

CS574

Computer Security

San Diego State University

Spring 2008

Lecture #5

Today's Structure

- Administrivia
- Questions
- Recent News
- Lectures
- Crashers

Administrivia

- Crashing
Last day for add codes before class on monday.
- Roll

Questions?

Recent News

- *Misdirected email*
- *Quantum Computing*
- *RSS*
- *Internet cables*
- *Safer Internet*
- *Diebold keys*
- *New botnet and spam*

Lecture

This Lecture

- block modes
- key management
- asymmetric keys and ciphers
- hard problems
- knapsack, RSA
- using public-key crypto

Block Modes

Block Modes

- ECB - Electronic Code Book
- CBC - Cipher Block Chaining
- CFB - Cipher Feedback
- OFB - Output Feedback

ECB Mode

- Apply encryption directly to data block.
- Very fast.
- Problems:
 - can 'replay' blocks
 - weak for large amounts of patterned data
 - same plaintext results in same ciphertext
- Best if used for short random data, like keys, or random access, like (database) files.

CBC Mode

- Adds a *feedback* mechanism to a block cipher.
- XOR plaintext with previous ciphertext:
$$C_i = E_k (P_i \oplus C_{i-1})$$
$$P_i = C_{i-1} \oplus D_k (C_i)$$
- Requires an *initialization vector* (random, but not secret, *should* be unique, but not required to be).
- Is “self-healing” or “self-recovering” – i.e., ciphertext corruption is self-limiting.
- Considered the best mode for files.

CBC Mode

- Cipher-text error garbles that block and introduces an error in the following block.
- Adversary might be able to take advantage of toggling a known bit.
- Adversary can append gibberish blocks.
- Very long messages will have patterns.

CFB Mode

- Analogous to a “self-synchronizing” stream cipher (what the military calls an ‘autokey’).
- XOR plaintext with encrypted ciphertext
$$C_i = P_i \oplus E_k (C_{i-1})$$
$$P_i = C_i \oplus E_k (C_{i-1})$$
- Requires an initialization vector (which need not be secret, but must be unique).
- Can encrypt entities smaller than the block size (which then are sent and added to queue).

CFB Mode

- A ciphertext error induces error in plaintext, and then garbles all blocks until the feedback buffer / shift register is clear.
- An adversary can toggle bits in a given block to set the desired plaintext result, but the next block will be garbage.
- Considered good for streams of individual characters.

OFB Mode

- Similar to the CFB mode.
- Analogous to a “synchronous stream cipher” (what the military calls a ‘Key Auto-Key’).
- XOR with a shared but independent state:

$$C_i = P_i \oplus S_i; S_i = E_k (S_{i-1})$$

$$P_i = C_i \oplus S_i; S_i = E_k (S_{i-1})$$

S is the state, which is independent of both P and C

OFB Mode

- Keystream is independent of the message stream.
- Garbled ciphertext only affects the text for that block.
- Protects against the insertion/deletion of blocks, but not against toggling of bits.
- Loss of synchronization garbles the output.

OFB Mode

- Requires an initialization vector (which should be unique, but need not be secret).
- Errors are not propagated.
- Keystream generator will be periodic (should not let keystream repeat).
- Considered good for high-speed synchronous systems where error propagation is intolerable.

Other Modes

- Block Chaining Mode
- Propagating Cipher Block Chaining Mode
- Cipher Block Chaining with Checksum
- Output Feedback with Nonlinear Function
- Plaintext Block Chaining
- etc.

A Note On Padding

- Problem: data doesn't always fit into a block.
- Possible solutions:
 - Use delimiters in the plaintext
 - Use fixed pattern
 - Put padding count in last byte
- Some modes can use a block fragment (no need for padding)

Key Management

Key Management

- Symmetric-key cryptosystems need a key for every possible pair of senders/receivers.
- How are keys distributed?
 - Trusted issuer or escrow agency.
 - Courier
 - In person
- Key explosion - graph problem: n choose 2

Key Management

Choosing

- $n \text{ choose } k = n! / (k! (n - k)!)$
- $n \text{ choose } 2 = n! / (2! (n - 2)!)$
 $= n! / (2 (n - 2)!)$
 $= (n (n - 1)) / 2$
 $= (n^2 - n) / 2$
- Number of keys increases as the square of the number of participants – *not practical!*

Key Management

The Solutions

- Diffie-Helman-Merkle Key Exchange
- Asymmetric Keys

Key Exchange

- Diffie-Hellman-Merkle Key Exchange
- Analogies with an Untrustworthy Post Office
- Order Independence
- Hellman's insight: $y^x \bmod p$

Key Exchange

1. alice and bob agree on a **y** and a **p**
2. alice chooses a number **a**, and bob, **b**
3. alice computes **alpha** = $y^a \bmod p$
4. bob computes **beta** = $y^b \bmod p$
5. alice sends bob **alpha**, bob sends alice **beta**
6. alice makes her key = $\mathbf{beta}^a \bmod p = y^{b^a} \bmod p$
7. bob makes his key = $\mathbf{alpha}^b \bmod p = y^{a^b} \bmod p$

Key Exchange

- Prevents “Eve”* from working out the key
- Requires an exchange of information
 - accept a long delay (or)
 - both people have to be “online”
- Does not prevent “Mallory”** from implementing a “Man in the Middle Attack”

* *Eve is our eavesdropper.*

** *Mallory is our malicious attacker who can intercept and modify all messages.*

Asymmetric Keys

- Asymmetric keys – one key is used to encrypt the plaintext, and another is used to decrypt the cipher text.
- Diffie thought it up.
- $C = E(K_e, P)$
 $P = D(K_d, C) \Rightarrow$
 $P = D(K_d, E(K_e, P))$

Asymmetric Keys

- Encrypting key can be shared – made *public*.
- Decrypting key must be kept private.
- Analogy:
Locks are easy to close but hard to open without the appropriate key or combination.

