate: ONE-TIME-PAD, everything else repeats
- Hash-based PRNG's are good approximations
- TROUBLE if you reuse the key stream:
  - If plaintext/ciphertext pair known, you have the keystream
  - If PRNG period too short (WEP/RC4 GSM), keystream will repeat
  - If you have multiple plaintext's encrypted with same keystream (Microsoft Excel/WORD), you can XOR ciphertexts and with word/character frequencies derive plaintexts: P_1 \oplus P_2

Polynomial arithmetic over GF(2^n) used in Rijndael, CRC's, ECC, and LFSR's

Use a block cipher if you can.

Choosing a cipher

- depends on application
- type: stream or block
- block mode: CBC, ECB, OFB, CFB, CTR
- compact (smart card)
- strength (key length, lifetime, test of time)
- licensed
- available/portability
- performance
- error properties (mode/losses) -- you need separate integrity check (hash)
- tested, widely used
- worry about padding and IV
- openSSL: DES, 3DES, AES, RC4, CoST
- don't build your own or buy snake oil

PAIN

Does symmetric key encryption provide:
- Privacy?
- Authenticity?
- Integrity?
- Non-repudiation?
- Availability?
- Virus protection?
**Key generation**

- choose strong keys
  - passwords/phrases
  - length
  - mixture: upper, lower, special, numerics
  - good generator/verifier
  - dictionary attacks
- random keys
  - unpredictable
  - random sources (keystrokes, system info, /dev/random)
  - mixing (MD5, X9.17)
  - watch out for rand()

---

**Key length**

- **Size matters**
  - depends on value of information and resources of attacker
  - depends on lifetime of secrets
  - assume algorithm perfect, then brute force
  - one more bit of key, doubles attacker’s work factor (exponential)

<table>
<thead>
<tr>
<th>Type</th>
<th>Lifetime</th>
<th>Key (in bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tactical military</td>
<td>minutes</td>
<td>56-64</td>
</tr>
<tr>
<td>product announce</td>
<td>days</td>
<td>64</td>
</tr>
<tr>
<td>business plan</td>
<td>years</td>
<td>64</td>
</tr>
<tr>
<td>trade secrets</td>
<td>decades</td>
<td>112</td>
</tr>
<tr>
<td>nuclear secrets</td>
<td>40 yrs</td>
<td>128</td>
</tr>
<tr>
<td>spy IDs</td>
<td>50 yrs</td>
<td>128</td>
</tr>
<tr>
<td>personal info</td>
<td>50 yrs</td>
<td>128</td>
</tr>
</tbody>
</table>

- 128-bit key: using all the computers in the world, and if they could do a million encryptions/sec, it would take a million times the age of the universe!

(AES key sizes: 128, 192, 256)

---

**Brute force key attacks**

- symmetric key
  - time and cost
  - software, FPGA, ASIC
  - hacker, corporate, government
  - 40-bit key: $400 FPGA, 5 hours
  - EFF DES cracker $250K, 3 days
  - DES key breaking ($1M/4 hr.) within budget of large corporation or criminal organization
  - $300M, DES keys in 12 seconds
  - 76-bit, $10M/6 yrs, $500M/70 days
  - recommend 76-90 bits key today, 128

---

**Brute force**

- equivalent resistance to attack (key length in bits)

<table>
<thead>
<tr>
<th>Type</th>
<th>Lifetime</th>
<th>Key (in bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetric</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Public</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>384</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>768</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>1792</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>2304</td>
<td></td>
</tr>
<tr>
<td>ref. Schneier</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- public key vulnerable to improvements in factoring algorithms (or discrete log)

---

**Key lifetime**

- lifetime is a function of key length (work factor for brute force)
- the more a key is used, the greater the loss if compromised
- the longer a key is used, the more likely it will be compromised
- lifetime of info (message, signature, file)
- amount of data encrypted can determine lifetime
  - bad guy accumulates ciphertext for cryptanalysis
  - for DES, don’t send more than $2^{24}$ bits under the same key
  - at 1 Gbit/sec, 5 minutes
- key hierarchy (master key, session key)
  - use master key only to encrypt temporary session keys

---

**Updating keys**

- Kerberos/PKCS/PGP/PKI include ticket/key lifetime fields
- password aging
- public keys, typically 2 yrs max
- may need key archive (key id with material), or re-key material
- archive CRL’s too (PKI : Certificate Revocation List ... later)
- risk in distributing new keys
  - need (secure) key renewal, key update protocol
  - perfect forward secrecy (don’t use old key to generate/send new key)
Key distribution

How to get keys (code books, one-time pads) to the end users?

- courier
  - Codebook (Enigma) get key of the day
- secret keys -- out of hand, splitting
- third party generates key and delivers to A and B
- KDC (Kerberos/DCE) -- must be secure
- key update using old key
  - Bob key encrypting key/master key
- ISAKMP/Oakley, SKIP, Photuris (IPsec)
  - set up key on behalf of application
- Diffie-Hellman (perfect forward secrecy)
- public key cryptography

KDC

key distribution center

- N keys for N nodes
  - one for every pair
- KDC:
  - Just N KDC keys for N users
  - KDC generates session key for Bob and Alice, sends it to them encrypted with their respective KDC key
  - KDC must be secure
- protocol for requesting session key
- More on key distribution when we look at Kerberos

Key recovery

key recovery is when I can find out my key
Key escrow is when you can find out my key.

