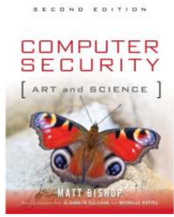
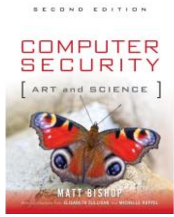


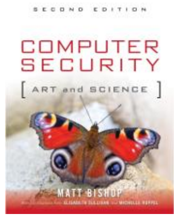
Cipher Techniques

Chapter 12



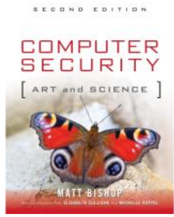
Overview

- Problems
  - What can go wrong if you naively use ciphers
- Cipher types
  - Stream or block ciphers?
- Networks
  - Link vs end-to-end use
- Examples
  - Privacy-Enhanced Electronic Mail (PEM)
  - Secure Socket Layer (SSL)
  - Security at the Network Layer (IPsec)



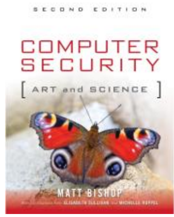
Problems

- Using cipher requires knowledge of environment, and threats in the environment, in which cipher will be used
  - Is the set of possible messages small?
  - Can an active wiretapper rearrange or change parts of the message?
  - Do the messages exhibit regularities that remain after encipherment?
  - Can the components of the message be misinterpreted?



Attack #1: Precomputation

- Set of possible messages *M* small
- Public key cipher *f* used
- Idea: precompute set of possible ciphertexts *f*(*M*), build table (*m*, *f*(*m*))
- When ciphertext *f*(*m*) appears, use table to find *m*
- Also called *forward searches*

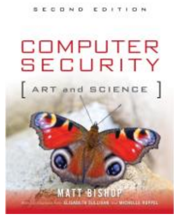


Example

- Cathy knows Alice will send Bob one of two messages: enciphered BUY, or enciphered SELL
- Using public key e_{Bob}, Cathy precomputes

$$m_1 = \{ \text{ BUY } \} \, e_{Bob}, \; m_2 = \{ \text{ SELL } \} \, e_{Bob}$$

- Cathy sees Alice send Bob m_2
- Cathy knows Alice sent SELL

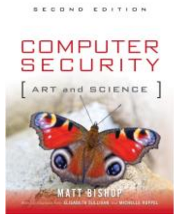


May Not Be Obvious

- Digitized sound
    - Seems like far too many possible plaintexts, aa initial calculations suggest 2^{32} such plaintexts
    - Analysis of redundancy in human speech reduced this to about 100,000 ($\approx 2^{17}$), small enough for precomputation attacks

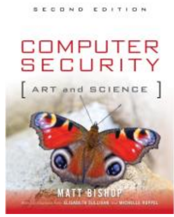
Misordered Blocks

- Alice sends Bob message
  - $n_{Bob} = 262631$, $e_{Bob} = 45539$, $d_{Bob} = 235457$
- Message is TOMNOTANN (191412 131419 001313)
- Enciphered message is 193459 029062 081227
- Eve intercepts it, rearranges blocks
  - Now enciphered message is 081227 029062 193459
- Bob gets enciphered message, deciphers it
  - He sees ANNNOTTOM, opposite of what Alice sent



Solution

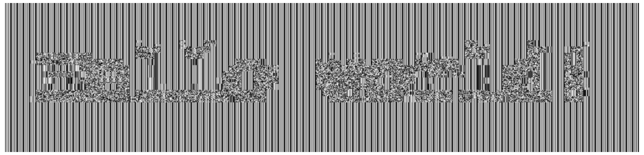
- Digitally signing each block won't stop this attack
- Two approaches:
  - Cryptographically hash the *entire* message and sign it
  - Place sequence numbers in each block of message, so recipient can tell intended order; then sign each block



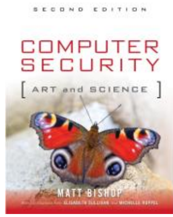
Statistical Regularities

- If plaintext repeats, ciphertext may too
- Example using AES-128:
  - Input image:

  - corresponding output image:



  - Note you can still make out the words
- Fix: cascade blocks together (chaining) More details later



Type Flaw Attacks

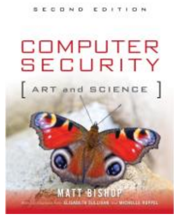
- Assume components of messages in protocol have particular meaning
- Example: Otway-Rees:

Alice → Bob: $n \,||\, \text{Alice} \,||\, \text{Bob} \,||\, \{ r_1 \,||\, n \,||\, \text{Alice} \,||\, \text{Bob} \}\, k_A$

Cathy ← Bob: $n \,||\, \text{Alice} \,||\, \text{Bob} \,||\, \{ r_1 \,||\, n \,||\, \text{Alice} \,||\, \text{Bob} \}\, k_A \,||\, \{ r_2 \,||\, n \,||\, \text{Alice} \,||\, \text{Bob} \}\, k_B$

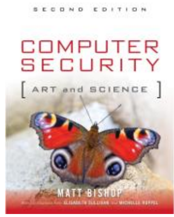
Cathy → Bob: $n \,||\, \{ r_1 \,||\, k_s \}\, k_A \,||\, \{ r_2 \,||\, k_s \}\, k_B$

Alice ← Bob: $n \,||\, \{ r_1 \,||\, k_s \}\, k_A$



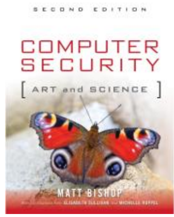
The Attack

- Ichabod intercepts message from Bob to Cathy in step 2
- Ichabod *replays* this message, sending it to Bob
  - Slight modification: he deletes the cleartext names
- Bob *expects* $n \mid\mid \{ r_1 \mid\mid k_s \} k_A \mid\mid \{ r_2 \mid\mid k_s \} k_B$
- Bob *gets* $n \mid\mid \{ r_1 \mid\mid n \mid\mid \text{Alice} \mid\mid \text{Bob} \} k_A \mid\mid \{ r_2 \mid\mid n \mid\mid \text{Alice} \mid\mid \text{Bob} \} k_B$
- So Bob sees $n \mid\mid \text{Alice} \mid\mid \text{Bob}$ as the session key — and Ichabod knows this
- When Alice gets her part, she makes the same assumption
- Now Ichabod can read their encrypted traffic



Solution

- Tag components of cryptographic messages with information about what the component is
  - But the tags themselves may be confused with data …

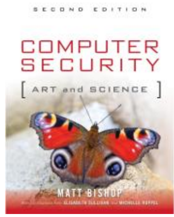


What These Mean

- Use of strong cryptosystems, well-chosen (or random) keys not enough to be secure
- Other factors:
  - Protocols directing use of cryptosystems
  - Ancillary information added by protocols
  - Implementation (not discussed here)
  - Maintenance and operation (not discussed here)

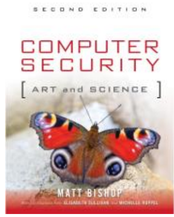
Stream, Block Ciphers

- E encipherment function
  - $E_k(b)$ encipherment of message b with key k
  - In what follows, $m = b_1 b_2 \ldots$, each b_i of fixed length
- Block cipher
  - $E_k(m) = E_k(b_1)E_k(b_2) \ldots$
- Stream cipher
  - $k = k_1 k_2 \ldots$
  - $E_k(m) = E_{k1}(b_1)E_{k2}(b_2) \ldots$
  - If $k_1 k_2 \ldots$ repeats itself, cipher is *periodic* and the kength of its period is one cycle of $k_1 k_2 \ldots$



Example

- AES-128
  - b_i = 128 bits, k = 128 bits
  - Each b_i enciphered separately using k
  - Block cipher



Stream Ciphers

- Often (try to) implement one-time pad by xor’ing each bit of key with one bit of message
  - Example:

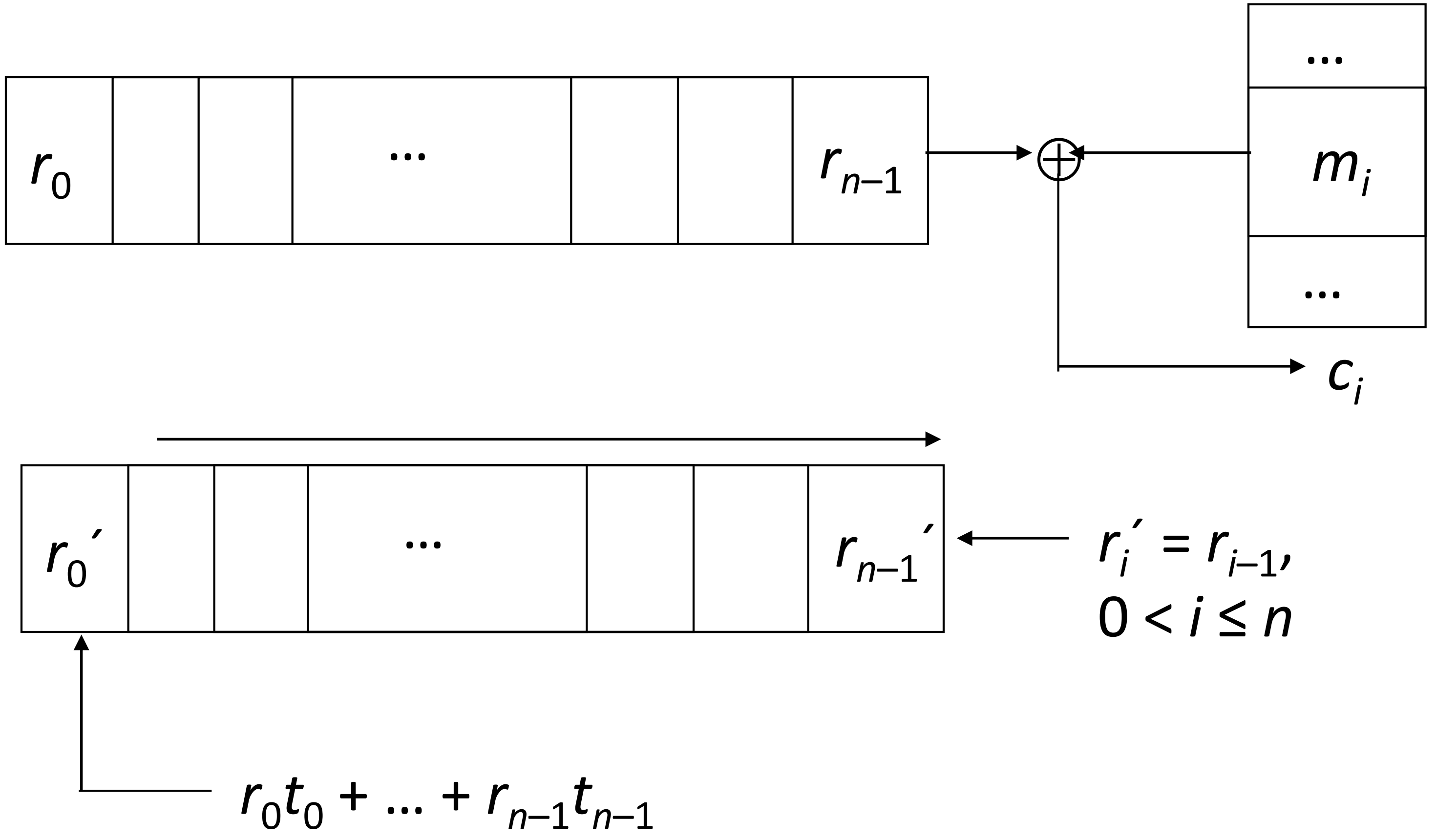
$$m = 00101$$
$$k = 10010$$
$$c = 10111$$

- But how to generate a good key?

