Architectures and Design Methods for Cryptography

Jim Goodman
jimg@lumictech.com

Ingrid Verbauwhede
ingrid@ee.ucla.edu

ISSCC 2002 Tutorial
February 3, 2002
Outline

- Introduction
- Overview of cryptographic primitives
- Secret key algorithms
  ⇒ examples
  ⇒ implementation
- Public key algorithms
  ⇒ examples
  ⇒ implementation
- Security protocols
  ⇒ IPsec
  ⇒ SSL/TLS
Outline

- Introduction
  - Overview of cryptographic primitives
  - Secret key algorithms
    - examples
    - implementation
  - Public key algorithms
    - examples
    - implementation
  - Security protocols
    - IPsec
    - SSL/TLS
Motivation: Why all of the concern now?

- Yesterday: physical world = physical security
  - Hand-written signatures
  - Physical verification of identities
  - Communication channels isolated, hard to reach
  - Had to work a lot harder to both communicate securely and to eavesdrop

- Today: digital world = electronic security
  - Bits can be easily copied/deleted/added
  - Channels are easily accessible, tappable
  - Web increases your exposure to malicious adversaries
  - Stakes are a lot higher now!
How do we do it?

- Provide secure and trusted communications using the following primitives:

<table>
<thead>
<tr>
<th>Confidentiality</th>
<th>information/identity indecipherable to all but those authorized to know</th>
<th>public/private key encryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authentication</td>
<td>ensure user is who they say they are and that data origin is valid</td>
<td>digital signatures certificates/PKI</td>
</tr>
<tr>
<td>Integrity</td>
<td>ensure that data hasn’t been modified by unauthorized parties</td>
<td>hash functions (MACs)</td>
</tr>
<tr>
<td>Non-repudiation</td>
<td>associates data with an identity so that it cannot be denied</td>
<td>digital signatures</td>
</tr>
</tbody>
</table>
Outline

- Introduction

- Overview of cryptographic primitives
  - Secret key algorithms
    - examples
    - implementation
  - Public key algorithms
    - examples
    - implementation
  - Security protocols
    - IPsec
    - SSL/TLS
Encryption Algorithms

- Transforms input data (plaintext) into indecipherable output data (ciphertext) ⇒ transformation controlled by a key ⇒ transformation is invertible if key is known

- Symmetry of keys determines type of encryption ⇒ secret vs. public-key cryptography
Secret Key Cryptography

Encrypt and decrypt keys are the same
⇒ parties share a secret

Security is derived from shared secret
⇒ algorithms can be designed for efficiency

Two basic types of private key algorithms
⇒ block ciphers vs. stream ciphers
Inputs and outputs are $n$-bit blocks of data

Repeated iterations of a round function under influence of key values generated from $k$-bit key

Security is a function of key size

$\Rightarrow$ ideally security $\propto 2^{\text{key size}}$
Secret Key Cryptography: Stream Ciphers

- Generates pseudo-random keystream that is mixed (e.g., XORed) with data stream
- More error tolerant than block ciphers
- Keystream generation decoupled from data ⇒ pre-compute?
- Synchronization is crucial ⇒ self-synchronization
Public Key Cryptography

- Different encrypt and decrypt keys
  - encryption key known to all
  - decryption key kept secret

- Anyone can encrypt message but only the intended person can decrypt it

- Security based on number theoretic problems
  - algorithms can’t be tailored for efficiency

  RSA1024: 36.1 ops/sec (36.1 kbps)
  DES: ~921 kops/sec (59 Mbps)
Hash Functions & Message Authentication Codes

- Hash functions map arbitrary length objects into fixed-length outputs in a uniform manner
  \[ y = H(x) \]
  ⇒ e.g., MD5, SHA-1, RIPEMD-160, SHA-256/384/512

- MACs are hash functions with extra, keyed-inputs
  ⇒ key requirement provides authentication
  \[ y = H_K(x) \]
  ⇒ e.g., \( \text{HMAC}_K(M) = H[(K^+ \oplus \text{pad}_o) || H[(K^+ \oplus \text{pad}_i) || M]] \)

\[ K^+ = \begin{array}{c|c} 000 \ldots 000 & K \end{array} \]
\[ \text{pad}_i = \begin{array}{c} 00110110 \ldots 00110110 \end{array} \]
\[ \text{pad}_o = \begin{array}{c} 01011010 \ldots 01011010 \end{array} \]
Digital Signatures

- Relies on premise that each user possesses a unique secret value (e.g., private key)

- Exploit reversibility of public key algorithms:
  - $\Rightarrow$ encrypt using private key so other party can decrypt using your public key
  - $\Rightarrow$ since you know the private key, you must have generated the value, hence you must be who you claim to be