Public Key Cryptography

- Uses asymmetric keys.
- The public key is given away or published.
- The private key is kept secret.
- Built on one-way problems (easy on one direction, hard in the other direction).

Hard Problems

- P
Can be solved in polynomial time.
- NP
Can be solved by a non-deterministic machine in polynomial time. (e.g., oracle + verification)
- NP-hard
All problems in NP can be reduced to an NP-hard problem via some algorithm that runs in polynomial time.
- NP-complete
The problems in NP that are NP-hard.

Useful Modulus Inverses

- $a^{-1} * a \bmod n = 1$ ($a \neq 0$)
- $a^n \bmod n = a$ ($a < n$)
- $a^{(n-1)} \bmod n = 1$
- $a^{-1} = a^{(n-2)} \bmod n$

Public Key Crypto

Example: Knapsack

- “First” algorithm for public-key encryption
- Developed by Merkle and Hellman
- Originally only for encryption (Shamir adapted for digital signatures)
- Knapsack problem: given a pile of (differing) weights and a stack of knapsacks, distribute weights so that all knapsacks weigh the same.

Public Key Crypto

Example: Knapsack

- The Knapsack Problem:
 - general case is NP-complete – i.e., hard
 - “simple” knapsack (aka “superincreasing knapsack”) in P – i.e., easy
- Use two knapsacks, one easy, one hard.
- Easy knapsack – private (decrypting) key
- Hard knapsack – public (encrypting) key

Public Key Crypto

Example: Knapsack

Key Creation

- Create a superincreasing knapsack S
- Choose a modulus n and a multiplier w
 - n must be prime
 - $n > \text{sum}(S)$
 - n and w are relatively prime
- Derive hard knapsack H with:
$$h_i = s_i * w \text{ mod } n$$

Public Key Crypto

Example: Knapsack

Encrypting

- Let m be the number of elements in S
- Let P be the plaintext message
- Divide P into P_0, P_1, P_2, \dots of m bits each
- Treat P_i as the “selection vector”
- $$C = \sum P_i * H = \sum w * S * P \text{ mod } n$$

Public Key Crypto

Example: Knapsack

Decrypting

- Use the inverse of w to get to simple knapsack
- $w^{-1} * C = w^{-1} * H * P \text{ mod } n$
 $= w^{-1} * w * S * P \text{ mod } n$
 $= S * P \text{ mod } n$
- S was our simple knapsack

Public Key Crypto

Example: Knapsack

- In practice
 - $100 < \text{digits}(n) < 200$
 - $1 \leq s_0 \leq 2^{200} ; 2^{200} \leq s_1 \leq 2^{201} ; \dots$
- Cryptanalysis (scheme is broken)
 - if we know n , we might guess the simple knapsack
 - we can make a reasonable guess for n when no hard knapsack element $> n$ (lower bound) and n isn't too much larger than the largest element
 - can then iteratively attack w over a much-reduced range for n .

Public Key Crypto

Example: RSA

- Ron Rivest, Leonard Adleman, and Adi Shamir
- The hard problem is factoring (into large primes)
- $C = P^e \bmod n$
- $P = C^d \bmod n$
 $= P^{e^d} \bmod n$
 $= P^{d^e} \bmod n$

RSA Key Creation

- Choose two (large) primes p and q
- Compute $n = p * q$
- Compute $\phi = (p - 1) * (q - 1)$
- Choose e relatively prime to ϕ
- Compute d such that $e * d \bmod \phi = 1$

```
rsa_key() :  
  p <- generate_random_prime()  
  q <- generate_random_prime()  
  assert( p != q )  
  
  n <- p * q  
  o <- (p - 1) * (q - 1)  
  
  e <- o  
  while ( gcd(o, e) != 1 )  
    e <- generate_random()  
  elihw  
  
  d <- 2 // could also start high and decrement  
  while ( (e * d mod o) != 1 )  
    increment( d )  
  elihw  
  
  answer [ d, e, n ]  
end
```

RSA Cryptanalysis

- Simple ciphertext attack if blocks too small.
- No known successful cryptanalysis if obvious implementation errors avoided.
- “PRIMES in P” apparently no effect
- Quantum Computers and Shor’s Algorithm

Using Public Key Cryptography

- Asymmetric keys very slow compared to symmetric keys (RSA 1000x slower than DES).
- Encrypt a symmetric key with asymmetric key and use a block cipher to exchange keys.
- Encrypt hash (or digest) of the message to verify integrity of contents and to digitally “sign” the message.

PGP

- PGP = Pretty Good Privacy
- Creation of Phil Zimmerman
- Uses RSA for keys, IDEA for encryption
- Alternate implementation from FSF:
GNU Privacy Guard

Lecture References

- Pfleeger, *Security in Computing*, 3rd & 4th edition
- Bishop, *Introduction to Computer Security*
- Schneier, *Applied Cryptography*
- Schneier, *Practical Cryptography*
- Schneier, *The Crypto-Gram Newsletter*, Sep. 15, 2002
- Singh, *The Code Book*
- RSA Laboratories, *RSA Crypto FAQ*
- Gutman, *Lessons Learned in Implementing and Deploying Crypto Software*
- Rothenburg, et al., *The Snake-Oil FAQ*
- *They Cryptography FAQ*

End Lecture

Reading

(For the next few classes)

- Pfleeger, Chapter 3

(Start Reading)

Crashers

- Last class to get add codes.
- Make arrangements with me if you can't get an add code today.

Finis