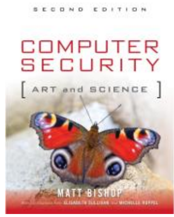
Synchronous Stream Ciphers

- n-stage Linear Feedback Shift Register: consists of
  - n bit register $r = r_0 \ldots r_{n-1}$
  - n bit tap sequence $t = t_0 \ldots t_{n-1}$
  - Use:
    - Use r_{n-1} as key bit
    - Compute $x = r_0 t_0 \oplus \ldots \oplus r_{n-1} t_{n-1}$
    - Shift r one bit to right, dropping r_{n-1}, x becomes r_0

Operation



r_i *i*th bit of register
t_i *i*th bit of tap sequence
m_i *i*th bit of message
c_i *i*th bit of ciphertext

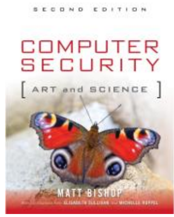


Example

- 4-stage LFSR; t = 1001

r	k_i	*new bit computation*	*new r*
0010	0	01⊕00⊕10⊕01 = 0	0001
0001	1	01⊕00⊕00⊕11 = 1	1000
1000	0	11⊕00⊕00⊕01 = 1	1100
1100	0	11⊕10⊕00⊕01 = 1	1110
1110	0	11⊕10⊕10⊕01 = 1	1111
1111	1	11⊕10⊕10⊕11 = 0	0111
1110	0	11⊕10⊕10⊕11 = 1	1011

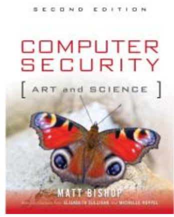
  - Key sequence has period of 15 (010001111010110)



NLFSR

- n-stage Non-Linear Feedback Shift Register: consists of
  - n bit register $r = r_0 \ldots r_{n-1}$
  - Use r_{n-1} as key bit
  - Compute $x = f(r_0, \ldots, r_{n-1})$; f is any function
  - Shift r one bit to right, dropping r_{n-1}, x becomes r_0

  Note same operation as LFSR but more general bit replacement function

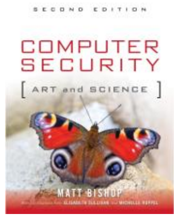


Example

- 4-stage NLFSR; $f(r_0, r_1, r_2, r_3) = (r_0 \& r_2) \mid r_3$

r	k_i	*new bit computation*	*new r*
1100	0	(1 & 0) | 0 = 0	0110
0110	0	(0 & 1) | 0 = 0	0011
0011	1	(0 & 1) | 1 = 1	1001
1001	1	(1 & 0) | 1 = 1	1100
1100	0	(1 & 0) | 0 = 0	0110
0110	0	(0 & 1) | 0 = 0	0011
0011	1	(0 & 1) | 1 = 1	1001

  - Key sequence has period of 4 (`0011`)

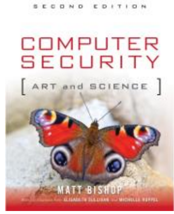


Eliminating Linearity

- NLFSRs not common
  - No body of theory about how to design them to have long period
- Alternate approach: *output feedback mode*
  - For E encipherment function, k key, r register:
    - Compute $r' = E_k(r)$; key bit is rightmost bit of r'
    - Set r to r' and iterate, repeatedly enciphering register and extracting key bits, until message enciphered
  - Variant: use a counter that is incremented for each encipherment rather than a register
    - Take rightmost bit of $E_k(i)$, where i is number of encipherment

Self-Synchronous Stream Cipher

- Take key from message itself (*autokey*)
- Example: Vigenère, key drawn from plaintext
  - *key* XTHEBOYHASTHEBA
  - *plaintext* THEBOYHASTHEBAG
  - *ciphertext* QALFPNFHSLALFCT
- Problem:
  - Statistical regularities in plaintext show in key
  - Once you get any part of the message, you can decipher more

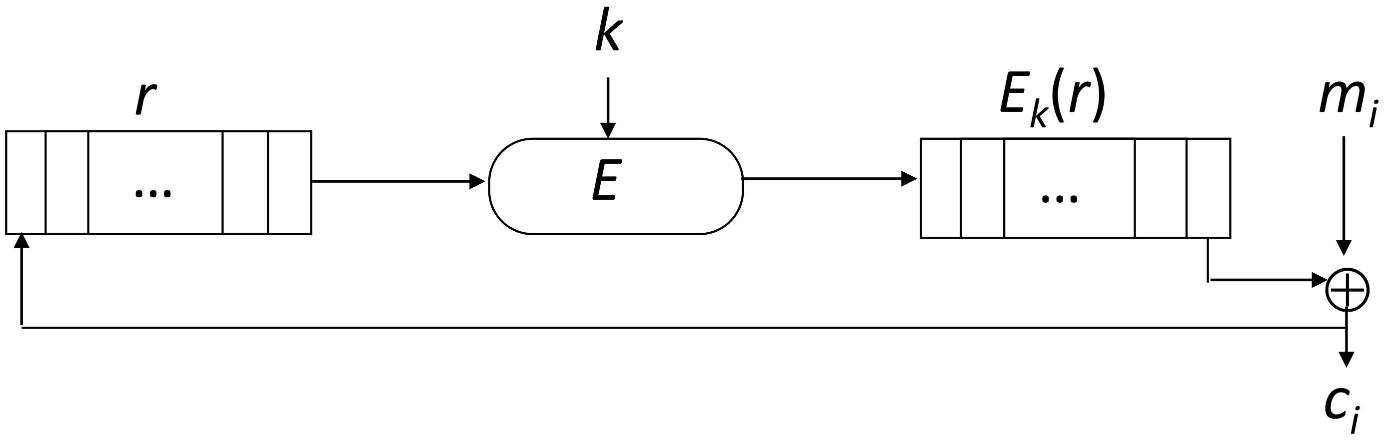


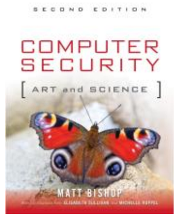
Another Example

- Take key from ciphertext (*autokey*)
- Example: Vigenère, key drawn from ciphertext
  - *key* `XQXBCQOVVNGNRTT`
  - *plaintext* `THEBOYHASTHEBAG`
  - *ciphertext* `QXBCQOVVNGNRTTM`
- Problem:
  - Attacker gets key along with ciphertext, so deciphering is trivial

Variant

- Cipher feedback mode: 1 bit of ciphertext fed into *n* bit register
  - Self-healing property: if ciphertext bit received incorrectly, it and next *n* bits decipher incorrectly; but after that, the ciphertext bits decipher correctly
  - Need to know *k*, *E* to decipher ciphertext

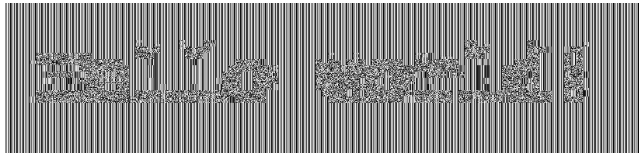




Block Ciphers

- Encipher, decipher multiple bits at once
- Each block enciphered independently
- Problem: identical plaintext blocks produce identical ciphertext blocks
- Plaintext image:

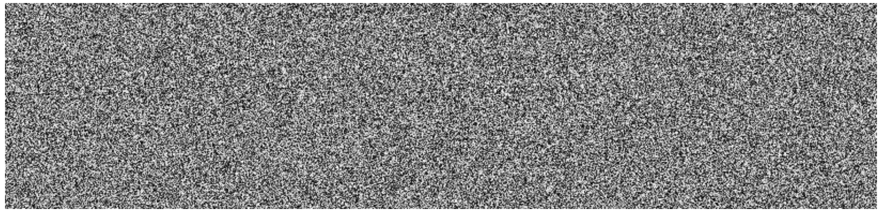
- Ciphertext image:

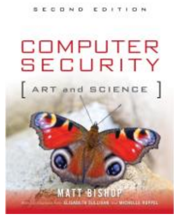


Solutions

- Insert information about block's position into the plaintext block, then encipher
- *Cipher block chaining*:
  - Exclusive-or current plaintext block with previous ciphertext block:
    - $c_0 = E_k(m_0 \oplus I)$
    - $c_i = E_k(m_i \oplus c_{i-1})$ for i > 0

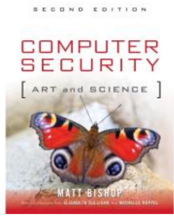
    where I is the initialization vector
- Example encipherment of image on previous slide:





Multiple Encryption

- Double encipherment: $c = E_{k'}(E_k(m))$
  - Effective key length is $2n$, if k, k' are length n
  - Problem: breaking it requires 2^{n+1} encryptions, not 2^{2n} encryptions
- Triple encipherment:
  - EDE (Encrypt-Decrypt-Encrypt) mode: $c = E_k(D_{k'}(E_k(m)))$
    - Problem: chosen plaintext attack takes $O(2^n)$ time using 2^n ciphertexts
  - Triple encryption mode: $c = E_k(E_{k'}(E_{k''}(m)))$
    - Best attack (p chosen plaintexts) requires $O(2^{n+1}p + 2^{h+b+1}/p)$ time, $O(2^n/p)$ memory



Authenticated Encryption

- Transforms message providing confidentiality, integrity, authentication simultaneously
- May be associated data that is not to be encrypted
    - Called Authenticated Encryption with Associated Data (AEAD)
- Two examples:
    - Counter with CBC-MAC (CCM)
    - Galois Counter Mode (GCM)
- *message* is part to be encrypted; *associated data* is part not to be encrypted
    - Both are authenticated and integrity-checked; if omitted, treat as having length 0

Counter with CBC-MAC Mode (CCM)

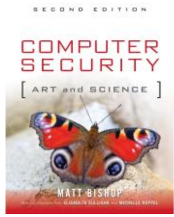
- Defined for block ciphers with block size 1287 (like AES)
- Parameters:
  - L_A size of authentication field (may be 4,6,8,10,12,14,16 octets)
  - L_M size of message length (may take up between 2 and 8 octets)
  - nonce of 15 – L_M octets
- Notation: *k* key, *n* nonce, *M* message, *A* associated data
- Three phases

CCM Phase 1

- Compute authentication field T
- Prepend set of blocks B_i to message; first block B_0 has message info:
  - Octet 0 has flags
    - Bits 0-2: $L_M - 1$
    - Bits 3-5: $(L_A - 2) / 2$
    - Bit 6: 1 if there is associated data, 0 otherwise
    - Bit 7: reserved, set to 0
  - Octets 1 . . . 15 – L_M: nonce
  - Octets 16 – L_M . . . 15: length of message in octets

