T-110.5210 Cryptosystems

Some cryptosystems

23.10.2008

Kaufman et al: Chapters 6.8 / 9.7 / 14 / 22

Stallings: Chapters 14.1 / 15.1

Outline

- Some mathematical cryptographic protocols
  - Secret sharing and splitting
  - Bit commitment
  - Zero-knowledge proofs
  - Secure multiparty computation
- Some real-world protocols
  - Kerberos
  - PGP
  - Hard drive encryption
  - SSH2
- Assignment
  - Certificate manager [crypto.implementations.CertificateManager]
  - Remaining TLS “tweaks” in tlscient.HttpsClient (see participant page)
  - **Extra**: MD5, SHA1, and/or DES from scratch [crypto.implementations.{Md5,Sha1,SingleDesEcb}]

Secret sharing and splitting

Secret sharing and splitting

- An example problem
  - You need to control access to your “top secret” database
  - But you can't trust one single person with the key – how to share the key to **n** participants?
- Trivial solution: split k-bit key into n parts of k/n bits each
  - If attacker captures 50% of parts, he obtains 50% of secret
  - If participants are lost, their secret bits are lost too
- What we would want ...
  - Secret cannot be recovered until enough participants come together (willfully or by coercion) and interact to recover it
  - Loss of some participants doesn't prevent the remaining participants (if sufficiently many) from recovering the entire secret
  - Partial recovery should yield no information
  - The scheme should tolerate cheating by (some) participants

Secret sharing and splitting

- Better solution: secret splitting
  - Splitting process (for **k**-bit message **M** and **n** participants)
    - Generate **n-1** **k**-bit random bit strings, $R_1, ..., R_{n-1}$
    - Give these bit strings to **n-1** participants
    - Give the last participant $R_n = M \text{ xor } R_1 \text{ xor } ... \text{ xor } R_{n-1}$
  - Recovery process
    - Obtain all parts $R_1, ..., R_n$
    - Compute $M = R_1 \text{ xor } R_2 \text{ xor } ... \text{ xor } R_n$
- Properties of secret splitting
  - All parts, $R_1, ..., R_n$ are required to recover the secret
  - If one or more parts missing, no information is gained (cf. one-time pad)
  - We're not there quite yet, but it's better (and easy)

Secret sharing and splitting

- Secret sharing using a **(m,n)-threshold scheme**
  - Message divided into **n** shares (shadows)
  - To recover message, any **m** shadows are sufficient
  - With less than **m** shadows, no information is gained
- An example scenario
  - Company CEO and three trusted board members, A, B, and C
  - The CEO and one of the board members should be able to recover the secret
  - Also, the three board members together (without the CEO) should be able to recover the message
- Solution
  - **(3,5)-**threshold scheme, CEO gets 2, board members 1 shadow
- How does such a scheme work in practice?

Secret sharing and splitting

- Shamir's scheme (LaGrange interpolating polynomial)
  - Assume we're trying to construct a **(m,n)**-threshold scheme for a message **M**
  - Pick a large prime, **p**, and compute modulo **p** in a finite field
  - Generate a polynomial with degree **m-1** (=> **m** coefficients), i.e.
    - $F(x) = c_{m-1}x^{m-1} + c_{m-2}x^{m-2} + ... + c_1x + M \pmod p$
    - Constant coefficient is message **M**, others are random
  - Create shadows by evaluating at **n** different points, x_i
    - $S_i = F(x_i)$ (**i**th share)
- Reconstructing the message, **M**
  - Obtain **m** shadows
  - We obtain **m** equations of **m** unknowns in a known finite field
  - Solve all coefficients from the set of equations => constant coefficient, **M**, is the message

Secret sharing and splitting

- Properties of Shamir's scheme
  - **m** and **n** can be picked arbitrarily
  - If less than **m** shadows are known, no information is leaked
    - Scheme is implemented in a finite field
    - When we have **m-1** shadows, the probability distribution for the constant coefficient is still uniform (= no information leak)
  - However, a single cheating participant can destroy message; thus, the basic scheme doesn't tolerate cheaters
- More advanced schemes
  - Can be made to tolerate cheating participants
  - The rules on how many and what type of participants are required to recover the message can be made arbitrarily complex