- good business sense
- key escrow
  - Clipper chip -- Big brother
  - Clipper III -- and export carrot
- commercial recovery (Entrust)
- secret sharing (Schneier)
  - split secret into n parts (n > m)
  - any m of them can be used to reconstruct secret
  - several algorithms (Schneier): Shamir, m-degree polynomial
  - part of a PKI?

Clipper chip

- Alice and Bob create a session key
  - (Diffie-Hellman, KEA, ...)
  - clipper encrypt message M with Skipjack
- LEAF is transmitted with encrypted message
  - Law Enforcement Agent’s Field (128 bits)
    - Ef(U, id, ac)
    - f secret family key (chips know, Key Escrow Decrypt Processor knows)
    - id is chip id (32b)
    - ac authentication code (16b) (used by receiving chip to verify authentic LEAF)
    - receiver won’t “decrypt” if LEAF is “bad”
    - u chip-specific key
      - u = k1 ⊕ k2 is split between two agencies
        - given k1 and k2, construct u

skipjack

- NSA encryption for Clipper chip (FORTEZZA)
- algorithm secret (till ‘98), hardware tamper-resistant
- reviewed by panel of experts
- strength not dependent on secrecy of algorithm
- design started in ’85
- evaluation completed in ’90
- specs
  - iterative block cipher
  - 64-bit block
  - 80-bit key
  - 32 rounds, XOR
  - 4x16 bit shift register/counter with 4-round Feistel using 8x8 S box

Clipper chip

- Alice and Bob create a session key
  - (Diffie-Hellman, KEA, ...)
  - clipper encrypt message M with Skipjack
- LEAF is transmitted with encrypted message
  - Law Enforcement Agent’s Field (128 bits)
- Ef(U, id, ac)
  - f secret family key (chips know, Key Escrow Decrypt Processor knows)
  - id is chip id (32b)
  - ac authentication code (16b) (used by receiving chip to verify authentic LEAF)
  - receiver won’t “decrypt” if LEAF is “bad”
  - u chip-specific key
    - u = k1 ⊕ k2 is split between two agencies
      - given k1 and k2, construct u

skipjack

- NSA encryption for Clipper chip (FORTEZZA)
- algorithm secret (till ‘98), hardware tamper-resistant
- reviewed by panel of experts
- strength not dependent on secrecy of algorithm
- design started in ’85
- evaluation completed in ’90
- specs
  - iterative block cipher
  - 64-bit block
  - 80-bit key
  - 32 rounds, XOR
  - 4x16 bit shift register/counter with 4-round Feistel using 8x8 S box
Protecting keys

- don’t PostIt
- one-time passwords or encrypted channel (ssh)
- encrypt private key
- key renewal
- worry about trojan horses, keyboard sniffers
- strong protection of KDC and CA’s
- strong protection of backups
- tamperproof hardware
  - token/Fortezza card
  - CA Certificate Signing Unit (CSU)

Where to encrypt?

Risk layer:
- encrypting routers, net board (wireless)
- transparent/fast
- suitable for private net
- provides only one link (pt-to-pt)
- info may be exposed in OS

network/transport layer
- port	p(IPSec)
- transparent
- adaptable (pluggy)
- app level keying
- works over public net
- virtual private network (VPN)
- system layer:
  - encrypting file systems (EFS/CFS)

application layer
- end-to-end over public net
- custom applications (SSL, sip, etc)
- intrusive, but flexible
- API for application development
- key for every logical circuit

Traffic analysis

encrypted traffic threats
- covert channels
- who’s talking to whom
- frequency, event correlation
- quantity, length, patterns of messages
- countermeasures
  - padding messages
  - continuous/random traffic

Attacks & Defenses

- Risk assessment
- Viruses
- Unix security
- authentication
- Network security
- Firewalls, vpn, Ipsec, D5
- MIDTERM

Cryptography
- Random numbers
- Hash functions
- MD5, SHA, RIPEMD
- Classical + stego
- Number theory
- Symmetric key
- DES, Rijndael, RC5
- Public key
- RSA, DSA, D-H-ECC
- Applied crypto
- SSH
- PGP
- S/Mime
- SSL
- Kerberos
- IPsec
- Kerberos
- IPsec
- RSA, DSA, D-H-ECC

Applied crypto
- SSH
- PGP
- S/Mime
- SSL
- Kerberos
- IPsec
- Kerberos
- IPsec
- RSA, DSA, D-H-ECC

Review

Lectures

http://www.cs.utk.edu/~dunigan/cns06/midrw.txt

Next time ...

Midterm
- part 1: take-home (analysis of MISTY1 cipher), work on your own
- see midterm (class 7 html)
- part 2: in class (open book/notes….) next Tuesday, you’ll need your textbook