CCM Phase 1

- Next octets contain information about length L_A:
  - $0 < L_A < 2^{16} - 2^8$: first 2 octets contain L_A
  - $2^{16} - 2^8 \le L_A < 2^{32}$: first 2 octets `0xff`, `0xfe`, next 4 octets contain L_A
  - $2^{32} \le L_A < 2^{64}$: first 2 octets both 0xff, next 6 octets contain L_A
- Block B_0, these octets prepended to associated data A; split this into 16-octet blocks, with 0 padding if needed
- Append message, split into 16-octet blocks, with 0 padding if needed
  - This gives $B_0 \ldots B_m$



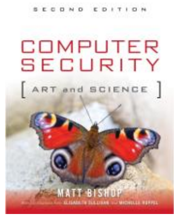
CCM Phase 1

- Compute CBC-MAC of $B_0 \ldots B_m$

$$x_1 = E_k(B_0)$$

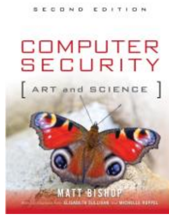
$$x_{i+1} = E_k(x_i \oplus B_i) \text{ for } i = 1, \ldots, m$$

- Authentication field T is first L_A blocks of x_{m+1}



CCM Phase 2

- This enciphers the message using counter mode
- A_i block with the following:
  - Octet 0 contains flags
    - Bits 0-2: contains $L_M - 1$
    - Bits 3-7: set to 0
  - Octets 1 . . . 15 – L_M: contain nonce
  - Octets 16 – L_M . . . 15: contain *i*th counter's value
- Key blocks $S_i = E_k(A_i)$



CCM Phases 2 and 3

Phase 2:

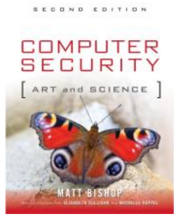
- Encrypt message with blocks $M_1 \dots M_z$: for $i = 1, \dots, z$, $C_i = M_i \oplus S_i$
- Let s_A be first L_A bytes of S0
- Compute authentication value $U = T \oplus s_A$

Phase 3:

- Sender constructs $C = C_1 \dots C_z$ and sends $C \mid\mid U$

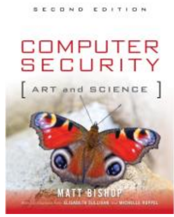
CCM Decryption

- Decryption and validation: simply reverse process
- Important requirement: if validation fails, recipient must *only* reveal that computed *T* is incorrect
    - Must *not* reveal the incorrect value, or any part of decrypted message



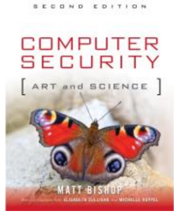
Galois Counter Mode (GCM)

- Can be implemented efficiently in hardware
- If encrypted, authenticated message is changed, new authentication value can be computed with cost proportional to number of changed bits
- Allows nonce (initialization vector) of any length
- Parameters
  - nonce *IV* up to 2^{64} bits; 96 bits recommended for efficiency reasons
  - message *M* up to $2^{39} – 2^{8}$ bits long; ciphertext C same length
  - associated data A up to 2^{64} bits long



GCM Notation

- Authentication value T is t bits long
- $M = M_0 \ldots M_n$, each block 128 bits long
  - M_n may not be complete block; call its length u bits
- $C = C_0 \ldots C_n$, each block 128 bits long; C is L_C bits long
  - Number of bits in C is the same as number of bits in M
- $A = A_0 \ldots A_m$, each block 128 bits long; A is L_A bits long
  - A_m may not be complete block; call its length v bits
- 0^x, 1^y mean x bits of 0 and y bits of 1, respectively



Multiplication in GF(2^{128})

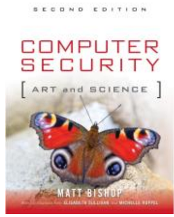
```
/* multiply X and Y to produce Z in GF (2128 ) */
function GFmultiply(X, Y: integer )
begin
        Z := 0
        V := X;
        for i := 0 to 127 do begin
                if  Yi  = 1 then Z := Z ⊕ V;
                V = rightshift(V, 1);
                if V127 = 1 then V := V ⊕ R;
        end
        return Z;
end
```

- This is written $Z = X \cdot Y$
- Y_i is *i*th leftmost bit of *Y*, so Y_{127} is the rightmost bit of *Y*
- rightshift(*V*, 1) means to shift *V* right 1 bit, and bring in 0 from the left
- R is bits `11100001` followed by 120 0 bits

GCM Hash Function

GHASH(H, A, C) computed as follows:

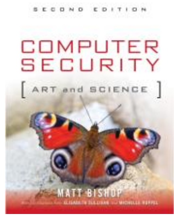
1. $X_0 = 0$
2. for $i = 1, \ldots, m-1$, $X_i = (X_{i-1} \oplus A_i) \cdot H$
3. $X_m = (X_{m-1} \oplus A_m) \cdot H$
    - A_m is right-padded with 0s if not a complete block
4. for $i = m+1, \ldots, m+n-1$, $X_i = (X_{i-1} \oplus C_i) \cdot H$
5. $X_{m+n} = (X_{m+n-1} \oplus C_n) \cdot H$
    - C_n is right-padded with 0s if not a complete block
6. $X_{m+n+1} = (X_{m+n} \oplus (L_A \mid\mid L_C)) \cdot H$
    - L_A, L_C left-padded with 0 bits to form 64 bits each



GCM Authenticated Encryption

This computes C and T:

1. $H = E_k(0^{128})$
2. If IV is 96 bits, $Y_0 = IV \mid\mid 0^{31}1$; otherwise, $Y_0 = \text{GHASH}(H, \nu, IV)$
    - ν empty string
3. for $i = 1, \ldots n$, $I_i = I_{i\text{-}1} + 1 \bmod 2^{32}$; set $Y_i = L_{i\text{-}1} \mid\mid I_i$
    - $I_{i\text{-}1}$ right part of $Y_{i\text{-}1}$; treat it as unsigned 32 bit integer; $L_{i\text{-}1}$ left part of $Y_{i\text{-}1}$
4. for $i = 1, \ldots n{-}1$, $C_i = M_i + E_k(Y_i)$
5. $C_n = M_n + \text{MSB}_u(E_k(Y_n))$
    - $\text{MSB}_u(X)$ is u most significant (leftmost) bits of X
6. $T = \text{MSB}_t(\text{GHASH}(H, A, C) + E_k(Y_0))$



GCM Transmission and Decryption

- Send *C*, *T*
- To verify, perform steps 1, 2, 6, 3, 4, 5
- When authentication value is computed, compare to sent value
  - Note this is done *before* decrypting the message
  - If they do not match, return failure and discard messages

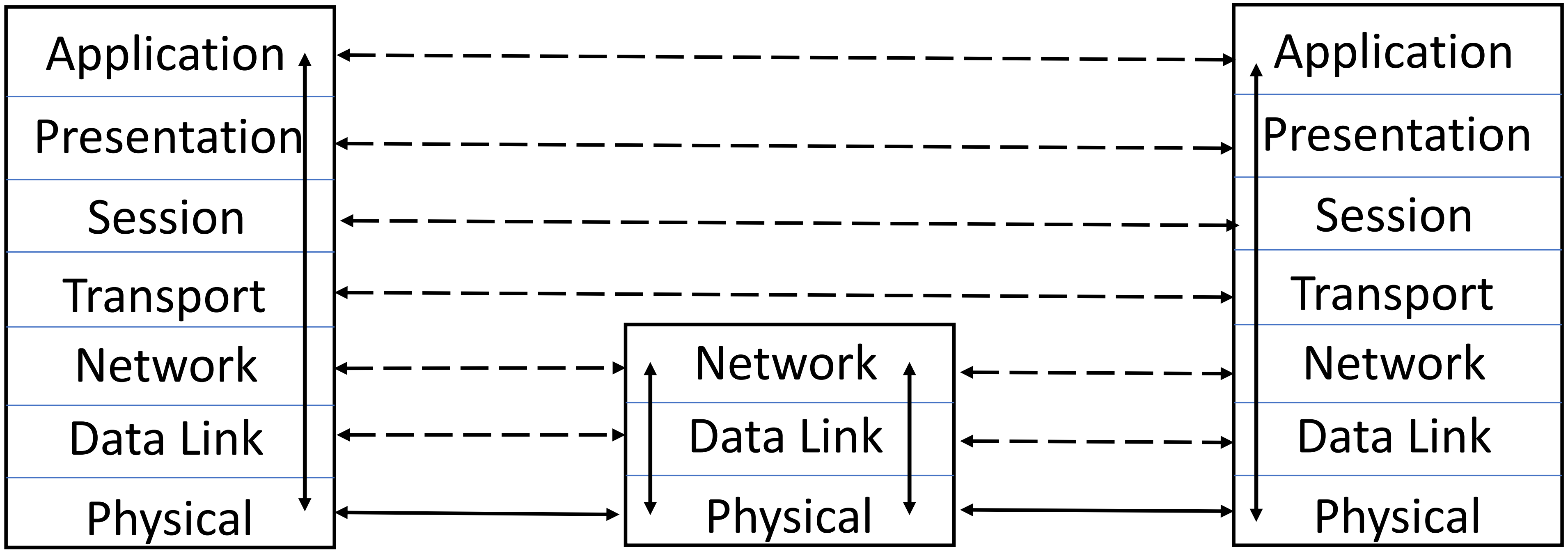
GCM Analysis

Strength depends on certain properties

- If *IV* (nonce) reused, part of *H* can be obtained
- If length of authentication value too short, forgeries can occur and from that, *H* can be determined (enabling undetectable forgeries)
- Under study is whether particular values of *H* make forging messages easier
- Restricting length of *IV* to 96 bits produces a stronger AEAD cipher than when the length is not restricted

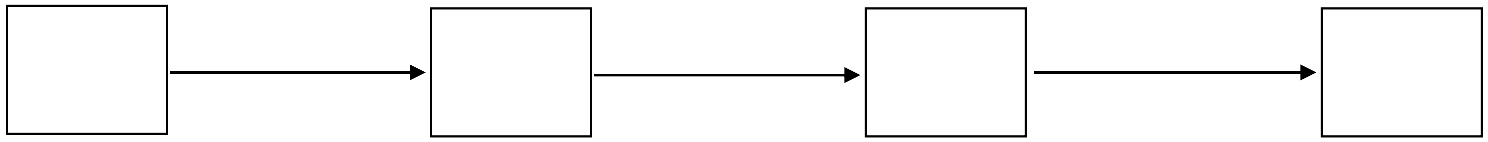
Networks and Cryptography

- ISO/OSI model
- Conceptually, each host communicates with peer at each layer

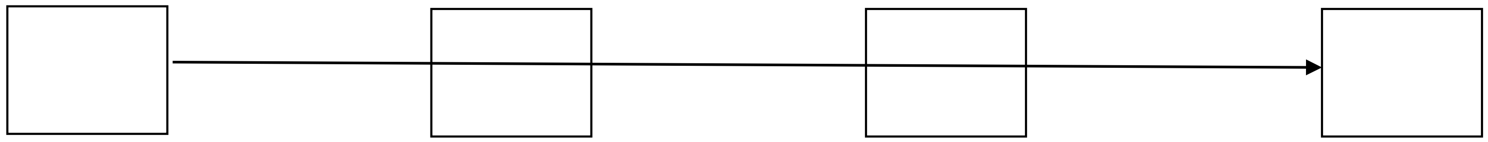


Link and End-to-End Protocols

Link Protocol

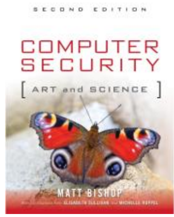


End-to-End (or E2E) Protocol



Encryption

- Link encryption
  - Each host enciphers message so host at “next hop” can read it
  - Message can be read at intermediate hosts
- End-to-end encryption
  - Host enciphers message so host at other end of communication can read it
  - Message cannot be read at intermediate hosts