Bit commitment

Bit commitment

- Problem
  - Let's say Alice can predict lotto numbers and wants to prove this ability to Bob
  - But Alice doesn't want to reveal next week's lotto numbers to Bob in advance (Bob would be sure to cash in without paying Alice)
  - (Alice would probably cash in herself, but let's ignore that in this example :-)
  - => Alice can use a **bit commitment protocol**
- Solution
  - Alice interacts with Bob to send information committing herself to a particular bit (bit sequence in practice; here just one bit)
  - Later (e.g. after lotto numbers have been drawn) Alice sends some information to Bob which
    - (1) allows Bob to "open" the earlier sequence, while
    - (2) not allowing Alice to change her mind
- How could this work in practice?

Bit commitment using symmetric crypto

- Phase 1 – commitment
  - Bob => Alice Random bit-string **R** gen. by Bob (eg 128 bits)
  - Alice => Bob $\mathbf{X} = \mathbf{E_K(R \| b)}$, where
    - **b** = Alice's bit of information
    - **K** = A random encryption key gen. by Alice
- Phase 2 – revealing the bit
  - Alice => Bob **K** (the random key)
  - Bob decrypts $\mathbf{D_K(X)} = \mathbf{(R \| b)}$ and
    - 1) Verifies that **R** is correct
    - 2) Looks up **b** and determines that Alice's information was good
- Why does this work?
  - Without **K**, Bob gains no useful information (needs to break encryption)
  - In phase 2, Alice cannot easily brute force a key, **K**, which simultaneously decrypts **X** to **b** *and* **R** (**R** long, eg 128 bits)
  - Note that the redundancy, **R**, is selected by Bob

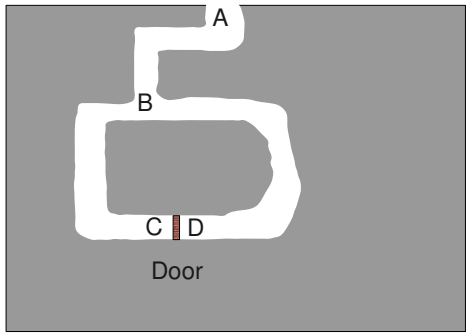
Zero-knowledge proofs

Zero-knowledge proofs

- Problem
  - Alice wants to prove to Bob that she knows something valuable
  - But Alice doesn't want to reveal *any* useful information to Bob
- Solution
  - Alice uses a zero-knowledge proof
- Properties of a zero-knowledge proof (probabilistic)
  - Completeness – if Alice and Bob are honest, Bob will become convinced
  - Soundness – if Alice is not honest, Bob will not be convinced
  - Zero-knowledge – Bob learns nothing from Alice that he couldn't learn without Alice, apart from the fact that Alice knows the claimed thing
- Shown by describing a simulator of the process
  - We show that Bob can create a fake transcript of a zero-knowledge proof which cannot be distinguished from a real one
  - Hence a third party will not be convinced by any zero-knowledge transcript, real or fake
  - (The formalization is of course more complicated than this)

Zero knowledge proof – toy example

- Alice claims to know the code for door between C and D
  - The proof is interactive, consisting of 5 repeated steps
  - After **n** repeats, Alice's probability of cheating successfully is $\mathbf{2^{-n}}$
  - Bob cannot demonstrate the proof to Carol – Bob could have colluded with Alice, as far as Carol knows
- Thus, the proof is zero knowledge



- **1) Bob stands at point A**
- **2) Alice walks all the way into the cave (randomly from either side) to C or D**
- **3) Bob walks to point B**
- **4) Bob shouts to Alice, asking Alice to come out of either the left or the right passage (randomly)**
- **5) Alice complies, opening the door if necessary (i.e. if she went in using the other passage)**
- **6) Alice and Bob repeat steps 1–5 n times**

Secure multiparty computation

Secure multiparty computation

- Secure multiparty computation
  - Yao's Millionaire Problem (1982): two millionaires want to determine who is richer without revealing how much $$$ they have
  - In general, two or more participants wish to computation some function **F** of data known only to individual participants => Only the result of **F** should be revealed
    - Each participant knows only their own input, and the shared output
  - Cheaters should be tolerated or caught
  - Example: We want to compute an average of salaries
- Specific solutions to specific problems
  - Secure voting, statistics, etc.
- In fact, any function F can be computed under certain assumptions
  - Most participants are honest
  - Specific definition of security
  - Efficiency not very good in general, specific algorithms are still interesting

But that's not nearly all ...