- RSA and Digital Signature Algorithm (DSA) are most popular examples
Outline

- Introduction
- Overview of cryptographic primitives
- **Secret key algorithms**
  - examples
  - implementation
- **Public key algorithms**
  - examples
  - implementation
- **Security protocols**
  - IPsec
  - SSL/TLS
DES (FIPS 46-3, 1999)

- DES originally proposed in 1977
  ⇒ replaced by AES this year but still in widespread use
  ⇒ strengthened via 3DES

key (K) = 56 key bits + 8 parity bits

plaintext (Pi) 64 64 ciphertext (Ci)

DES

key (K) = \{K1,K2\} = 112 key bits + 16 parity bits

plaintext (Pi) 64

K1

64

K1

DES

K2

64

DES\(^{-1}\)

K1

64

DES

3DES

64

ciphertext (Ci)
DES II: Feistel Cipher

- DES has 16 rounds + initial and final permutation
- Basic cipher structure is known as a Feistel cipher
  \[ \Rightarrow \text{other examples: IDEA, FEAL, and Kasumi} \]

- ASIC: encryption hardware = decryption hardware!
DES III: $f$ Function

$R_i$ $K_i$

Expansion

Expansion = 32b-to-48b permutation (wiring and duplication)

input of S-boxes is 8x6b = 48b

$+$

$S1 S2 S3 S4 S5 S6 S7 S8$

Si = 6b-to-4b non-linear substitution (ROM/logic-based LUT)

output of S-boxes is 8x4b = 32b

Permutation

Permutation = 32b-to-32b permutation (wiring)

$f(R_i, K_i)$
DES IV: Key Schedule

Initial Key

64

PC1

PC1: permute and drop 8 parity bits

56

28

C

28

D

C & D: rotate left 1 or 2 bits each round

DECRIPTION: rotate right

56

PC2

PC2: permute and select 48 output bits

56

48

Round Key \((K_i)\)

- Keys can be generated inline with above algorithm
  \(\Rightarrow\) alternative: pre-compute and store keys (96B)
Block Cipher Modes of Operation

- **Electronic code book (ECB)**
  - considered insecure (e.g., replay vulnerabilities, etc.)
  
  \[ P_i \rightarrow DES \rightarrow C_i \rightarrow DES^{-1} \rightarrow P_i \]

- **Cipher block chaining (CBC)**
  - used extensively in IPsec & SSL/TLS
  - feedback inhibits pipelining of ciphers
  - error propagation is a problem
  - variation with encryption-only for MAC generation

\[ \begin{align*}
  P_i & \rightarrow C_i^{-1} \rightarrow P_i \\
  C_i & \rightarrow DES \rightarrow C_i^-1 \rightarrow P_i
\end{align*} \]
Block Cipher Modes of Operation II

- Cipher feedback (CFB)
  ⇒ limits error propagation to 64-bit block
  ⇒ 1 bit CFB is self-synchronizing (errors for 65 bits)
  ⇒ important for wireless communications

![Diagram of Cipher Feedback (CFB) mode of operation]
Output feedback (OFB)

- Converts block cipher into a stream cipher
- No error propagation for any value of $r$
- Crucial to select unique IV’s to avoid replay attacks
Counter (CTR)

⇒ converts block cipher into a stream cipher
⇒ no feedback: enables high throughput via pipelining
⇒ crucial to choose non-repeating counter functions (e.g., LFSR’s)
⇒ crucial to choose counter IV’s that are unique
Block Cipher Modes of Operation V

<table>
<thead>
<tr>
<th></th>
<th>Sender</th>
<th>Receiver</th>
<th>Pipelining</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECB</td>
<td>Enc</td>
<td>Dec</td>
<td>Yes</td>
<td>not used</td>
</tr>
<tr>
<td>CBC</td>
<td>Enc</td>
<td>Dec</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>CBC-MAC</td>
<td>Enc</td>
<td>Enc</td>
<td>No</td>
<td>used for MAC</td>
</tr>
<tr>
<td>CFB</td>
<td>Enc</td>
<td>Enc</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>OFB</td>
<td>Enc</td>
<td>Enc</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>CTR</td>
<td>Enc</td>
<td>Enc</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

- Conclusion: most practical applications ONLY use encryption
  ⇒ important for AES because encryption is more efficient than decryption
  ⇒ area constraint applications (WLAN, IEEE802.11) can do with encryption-only hardware
AES (FIPS 197, 11/2001)

- Block cipher w/128b blocks
- Variable key size
  ⇒ 128/192/256 bits
  ⇒ key size dictates number of rounds (Nr):