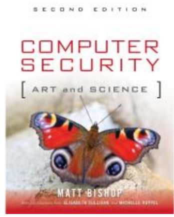


Examples

- SSH protocol
  - Messages between client, server are enciphered, and encipherment, decipherment occur only at these hosts
  - End-to-end protocol
- PPP Encryption Control Protocol
  - Host gets message, deciphers it
    - Figures out where to forward it
    - Enciphers it in appropriate key and forwards it
  - Link protocol

Cryptographic Considerations

- Link encryption
  - Each host shares key with neighbor
  - Can be set on per-host or per-host-pair basis
    - Windsor, stripe, seaview each have own keys
    - One key for (windsor, stripe); one for (stripe, seaview); one for (windsor, seaview)
- End-to-end
  - Each host shares key with destination
  - Can be set on per-host or per-host-pair basis
  - Message cannot be read at intermediate nodes



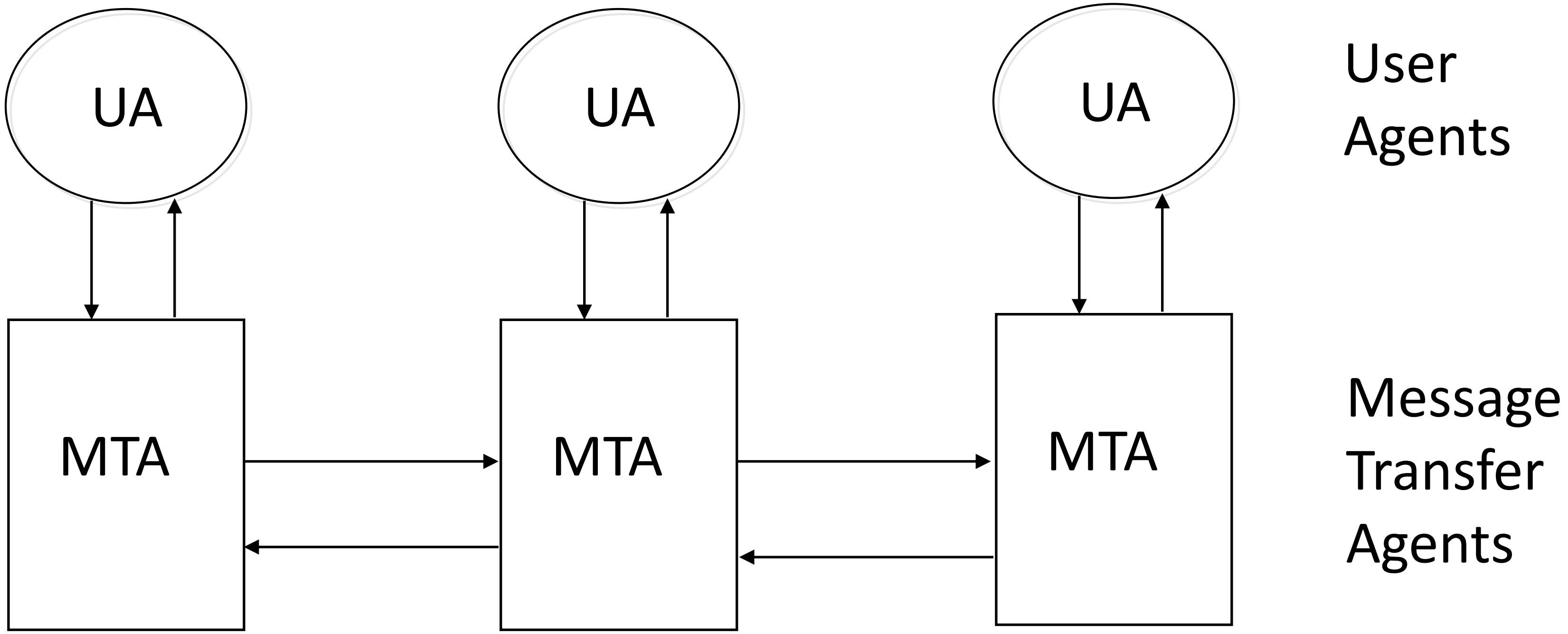
Traffic Analysis

- Link encryption
  - Can protect headers of packets
  - Possible to hide source and destination
    - Note: may be able to deduce this from traffic flows
- End-to-end encryption
  - Cannot hide packet headers
    - Intermediate nodes need to route packet
  - Attacker can read source, destination

Example Protocols

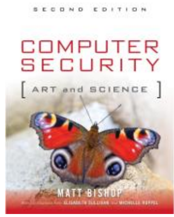
- Securing Electronic Mail (OpenPGP, PEM)
  - Applications layer protocol
  - Start with PEM as goals, design described in detail; then lool at OpenPGP
- Securing Instant Messaging (Signal)
  - Applications layer protocol
- Secure Socket Layer (TLS)
  - Transport layer protocol
- IP Security (IPSec)
  - Network layer protocol

How Email Works



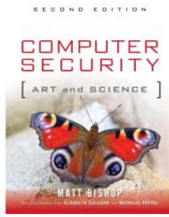
Goals of PEM

1. Confidentiality
   - Only sender and recipient(s) can read message
2. Origin authentication
   - Identify the sender precisely
3. Data integrity
   - Any changes in message are easy to detect
4. Non-repudiation of origin
   - Whenever possible …



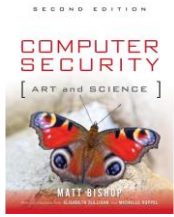
Design Principles

- Do not change related existing protocols
  - Cannot alter SMTP
- Do not change existing software
  - Need compatibility with existing software
- Make use of PEM optional
  - Available if desired, but email still works without them
  - Some recipients may use it, others not
- Enable communication without prearrangement
  - Out-of-bands authentication, key exchange problematic



Basic Design: Keys

- Two keys
  - *Interchange keys* tied to sender, recipients and is static (for some set of messages)
    - Like a public/private key pair (indeed, may be a public/private key pair)
    - Must be available *before* messages sent
  - *Data exchange keys* generated for each message
    - Like a session key, session being the message

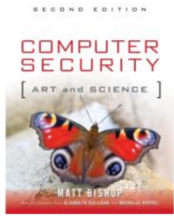


Basic Design: Sending

Confidentiality

- m message
- k_s data exchange key
- k_B Bob's interchange key

$$\text{Alice} \xrightarrow{\{\, m \,\}\, k_s \;||\; \{\, k_s \,\}\, k_B} \text{Bob}$$



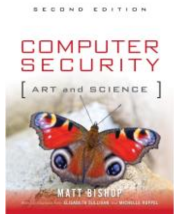
Basic Design: Integrity

Integrity and authentication:

- m message
- $h(m)$ hash of message m —Message Integrity Check (MIC)
- k_A Alice’s interchange key

$$\text{Alice} \xrightarrow{\quad m \, \{ \, h(m) \, \} \, k_A \quad} \text{Bob}$$

Non-repudiation: if k_A is Alice’s private key, this establishes that Alice’s private key was used to sign the message



Basic Design: Everything

Confidentiality, integrity, authentication:

- Notations as in previous slides
- If k_A is Alice's private key, get non-repudiation too

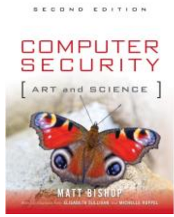
$$\text{Alice} \xrightarrow{\{\, m \,\}\, k_s \,||\, \{\, h(m) \,\}\, k_A \,||\, \{\, k_s \,\}\, k_B} \text{Bob}$$

Practical Considerations

- Limits of SMTP
  - Only ASCII characters, limited length lines
- Use encoding procedure
  1. Map local char representation into canonical format
     - Format meets SMTP requirements
  2. Compute and encipher MIC over the canonical format; encipher message if needed
  3. Map each 6 bits of result into a character; insert newline after every 64th character
  4. Add delimiters around this ASCII message

Problem

- Recipient without PEM-compliant software cannot read it
  - If only integrity and authentication used, should be able to read it
- Mode MIC-CLEAR allows this
  - Skip step 3 in encoding procedure
  - Problem: some MTAs add blank lines, delete trailing white space, or change end of line character
  - Result: PEM-compliant software reports integrity failure

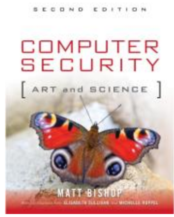


PEM vs. OpenPGP

- Use different ciphers
  - PGP allows several ciphers
    - Public key: RSA, El Gamal, DSA, Diffie-Hellman, Elliptic curve
    - Symmetric key: IDEA, Triple DES, CAST5, Blowfish, AES-128, AES-192, AES-256, Twofish-256
    - Hash algorithms: MD5, SHA-1, RIPE-MD/160, SHA256, SHA384, SHA512, SHA224
  - PEM allows RSA as public key algorithm, DES in CBC mode to encipher messages, MD2, MD5 as hash functions
- Use different certificate models
  - PGP uses general “web of trust”
  - PEM uses hierarchical certification structure
- Handle end of line differently
  - PGP remaps end of line if message tagged “text”, but leaves them alone if message tagged “binary”
  - PEM always remaps end of line

Signal: Instant Messaging

- Provides confidentiality, authentication, integrity, perfect forward secrecy
- Three steps:
    - Client registers with messaging server
    - Two clients set up a session
    - They exchange messages

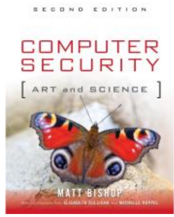


Client Keys

- Long-term identity key pair IK
  - Curve25519 key generated when client program is installed
- Medium-term signed pre-key pair SPK
  - Also a Curve25519 key generated when client program is installed
  - Change periodically
- Ephemeral one-time pre-key pair OPK
  - Also a Curve25519 key selected from a list generated when client program is installed; when the list is used up, another list is generated

Session Keys

- *message key*: 80-byte key used to encrypt messages
  - 32-byte key for AES-256 encryption
  - 32-byte key for HMAC-SHA256 cryptographic checksum
  - 16-byte initialization vector
- *chain key*: 32-byte value used to generate message keys
- *root key*: 32-byte value used to generate chain keys