- There is a wide variety of mathematical crypto protocols
  - Oblivious transfer
  - Blind signatures
  - Fail-stop signatures
  - Quantum cryptography
  - ...
- And much development is going on
  - Cryptography is still a very active field

Kerberos

Kerberos

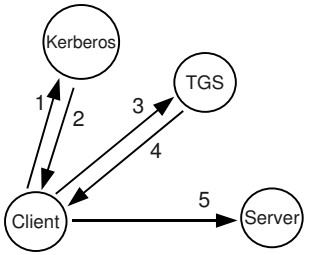
- Kerberos (V5) in short
  - Trusted third-party authentication protocol based on symmetric crypto
- Secure communication between client and server
  - 1) Server needs to know client has been authenticated properly
  - 2) We need to encrypt and integrity protect network traffic
- Specification
  - RFC 1510
- Kerberos **ticket –** an end result of authentication, with
  - Identifiers of client and server, client's network address
  - A validity period
  - A session key to be used between the client and the server, $\mathbf{K}_{c,s}$

Kerberos entities

- Kerberos server
  - Has secret keys of all clients, servers, and TGSs, denoted $\mathbf{K}_x$
  - Hands out Ticket-Granting Tickets (TGTs) for clients
- Ticket-Granting Server (TGS)
  - Also has a database of secret keys of (at least) all servers
  - Authenticates clients (using TGTs) and generates session keys for client-server communication
- Server for some service, e.g. network file sharing
  - Authenticates client using a Server Ticket, and takes client-server session key into use in providing its service
- Client - wants to communicate with a particular server
  - Gets a TGT from Kerberos server, then a server ticket from TGS
  - Authenticates to the server using the server ticket, and establishes session encryption

Kerberos protocol in brief

- **$\{ X \}K_a$**
  - Plaintext **X** encrypted using symmetric key K_a
- **$\{T_{c,s}\}K_s = \{s, c, a, v, K_{c,s}\}K_s$**
  - Ticket between client **c** and server **s**
  - **s**=server, **c**=client, **a**=client address, **v**=validity, $K_{c,s}$=session key
- **$\{A_{c,s}\}K_{c,s} = \{c, t, key\}K_{c,s}$**
  - Authenticator for client **c** and server **s**
  - **c**=client, **t**=timestamp, **key**=optional additional session key



1. Request for Ticket-Granting Ticket	**c, tgs**
2. Ticket-Granting Ticket	$\{K_{c,tgs}\}K_c$, $\{T_{c,tgs}\}K_{tgs}$
3. Request for Server Ticket	$\{A_{c,tgs}\}K_{c,tgs}$, $\{T_{c,tgs}\}K_{tgs}$
4. Server Ticket	$\{K_{c,s}\}K_{c,tgs}$, $\{T_{c,s}\}K_s$
5. Request for Service	$\{A_{c,s}\}K_{c,s}$, $\{T_{c,s}\}K_s$
(6. Encrypted communication)	**{ (traffic) }**$K_{c,s}$

PGP

PGP

- Background
  - Version 1.0 of PGP released by Philip Zimmermann in 1991
  - PGP uses public key cryptography to secure files and other data
  - Digital signatures, encryption, compression, radix-64 conversion
- Specifications
  - RFC 2440: OpenPGP message format (PGP 5.x / PGP 3)
  - RFC 1991: PGP 2.6 / 2.7
- PGP is a hybrid cryptosystem
  - Bulk data encryption always uses a symmetric algorithm
  - “Session key” is encrypted using asymmetric cryptography

PGP operations (in order)

- Digital signatures
  - Sender computes a hash of the message, and signs the hash with his private key; the signature is attached to message
- Compression
  - Ordinary compression algorithms, applied after digital signature
- Encryption
  - Alt 1: generate a random session key, **K**, encrypt bulk data with **K**, and attach **K** encrypted with recipients' public keys
  - Alt 2: generate a random session key, **K**, encrypt bulk data with **K**, and attach **K** encrypted with another symmetric key, **L**, derived from a passphrase
  - Alt 3: encrypt bulk data with symmetric key, **L**, derived from a passphrase
- Radix-64 conversion (optional)
  - Convert message to printable ASCII chars, 6 bits / character

PGP binary syntax

- A message is a sequence of *packets*
  - Packet header – variable length
    - Content type
    - Content length
    - New (PGP 5.x) / old (PGP 2.x) packet format – “boring details”
  - Packet content
    - Binary content, semantics depend on content type
- PGP has lived over a decade, so the format has some baggage

Dump of a PGP-signed file

```
$ gpg --openpgp --sign lect1.pdf
...
$ gpg --list-packets lect1.pdf.gpg
:compressed packet: algo=1
:onepass_sig packet: keyid 2D052FBB6DD926A5
        version 3, sigclass 00, digest 2, pubkey 1, last=1
:literal data packet:
        mode b (62), created 1129675872, name="lect1.pdf",
        raw data: 699557 bytes
:signature packet: algo 1, keyid 2D052FBB6DD926A5
        version 3, created 1129675872, md5len 5, sigclass 00
        digest algo 2, begin of digest 88 da
        data: [2048 bits]
```


Hard drive encryption

Typical requirements

- Encrypt disk blocks at the block level, not filesystem
- Password protection – password required to access data
- Random access for read and write (at least on a per-block or per-block-group basis)
- Multiple passwords and key escrow (e.g. in a large corporation)
- Amount of usable data per disk block should not decrease (no per-block overhead)
- Withstand attacks if hard drive is captured

Implications of requirements

- Encryption mode
  - Use counter mode (counter = offset on disk), or other stream cipher where stream can be generated starting from a point in the middle of the stream
  - Use CBC (or similar), but compute a per-block IV algorithmically, as there is no space for a stored per-block IV
  - However, each mode has some security issues (we'll cover a few examples)
- Encryption key
  - User password should not be used directly, but a random session key should be encrypted using (hashed) user password or using public key encryption (cf. PGP)
  - For multiple decryption passwords and key escrow, multiple copies of the password should be stored (=> reserve header blocks)

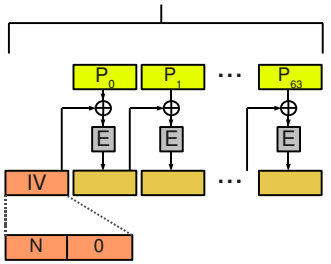
Implications of requirements

- Disk block format
  - We can't comfortably have per-block overhead, so there cannot be an explicit IV, explicit counter, etc.
  - Difficult to do integrity protection (MAC) of data efficiently
- Escrow
  - We need a more comprehensive key management infrastructure
  - For instance, store escrow key using public key encryption to avoid revealing escrow password
  - The escrow private key could benefit from secret sharing

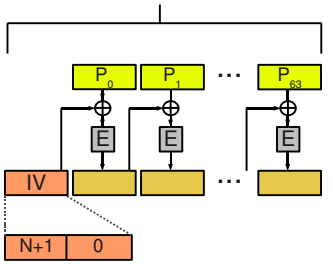
Example: **(old)** Linux 2.6.8 cryptoloop

- Basic parameters
  - Both ECB and CBC mode, we depict CBC mode here
  - Sector size 512 bytes (1 << 9 bytes)
  - CBC IV = sector number (padded with zero), little-endian
  - Encryption key is hashed from password
    - E.g. AES-128 => SHA-256, 3DES => RIPEMD-160, etc.