Cryptographic Functions

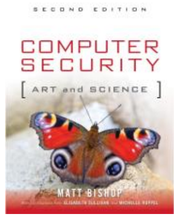
Symmetric key generation:

- Use HMAC-SHA256
- Use a 2-stage HMAC-based key derivation function

  First stage:
  - s a non-secret salt; if omitted, use 0; x is other material, k is key:

    k = HMAC_SHA256(s, x)

  Second stage:
  - *info* string of characters like “WhisperGroup”, L number of octets to produce
  - $T(0)$ = “” (empty string), $T(i)$ = HMAC_SHA256(k, $T(i-1)$ || *info* || i)
  - Compute to L octets HDKF_Extend(k, *info*) = $T(1)$ || $T(2)$ || . . .
  - First L octets are the result, HDKF(s, x)



Notation

- *W* is signal message server
- $k_{pub,\mathrm{A}}$ is A’s public key, $k_{priv,\mathrm{A}}$ is A’s private key
- ECDH is elliptic curve Diffie-Hellman
- Alice wishes to communicate with Bob

Registration Step

- Alice signs her public key $SPK_{pub,\text{Alice}}$:

$$SSPK_{\text{Alice}} = \{SPK_{pub,\text{Alice}}\}\, IK_{priv,\text{Alice}}$$

- She sends her *pre-key bundle*:

$$\text{Alice} \xrightarrow[OPK_{pub,\text{Alice},2} \,||\, \ldots\}]{\{IK_{pub,\text{Alice}} \,||\, SPK_{pub,\text{Alice}} \,||\, SSPK_{\text{Alice}} \,||\, OPK_{pub,\text{Alice},1} \,||} W$$

where $OPK_{pub,\text{Alice},1}$, $OPK_{pub,\text{Alice},2}$, . . . are the ephemeral one-time pre-key public keys

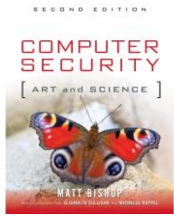
- Bob also must register

Session Setup and Initial Message

- Alice requests Bob's pre-key bundle from *W*
  - *W* sends it; note only 1 ephemeral one-time pre-key public key is included
  - If Bob's one-time pre-keys are all used, no such keys included

$$\text{Alice} \xrightarrow{\textit{message requesting Bob's pre-key bundle}} W$$

$$\text{Alice} \xleftarrow{\{\, IK_{pub,\text{Bob}} \,||\, SPK_{pub,\text{Bob}} \,||\, SSPK_{\text{Bob}} \,||\, OPK_{pub,\text{Bob},I} \,\}} W$$

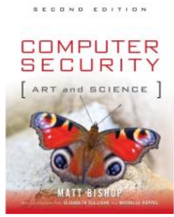


Session Setup and Initial Message

- Alice verifies $SSPK_{\text{Bob}}$ is the signature for $SPK_{pub,\text{Bob}}$
  - If it isn't, setup stops
- Alice generates another ephemeral key pair EK
  - It's another Curve25519 key pair
- Alice now computes a master secret *ms*:

$$ms = \text{ECDH}(IK_{priv,\text{Alice}}, SPK_{pub,\text{Bob}}) \,||\, \text{ECDH}(EK_{priv,\text{Alice}}, IK_{pub,\text{Bob}}) \,||\, \text{ECDH}(EK_{priv,\text{Alice}}, SPK_{pub,\text{Bob}}) \,||\, \text{ECDH}(EK_{priv,\text{Alice}}, OPK_{pub,\text{Bob},i})$$

  - If $OPK_{pub,\text{Bob},i}$ not sent, omit last encryption
- Alice deletes $EK_{priv,\text{Alice}}$, all intermediate values used to compute *ms*



Session Setup and Initial Message

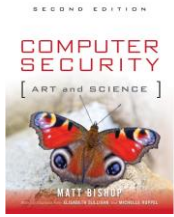
- Alice computes HDKF(c_0, c_1 || ms)
  - c_0 is 256 0 bits and c_1 is 256 1 bits
- First 32 bits are root key k_r, next 32 bits are first chain key $k_{c,1}$
- Alice creates associated data $A = IK_{pub,\text{Alice}} \,||\, IK_{pub,\text{Bob}}$
  - May also append additional information

Sending Messages

- Alice creates message key k_m = HMAC_SHA256($k_{c,1}$, 1)
- Alice encrypts message using AEAD scheme with AES-256 in CBC mode for encryption and HMAC_SHA256 for authentication
  - Call result C

$$\text{Alice} \xrightarrow{\{\, IK_{pub,\text{Alice}} \,||\, EK_{pub,\text{Alice}} \,||\, \textit{pre-key indicator} \,||\, C \,\}} \text{Bob}$$

  - $EK_{pub,\text{Alice}}$ is a *new* ephemeral Curve25519 public key
  - *pre-key indicator* indicates to Bob which of his ephemeral one-time pre-keys was used



Sending Messages

- Bob receives message
- Bob computes master secret *ms* analogously to Alice, but using his private keys and Alice's public keys
  - After, Bob deletes ($OPK_{pub,\text{Bob},I}$, $OPK_{priv,\text{Bob},i}$)
- Bob computes the root and chain keys
  - All information to do this is in what Alice sent him, so can do it offline
- Now they begin to exchange messages

Sending Messages

- When Alice sends messages before receiving Bob's reply to any, uses a *hash ratchet* to change message key for each message:

$$k_{m,i+1} = \text{HMAC\_SHA256}(k_{c,i}, 1)$$
$$k_{c,i+1} = \text{HMAC\_SHA256}(k_{c,i}, 2)$$

- When Alice receives a reply from Bob, she computes new chain, root key:

$$x = \text{HKDF}(k_r, \text{ECDH}(EK_{pub,\text{Bob}}, EK_{priv,\text{Alice}}))$$

where $EK_{pub,\text{Bob}}$ in received message, $EK_{priv,\text{Alice}}$ private key associated with $EK_{pub,\text{Alice}}$ that Alice sent in message Bob is replying to

- First 32 octets are new chain key, next 32 octets new root key

Signal Protocol Use

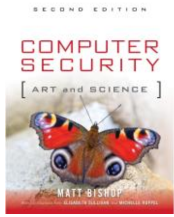
- Much of the manipulation is to provide perfect forward secrecy
  - So previously sent messages remain secret if current keys are discovered
- Signal widely used in instant messaging services like Signal and WhatsApp

Transport Layer Security

- Internet protocol: TLS
  - Provides confidentiality, integrity, authentication of endpoints
  - Focus on version 1.2
- Old Internet protocol: SSL
  - Developed by Netscape for WWW browsers and servers
  - Use is deprecated

TLS Session

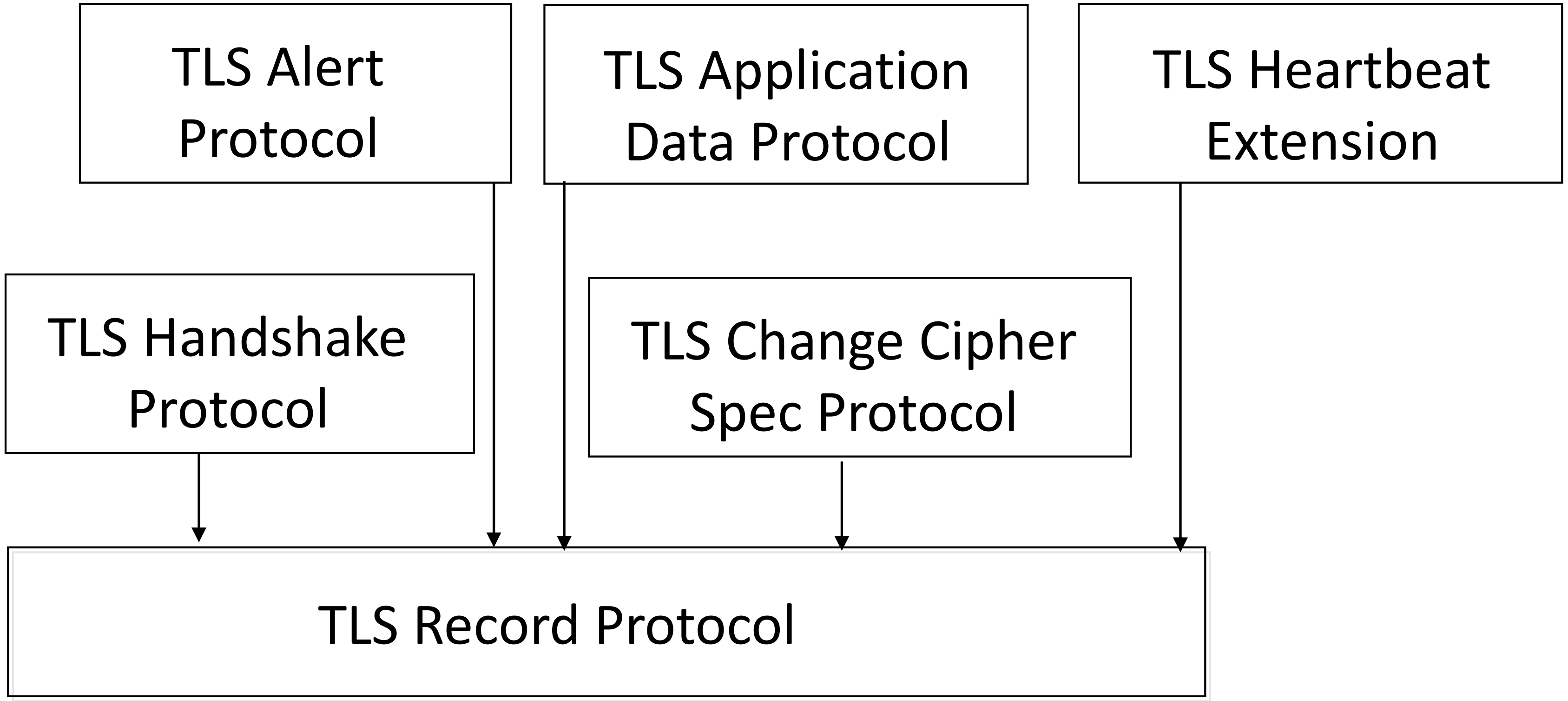
- Association between two peers
  - May have many associated connections
  - Information related to session for each peer:
    - Unique session identifier
    - Peer’s X.509v3 certificate, if needed
    - Compression method
    - Cipher spec for cipher and MAC
    - “Master secret” of 48 bits shared with peer
    - Flag indicating whether this session can be used to start new conncetion

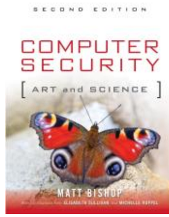


TLS Connection

- Describes how data exchanged with peer
- Information for each connection
  - Whether a server or client
  - Random data for server and client
  - Write keys (used to encipher data)
  - Write MAC key (used to compute MAC)
  - Initialization vectors for ciphers, if needed
  - Sequence numbers for server, client

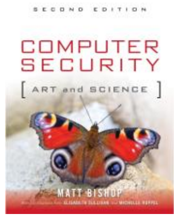
Structure of TLS





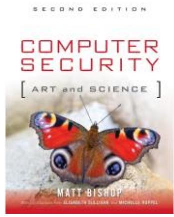
Supporting Cryptogrphy

- All parts of TLS use them
- Initial phase: public key system exchanges keys
    - Messages enciphered using classical ciphers, checksummed using cryptographic checksums
    - Only certain combinations allowed
        - Depends on algorithm for interchange cipher
    - Interchange algorithms: RSA, Diffie-Hellman



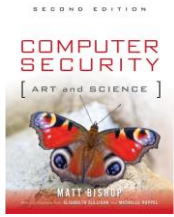
Diffie-Hellman: Types

- Diffie-Hellman: certificate contains D-H parameters, signed by a CA
    - DSS or RSA algorithms used to sign
- Ephemeral Diffie-Hellman: DSS or RSA certificate used to sign D-H parameters
    - Parameters not reused, so not in certificate
- Anonymous Diffie-Hellman: D-H with neither party authenticated
    - Use is “strongly discouraged” as it is vulnerable to attacks
- Elliptic curve Diffie-Hellman supports Diffie-Hellman and ephemeral Diffie-Hellman
    - But not anonymous Diffie-Hellman



Derivation of Master Secret

- *master_secret* = *PRF*(*premaster*, "master secret", $r_1 \,||\, r_2$)
  - *premaster* set by client, ˚sent to server during setup
  - r_1, r_2 random numbers from client, server respectively
- *PRF*(*secret*, *label*, *seed*) = P_*hash*(*secret*, *label* || *seed*)
- P_*hash*(*secret*, *seed*) = HMAC_*hash*(*secret* || *A*(1) || *seed*) ||
  HMAC_*hash*(*secret* || *A*(2) || *seed*) ||
  HMAC_*hash*(*secret* || *A*(3) || *seed*) || …
  - Use first 48 bits of output to set *PRF*
- *A*(0) = *seed*, *A*(*i*) = HMAC_*hash*(*secret*, *A*(*i*-1)) for $i > 0$



Derivation of Keys

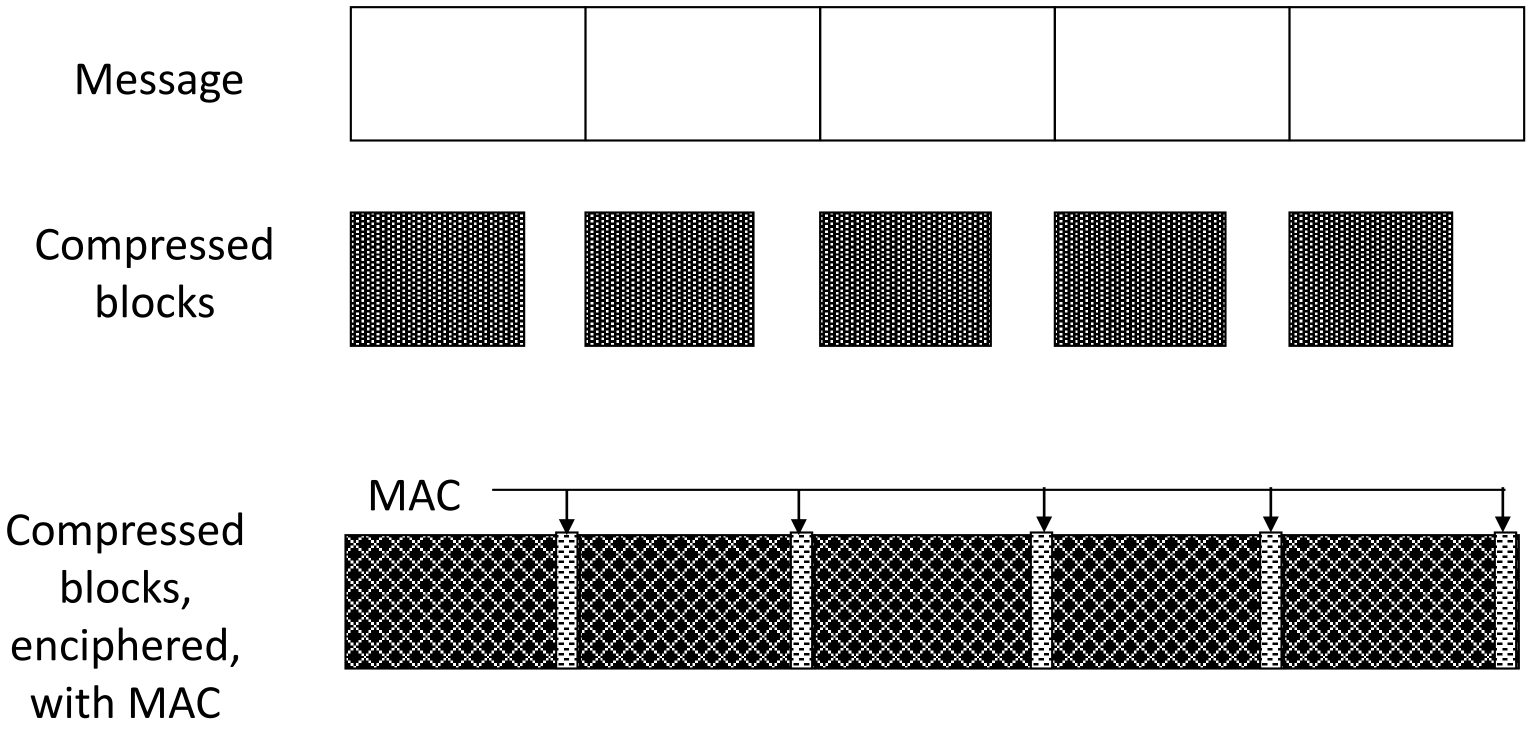
- *key_block* = PRF(*master*, "key expansion", $r_1 \,||\, r_2$)
  - r_1, r_2 as before
- Break it into blocks of 48 bits
  - First two are client, server keys for computing MACs
  - Next two are client, server keys used to encipher messages
  - Next two are client, server initialization vectors
    - Omitted if cipher does not use initialization vector

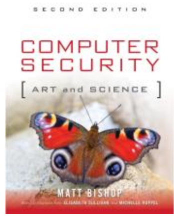
MAC for Block

hash(*MAC_ws*, *seq* || *TLS_comp* || *TLS_vers* || *TLS_len* || *block*)

- *MAC_ws*: MAC write key
- *seq*: sequence number of *block*
- *TLS_comp*: message type
- *TLS_vers*: TLS version
- *TLS_len*: length of *block*
- *block*: block being sent

SSL Record Layer





Record Protocol Overview

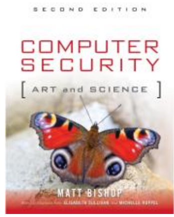
- Lowest layer, taking messages from higher
  - Max block size 2^{14} = 16,384 bytes
  - Bigger messages split into multiple blocks
- Construction
  - Block *b* compressed; call it b_c
  - MAC computed for b_c
    - If MAC key not selected, no MAC computed
  - b_c, MAC enciphered
    - If enciphering key not selected, no enciphering done
  - TLS record header prepended

TLS Handshake Protocol

- Used to initiate connection
  - Sets up parameters for record protocol
  - 4 rounds
- Upper layer protocol
  - Invokes Record Protocol
- Note: what follows assumes client, server using RSA as interchange cryptosystem

Overview of Rounds

1. Create TLS connection between client, server
2. Server authenticates itself
3. Client validates server, begins key exchange
4. Acknowledgments all around



Handshake Round 1

1. Client $\xrightarrow{\{ v_C \,||\, r_1 \,||\, s_1 \,||\, ciphers \,||\, comps \,||\, ext_C \}}$ Server

2. Client $\xleftarrow{\{ v \,||\, r_2 \,||\, s_2 \,||\, cipher \,||\, comp \,||\, ext \}}$ Server

v_C	Client's version of SSL
v	Highest version of SSL that client, server both understand
r_1, r_2	nonces (timestamp and 28 random bytes)
s_1	Current session id (empty if new session)
s_2	Current session id (if s_1 empty, new session id)
ciphers	Ciphers that client understands
comps	Compression algorithms that client understand
cipher	Cipher to be used
comp	Compression algorithm to be used
ext_C	List of extensions client supports
ext	List of extensions server supports (subset of ext_C)

Handshake Round 2

3. Client ← $\{$ *certificate chain* $\}$ — Server

4. Client ← $\{ p \mid\mid g \mid\mid K_S \mid\mid \{ h(r_1 \mid\mid r_2 \mid\mid p \mid\mid g \mid\mid K_S) \} k_S \}$ — Server

5. Client ← $\{$ *ctype* || *sigalgs* || *gca* $\}$ — Server

6. Client ← $\{$ *server_hello_done* $\}$ — Server

If server not going to authenticate itself, only last message sent

Second step is for Diffie-Hellman with RSA certificate

Third step omitted if server does not need client certificate

K_S, k_S	Server's Diffie-Hellman public, private keys
ctype	Certificate type accepted (by cryptosystem)
sigalgs	List of hash, signature algorithm pairs server can use
gca	Acceptable certification authorities

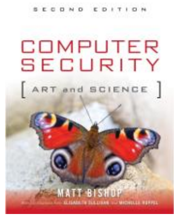
Handshake Round 3

7. Client $\xrightarrow{\{\,client\_certificate\,\}}$ Server

8. Client $\xrightarrow{\{\,pre\,\}\,K_S}$ Server

9. Client $\xrightarrow{\{\,hash(all\ previous\ messages)\,\}\,k_C}$ Server

pre	Premaster secret
K_S	Server's public key
k_C	Client's private key



Handshake Round 4

10. Client → *change_cipher_spec* → Server

11. Client → { *PRF*(*master* || “client finished” || *hash*(*all previous messages*) } → Server

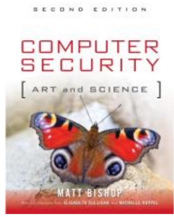
12. Client ← *change_cipher_spec* ← Server

13. Client ← { *PRF*(*master* || “server finished” || *hash*(*all previous messages*) } ← Server

change_cipher_spec Begin using cipher specified

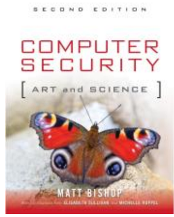
TLS Change Cipher Spec Protocol

- Send single byte
- In handshake, new parameters considered “pending” until this byte received
  - Old parameters in use, so cannot just switch to new ones



TLS Alert Protocol

- Closure alert
  - Sender will send no more messages
  - Pending data delivered; new messages ignored
- Error alerts
  - Warning: connection remains open
  - Fatal error: connection torn down as soon as sent or received

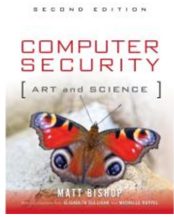


TLS Heartbeat Extension

- Message has 4 fields
  - Value indicating message is request
  - Length of data in message
  - Data of given length
  - Random data
- Message sent to peer; peer replies with similar message
  - If second field is too large (> 214 bytes), ignore message
  - Reply message has same data peer sent, new random data
- When peer sends this for the first time, it sends nothing more until a response is received