Encryption in CBC mode for sector N



Encryption in CBC mode for sector N+1



Some counter mode issues

- Counter mode acts as a fixed XOR mask for the entire disk
  - The XOR mask remains constant while data changes
  - In essence, when data changes, we're reusing the same keystream to encrypt different data
- = Problem if attacker can access disk multiple times
  - Attacker gets access to disk and copies it at time X to D1
  - Time passes, data changes
  - Attacker gets access to disk and copies it at time Y to D2
  - C1 xor C2 = (P1 xor K) xor (P2 xor K) = P1 xor P2
    - P1/C1 = plain/ciphertext bit 1, P2/C2 = plain/ciphertext bit 2, K = keystream bit
- Implications
  - Attacker gets to know which bits have been flipped
  - If attacker can convince victim to copy known data to the disk and can predict where it will go (offset), P2 is known => get P1
  - Many other subtle implications
- CTR is good if disk stolen (once) but not if "eavesdropped"

Some CBC mode issues

- If initial CBC computation is deterministic, we may have trouble
  - Assumes that attacker can craft attack files which exploit the deterministic IV
  - The exploits are usually not obvious, but can be damaging
- Let's assume that the implicit CBC IV is a (padded) counter
  - Successive IVs have very small bit differences (32-bit counter example)
    - 0x0000000E = 00000000 00000000 00000000 00001110
    - 0x0000000F = 00000000 00000000 00000000 00001111
    - 0x00000010 = 00000000 00000000 00000000 00010000
    - 0x00000011 = 00000000 00000000 00000000 00010001
  - 50% of the time only the lowest bit changes when we move from one block to the next (= when lowest bit changes from 0 to 1 without carry)
- Let's see a "file watermark attack" example
  - Attacker wants to know whether a specific file is stored on victim's hard disk
  - E.g. secret file released under a non-disclosure agreement on a competitor's employee's laptop with full disk encryption
  - We need to know: cipher block size, cbc-block-size (= at which intervals is the CBC IV implicitly reset, e.g. every disk block)
  - Here: 32-bit cipher block (fictitious), 512 bytes block size (typical)

CBC attack – "watermark"

- Creating a "marker" into an attack file
  - File at offset N contains e.g. 0x00000000 (of cipher block size)
  - File at offset N + 512 contains e.g. 0x00000001 (of cipher block size)
  - In other parts can store any data
- When IV is XORred, 50% chance that the result will be same:
  - IV1 = `XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXX0`
  - DATA1 = `00000000 00000000 00000000 00000000`
  - **XORred =** **`XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXX0`**
  - IV2 = `XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXX1`
  - DATA2 = `00000000 00000000 00000000 00000001`
  - **XORred =** **`XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXX0`**
  - (Note: XXXX... is unknown but same in both cases)
  - **Result => cipher blocks will be equal after encryption**
- To make this reliable, need to repeat this at two offsets
  - Works only if lowest bit of IV changes 0->1; in 1->0 the carry is unpredictable

CBC attack – "watermark"

- What we have now is a primitive to encode a detectable one-bit mark in a certain part of a file
  - Scan the hard disk for ciphertext block collisions; if same ciphertext block occurs with 512 byte offset, we have a marker
  - Otherwise, with overwhelming probability, ciphertext blocks do not match and we don't have a marker
- File "watermark" is encoded by using multiple one-bit marks
  - For instance, generate a unique 128-bit ID for a file
  - Encode each bit by distance in file between marker bits
  - E.g. 0-bit = 10 blocks between markers, 1-bit = 20 blocks between markers
- Detecting the watermark
  - Scan the disk for "markers" (ciphertext collisions), interpret pattern
  - Need to deal with file fragmentation and other disturbances => embed markers multiple times throughout the file, employ error correcting codes, etc.
- Detection (from a victim point of view)
  - Ciphertext collisions are a giveaway – they should be quite unlikely normally

CBC attack – "watermark"