TLS Application Data Protocol

- Passes data from application to TLS Record Protocol layer



Differences Between TLSv2 and SSLv3

- Master secret computed differently

$$
\begin{aligned}
\textit{master} = {} & \mathrm{MD5}(\textit{premaster} \,||\, \mathrm{SHA}(\text{‘A’} \,||\, \textit{premaster} \,||\, r_1 \,||\, r_2) \,||\\
& \mathrm{MD5}(\textit{premaster} \,||\, \mathrm{SHA}(\text{‘BB’} \,||\, \textit{premaster} \,||\, r_1 \,||\, r_2) \,||\\
& \mathrm{MD5}(\textit{premaster} \,||\, \mathrm{SHA}(\text{‘CCC’} \,||\, \textit{premaster} \,||\, r_1 \,||\, r_2)
\end{aligned}
$$

- Key block also computed differently

$$
\begin{aligned}
\textit{key\_block} = {} & \mathrm{MD5}(\textit{master} \,||\, \mathrm{SHA}(\text{‘A’} \,||\, \textit{master} \,||\, r_1 \,||\, r_2) \,||\\
& \mathrm{MD5}(\textit{master} \,||\, \mathrm{SHA}(\text{‘BB’} \,||\, \textit{master} \,||\, r_1 \,||\, r_2) \,||\\
& \mathrm{MD5}(\textit{master} \,||\, \mathrm{SHA}(\text{‘CCC’} \,||\, \textit{master} \,||\, r_1 \,||\, r_2) \;\; || \ldots
\end{aligned}
$$

Differences Between TLSv2 and SSLv3

MAC for each block computed differently:

hash(*MAC_ws* || *opad* ||

hash(*MAC_ws* || *ipad* || *seq* || *SSL_comp* || *SSL_len* || *block*))

- *hash*: hash function used
- *MAC__ws*, *seq*, *SSL_comp*, *SSL_len*, *block*: as for TLS (with obvious changes)
- *ipad, opad*: as for HMAC

Differences Between TLSv2 and SSLv3

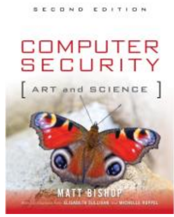
- Verification message (9, above) is different:

9′. Client $\xrightarrow{\{\, \textit{hash}(\textit{master} \,||\, \textit{opad} \,||\, \textit{hash}(\textit{all previous messages} \,||\, \textit{master} \,||\, \textit{ipad}))\, \}}$ Server

- Messages after change cipher spec (11, 13 above) are also different:

11′. Client $\xrightarrow{\{\, \textit{hash}(\textit{master} \,||\, \textit{opad} \,||\, \textit{hash}(\textit{all previous messages} \,||\, \text{0x434C4E54} \,||\, \textit{master} \,||\, \textit{ipad}))\, \}}$ Server

13′. Client $\xrightarrow{\{\, \textit{hash}(\textit{master} \,||\, \textit{opad} \,||\, \textit{hash}(\textit{all previous messages} \,||\, \text{0x53525652} \,||\, \textit{master} \,||\, \textit{ipad}))\, \}}$ Server

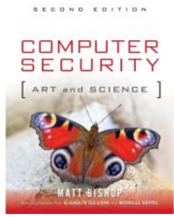


Differences Between TLSv2 and SSLv3

- Different sets of ciphers
  - SSL allows use of RC4, but its use is deprecated
  - SSL allows set of ciphers for the Fortezza cryptographic token used by the U.S. Department of Defense

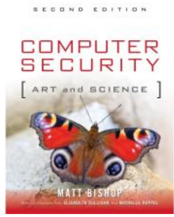
Problems with SSL

- POODLE attack focuses on padding of messages
  - In SSL, all but the last byte of the padding are random and so cannot be checked
- How padding works (assume block size of *b*):
  - Message ends in a full block: add additional block of padding, and last byte is the number of bytes of random padding (b – 1)
  - Message ends in part of a block: add random bytes out to last byte, set that to number of random bytes (so if block is b – 1 bytes, one padding byte added and it is 0)



The POODLE Attack

- Peer receives incoming ciphertext message $c_1, \ldots, c_n$
- Peer decrypts it to $m_1, \ldots, m_n$: $m_i = D_k(c_i) \oplus c_{i-1}$, where c_0 is initialization vector
  - Validates by removing padding, computes and checks MAC over remaining bytes
- Attacker replaces c_n with some earlier block, say c_j, $j \neq n$
  - If last byte of c_j is same as c_n, message accepted as valid; otherwise, rejected
- So attacker arranges for HTTP messages to end with known number of padding bytes
  - Then server should accept changed message in at least 1 out of 256 tries



Example POODLE Attack

- Here's HTTP request (somewhat simplified):

GET / HT TP/1.1\r\n Cookie: abcdefgh \r\n\r\nxxxx MAC •••••••7

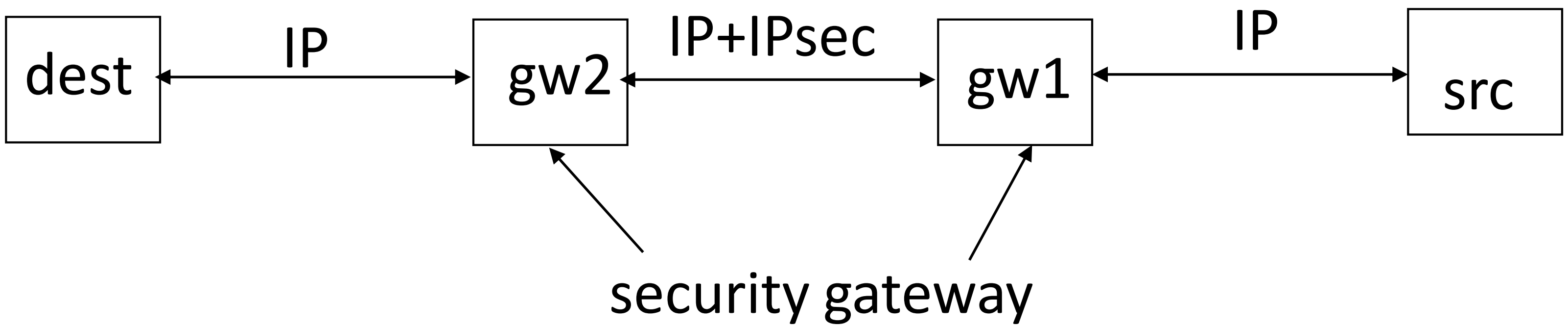
- Attacker cannot see plaintext
- Run Javascript in browser that duplicates cookie block and overwrites last block
  - It's enciphered using (for example) 3DES-CBC
- You see enciphered block
  - If it is accepted, then plaintext block xor'ed with previous ciphertext block ends in 7

SSL, TLS, and POODLE

- POODLE serious enough that SSL is being discarded in favor of TLS
- TLS not vulnerable, as all padding bytes set to length of padding
  - And TLS implementations must check this padding (all of it) for validity before accepting messages

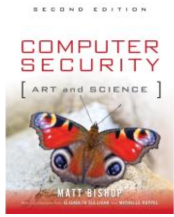
IPsec

- Network layer security
  - Provides confidentiality, integrity, authentication of endpoints, replay detection
- Protects all messages sent along a path

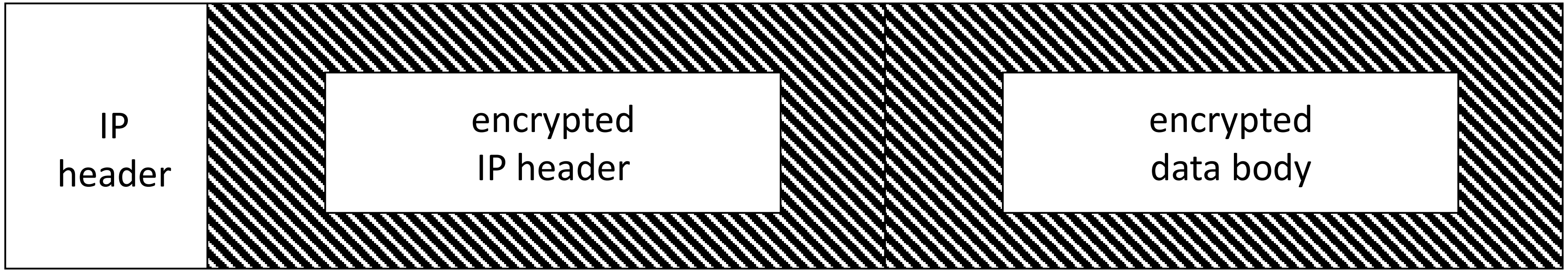


IPsec Transport Mode

- Encapsulate IP packet data area
- Use IP to send IPsec-wrapped data packet
- Note: IP header not protected



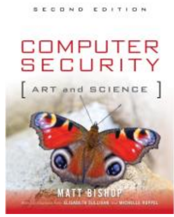
IPsec Tunnel Mode



- Encapsulate IP packet (IP header *and* IP data)
- Use IP to send IPsec-wrapped packet
- Note: IP header protected

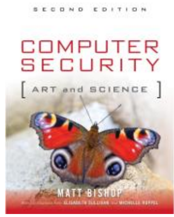
IPsec Protocols

- Authentication Header (AH)
  - Message integrity
  - Origin authentication
  - Anti-replay
- Encapsulating Security Payload (ESP)
  - Confidentiality
  - Others provided by AH



IPsec Architecture

- Security Policy Database (SPD)
  - Says how to handle messages (discard them, add security services, forward message unchanged)
  - SPD associated with network interface
  - SPD determines appropriate entry from packet attributes
    - Including source, destination, transport protocol

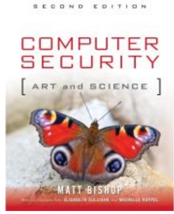


Example

- Goals
  - Discard SMTP packets from host 192.168.2.9
  - Forward packets from 192.168.19.7 without change
- SPD entries

```
src 192.168.2.9, dest 10.1.2.3 to 10.1.2.103, port 25, discard
src 192.168.19.7, dest 10.1.2.3 to 10.1.2.103, port 25, bypass
dest 10.1.2.3 to 10.1.2.103, port 25, apply IPsec
```

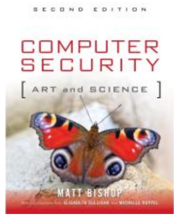
- Note: entries scanned in order
  - If no match for packet, it is discarded



IPsec Architecture

- Security Association (SA)
    - Association between peers for security services
        - Identified uniquely by dest address, security protocol (AH or ESP), unique 32-bit number (security parameter index, or SPI)
    - Unidirectional
        - Can apply different services in either direction
    - SA uses either ESP or AH; if both required, 2 SAs needed

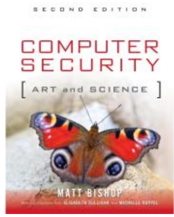
SA Database (SAD)



- Entry describes SA; some fields for all packets:
  - AH algorithm identifier, keys
    - When SA uses AH
  - ESP encipherment algorithm identifier, keys
    - When SA uses confidentiality from ESP
  - ESP authentication algorithm identifier, keys
    - When SA uses authentication, integrity from ESP
  - ESP integrity algorithm identifier, keys
    - When SA uses authentication, integrity from ESP
  - SA lifetime (time for deletion or max byte count)
  - IPsec mode (tunnel, transport, either)