- What to do about this problem?
  - Predictable IVs are a problem if the attacker can, in any situation, "craft" data which will be encrypted
  - Best is to avoid predictable IVs altogether, either by:
    - (1) using cryptographically random, explicit IVs (not good for HD encryption)
    - (2) using unpredictable (computationally infeasible) pseudorandom, implitic IV
- General guidance for CBC
  - To be safe side, **avoid predictable IVs whenever possible**
- Small side note
  - There are theoretical attacks similar to this one for IPsec and TLS if predictable IVs are used (adaptive chosen plaintext attacks); very difficult to exploit
  - TLS 1.0 uses implicit IVs, so they are always predictable
  - IPsec uses explicit (per packet) IVs, so the fix is to choose them in an unpredictable manner

SSH2

SSH2 overview

- Background
  - SSH1 from personal project into commercial software
  - SSH2 standardization in the IETF now finished
  - SSH2 has some functional overlap with TLS
- SSH2 protocol model
  - Typically runs over a TCP connection
  - Provides mutual authentication, confidentiality, integrity, compression
  - Provides “logical channels” multiplexed over a secure connection to upper layers
- SSH2 specifications (main documents only)
  - Architecture: RFC 4251
  - Transport: RFC 4253
  - Authentication: RFC 4252
  - Connection protocol: RFC 4254

SSH2 protocol layers

- Lowest layer – transport protocol
  - E.g. TCP
- SSH2 Transport Layer Protocol
  - Server auth, confidentiality, integrity, compression (opt.)
- SSH2 User Authentication Protocol
  - Client authentication
  - Runs over the Transport Layer Protocol
- SSH2 Connection Protocol
  - Multiplexes encryption connection to multiple logical channels
  - Runs over Transport Layer Protocol, requires User Auth. Protocol
  - Lots of functionality: port/X11 forwarding, signaling, environment variables, shell command execution, window dimensions, etc.
- Separate specifications for each layer + architecture

SSH2 Transport Layer Protocol sequence

- 1. Client initiates, sends an ident. string; server responds
- 2. Switch to binary packet protocol; packets consists of
  - 32-bit packet length + 8-bit padding length
  - N bytes of payload (starting with 8-bit payload type)
  - M bytes of padding + packet MAC
- 3. SSH_MSG_KEXINIT packets sent by both parties
  - Cookie values, key exchange algs, server host key algorithms
  - Encryption, integrity protection, and compression algorithms
- 4. Key exchange => K (shared secret), H (exchange hash)
  - E.g. authenticated Diffie-Hellman using a 1024-bit group
  - Signature using server host key
- 5. SSH_MSG_NEWKEYS – take keys & algs into use
  - Computed using K and H

SSH2 User Authentication Protocol sequence

- SSH_MSG_USERAUTH_REQUEST
  - Username, service name, authentication method name (or none)
  - + method specific data
  - Client keeps on sending until no more methods, or gets success
- SSH_MSG_USERAUTH_FAILURE
  - A list of authentication methods that client should still try
  - Partial success flag = last meth. OK, but need more to complete
- SSH_MSG_USERAUTH_SUCCESS
  - Authentication completed, service traffic may begin
- SSH_MSG_USERAUTH_BANNER
  - Banner msg to be shown to end user, can be sent at any time
- Authentication methods
  - None, Public key, Password, Host based (extensible, of course)

SSH2 vs. TLS

- Authentication
  - Client authentication is richer in SSH2
    - Truly interactive, use of banners, etc.
  - TLS client authentication is either done by the upper layer (out of TLS scope, e.g. HTTP authentication) or is certificate-based
- Service provided to upper layers
  - SSH2 provides multiple logical channels with rich functionality
  - TLS provides one application data channel (= one logical channel only)

Summary

- Some mathematical cryptographic protocols
  - Secret sharing and splitting
  - Bit commitment
  - Zero-knowledge proofs
  - Secure multiparty computation
- Some real-world protocols
  - Kerberos, PGP, hard drive encryption, SSH2

- Assignment
  - Certificate manager [crypto.implementations.CertificateManager]
  - Remaining TLS “tweaks” in tlscient.HttpsClient (see participant page)
  - **Extra**: MD5, SHA1, and/or DES from scratch [crypto.implementations.{Md5,Sha1,SingleDesEcb}]