SAD Fields

- Antireplay (inbound only)
  - When SA uses antireplay feature
- Sequence number counter (outbound only)
  - Generates AH or ESP sequence number
- Sequence counter overflow field (outbound only)
  - Stops traffic over this SA if sequence counter overflows



IPsec Architecture

- Packet arrives
- Look in SPD
  - Find appropriate entry based on attributes of packet such as source, destination addresses and ports, protocol, etc.
  - Identifies entry or entries in SAD based on SPD entries, packet information
- Find associated SA in SAD
  - Search for match on SPI, source, destination address; if none, search for match on SPI, destination address; if none, use just SPI or both SPI, protocol; if none, discard packet
  - Apply security services in SA (if any)

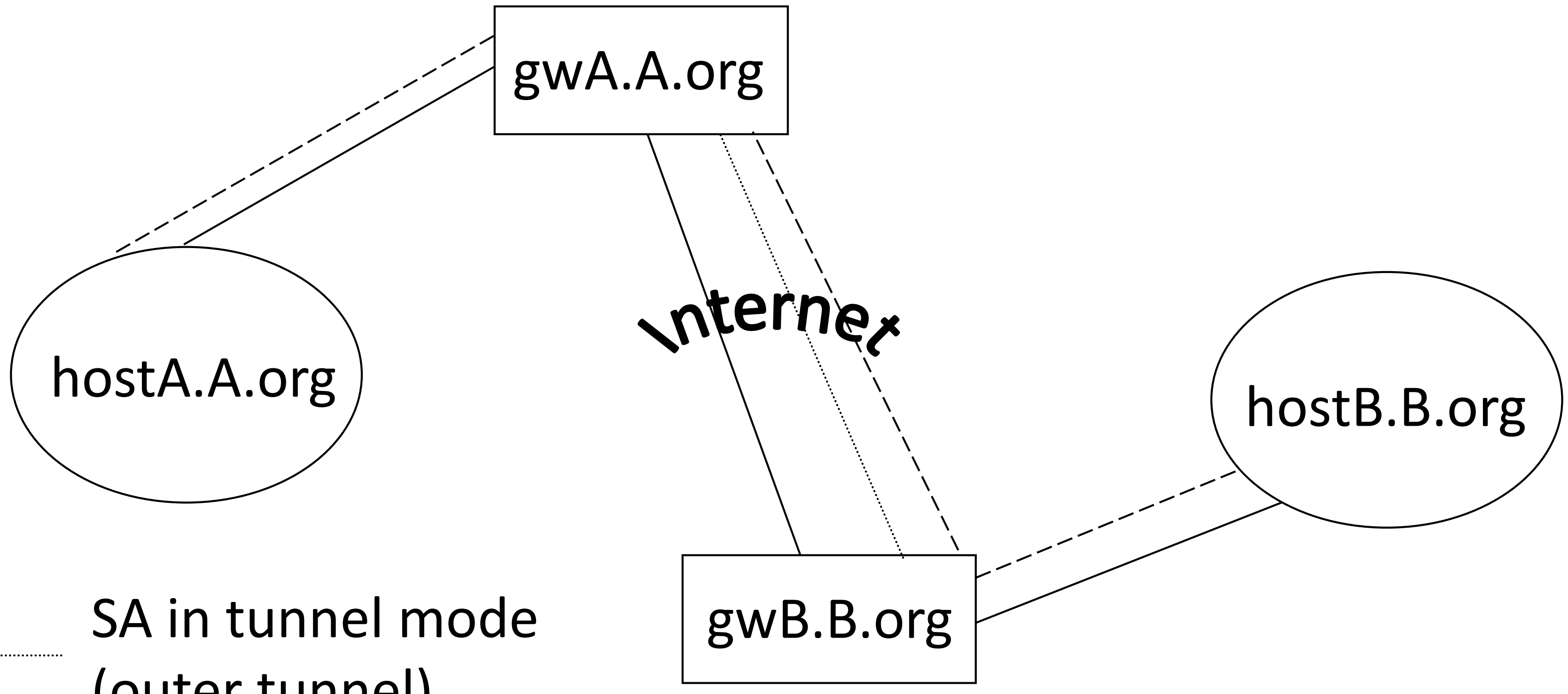
SA Bundles and Nesting

- Sequence of SAs that IPsec applies to packets
  - This is a *SA bundle*
- Nest tunnel mode SAs
  - This is *iterated tunneling*

Example: Iterated Tunneling

- Group in A.org needs to communicate with group in B.org
- Gateways of A, B use IPsec mechanisms
  - But the information must be secret to everyone except the two groups, even secret from other people in A.org and B.org
- Inner tunnel: a SA between the hosts of the two groups
- Outer tunnel: the SA between the two gateways

Example: Systems



SA in tunnel mode (outer tunnel)

SA in tunnel mode (inner tunnel)

Example: Packets

IP header from gwA	AH header from gwA	ESP header from gwA	IP header from hostA	AH header from hostA	ESP header from hostA	IP header from hostA	Transport layer headers, data

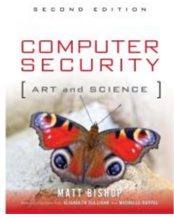
- Packet generated on hostA
- Encapsulated by hostA's IPsec mechanisms
- Again encapsulated by gwA's IPsec mechanisms
    - Above diagram shows headers, but as you go left, everything to the right would be enciphered and authenticated, *etc*.

AH Protocol

- Parameters in AH header
  - Length of header
  - SPI of SA applying protocol
  - Sequence number (anti-replay)
  - Integrity value check
- Two steps
  - Check that replay is not occurring
  - Check authentication data

Sender

- Check sequence number will not cycle
- Increment sequence number
- Compute IVC of packet
  - Includes IP header, AH header, packet data
    - IP header: include all fields that will not change in transit; assume all others are 0
    - AH header: authentication data field set to 0 for this
    - Packet data includes encapsulated data, higher level protocol data

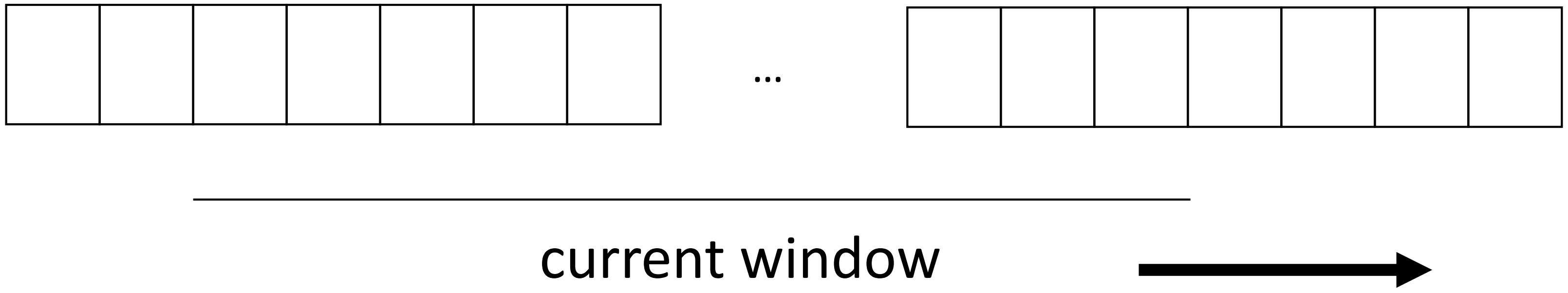


Recipient

- Assume AH header found
- Get SPI, destination address
- Find associated SA in SAD
  - If no associated SA, discard packet
- If antireplay not used
  - Verify IVC is correct
    - If not, discard

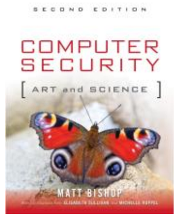
Recipient, Using Antireplay

- Check packet beyond low end of sliding window
- Check IVC of packet
- Check packet's slot not occupied
  - If any of these is false, discard packet



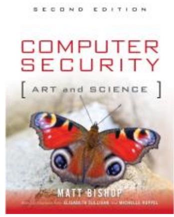
AH Cryptosystems

- RFCs say what algorithms must, should, may, must not be supported
- These change over time
    - Example: HMAC-MD5_96 acceptable before 2014; then deprecated, and now (October 2017) unacceptable
- Current (October 2017) list in RFC 8221



ESP Protocol

- Parameters in ESP header
  - SPI of SA applying protocol
  - Sequence number (anti-replay)
  - Generic “payload data” field
  - Padding and length of padding
    - Contents depends on ESP services enabled; may be an initialization vector for a chaining cipher, for example
    - Used also to pad packet to length required by cipher
  - Optional authentication data field



Sender

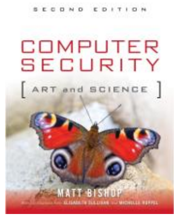
- Add ESP header
  - Includes whatever padding needed
- Encipher result
  - Do not encipher SPI, sequence numbers
- If authentication/integrity desired, compute as for AH protocol *except* over ESP header, payload and *not* encapsulating IP header

Recipient

- Assume ESP header found
- Use SPI, possibly protocol and destination address to find associated SA in SAD
  - If no associated SA, discard packet
- If authentication/integrity used
  - Do IVC, antireplay verification as for AH
    - Only ESP, payload are considered; *not* IP header
    - Note authentication data inserted after encipherment, so no deciphering need be done

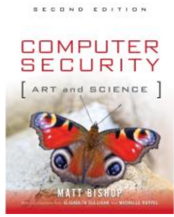
Recipient

- If confidentiality used
  - Decipher enciphered portion of ESP heaser
  - Process padding
  - Decipher payload
  - If SA is transport mode, IP header and payload treated as original IP packet
  - If SA is tunnel mode, payload is an encapsulated IP packet and so is treated as original IP packet



ESP Miscellany

- Must use at least one of confidentiality, authentication services
- Synchronization material must be in payload
  - Packets may not arrive in order, so if not, packets following a missing packet may not be decipherable
- Implementations of ESP assume symmetric cryptosystem
  - Implementations of public key systems usually far slower than implementations of symmetric systems
  - Not required



ESP Cryptosystems

- RFCs say what algorithms must, should, may, must not be supported
- These change over time
  - Example: DES in CBC mode acceptable before 2005; then deprecated, and as of August 2014 unacceptable
- Current (October 2017) list in RFC 8221

Which to Use: PGP, Signal, TLS, IPsec

- What do the security services apply to?
  - If applicable to one application *and* application layer mechanisms available, use that
    - PGP for electronic mail, Signal for instant messaging
  - If more generic services needed, look to lower layers
    - TLS for transport layer, end-to-end mechanism
    - IPsec for network layer, either end-to-end or link mechanisms, for connectionless channels as well as connections
  - If endpoint is host, TLS and IPsec sufficient; if endpoint is user, application layer mechanism such as PGP or Signal needed

Key PointsXXXXXXXX

- 12.3, authenticated encryption
- 12.5.2, Signal