<table>
<thead>
<tr>
<th>Key Size</th>
<th>Nr</th>
<th>Nk</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-128</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>AES-192</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>AES-256</td>
<td>14</td>
<td>8</td>
</tr>
</tbody>
</table>
AES II: Processing Subcircuits

- **ByteSub**: map to \( \text{GF}(2^8) \), invert, and permute result
  \[ \Rightarrow \text{utilize LUT that combines inversion and permutation} \]

  \[
  \begin{array}{cccc}
  a_0 & a_4 & a_8 & a_{12} \\
  a_1 & a_5 & a_9 & a_{13} \\
  a_2 & a_6 & a_{10} & a_{14} \\
  a_3 & a_7 & a_{11} & a_{15} \\
  \end{array}
  \]

  \[
  \begin{array}{cccc}
  b_0 & b_4 & b_8 & b_{12} \\
  b_1 & b_5 & b_9 & b_{13} \\
  b_2 & b_6 & b_{10} & b_{14} \\
  b_3 & b_7 & b_{11} & b_{15} \\
  \end{array}
  \]

  \[ a_i \xrightarrow{\text{ByteSub}} b_i \]

- **ShiftRow**: circularly rotate each row of state array
  \[ \Rightarrow \text{simple wiring} \]

  \[
  \begin{array}{cccc}
  a_0 & a_4 & a_8 & a_{12} \\
  a_1 & a_5 & a_9 & a_{13} \\
  a_2 & a_6 & a_{10} & a_{14} \\
  a_3 & a_7 & a_{11} & a_{15} \\
  \end{array}
  \]

  \[
  \begin{array}{cccc}
  a_0 & a_4 & a_8 & a_{12} \\
  a_5 & a_9 & a_{13} & a_1 \\
  a_{10} & a_{14} & a_2 & a_6 \\
  a_{15} & a_3 & a_7 & a_{11} \\
  \end{array}
  \]

\[ a_i \xrightarrow{\text{ShiftRow}} a_i \]
AES III: More Processing Subcircuits

- **MixColumn**
  \[ \Rightarrow \text{matrix multiplication of state array columns over } GF(2^8) \]

| \(a_0\) | \(a_4\) | \(a_8\) | \(a_{12}\) |
| \(a_1\) | \(a_5\) | \(a_9\) | \(a_{13}\) |
| \(a_2\) | \(a_6\) | \(a_{10}\) | \(a_{14}\) |
| \(a_3\) | \(a_7\) | \(a_{11}\) | \(a_{15}\) |

\[
\begin{bmatrix}
  b_i \\
  b_{i+1} \\
  b_{i+2} \\
  b_{i+3}
\end{bmatrix}
\begin{bmatrix}
  2 & 3 & 1 & 1 \\
  1 & 2 & 3 & 1 \\
  1 & 1 & 2 & 3 \\
  3 & 1 & 1 & 2
\end{bmatrix}
\begin{bmatrix}
  a_i \\
  a_{i+1} \\
  a_{i+2} \\
  a_{i+3}
\end{bmatrix}
\]

- **KeyAdd**: byte-by-byte XOR of round key and state
  \[ \Rightarrow b_i = a_i \oplus k_{j,i} \]
AES IV: Key Schedule

- Keys generated using 32b recurrences (32b = 1 col)
  - four 32b words grouped to form a round key
  - four different recurrences used:

  1. input key
  2. \( W[i-Nk] \oplus \text{ByteSub}(W[i-1]) \)
  3. \( W[i-Nk] \oplus W[i-1] \)
  4. \( W[i-Nk] \oplus \text{ByteSub}(	ext{RotByte}(W[i-1])) \oplus \text{Rcon}[i/Nk] \)

Round #

- 128b
- 192b
- 256b
ByteSub in GF((2^4)^2)

- Inversion over GF(2^8) requires large LUT’s
- Map to isomorphic field GF((2^4)^2) and invert there

Smaller (2.25x) but slower (1.67x)
Universal AES Architecture

- AES not Feistel: decryption is extra hardware

Encryption

Input Data

$K_0$ → KeyAdd

$K_i$ →

\[
\begin{align*}
\text{ByteSub} & \rightarrow \\
\text{ShiftRow} & \rightarrow \\
\text{MixColumn} & \rightarrow \\
\text{KeyAdd} & \rightarrow \\
\end{align*}
\]

Output Data

Decryption

Input Data

$K_{Nr}$ → KeyAdd$^{-1}$

$K_i$ →

\[
\begin{align*}
\text{ByteSub}^{-1} & \rightarrow \\
\text{ShiftRow}^{-1} & \rightarrow \\
\text{KeyAdd}^{-1} & \rightarrow \\
\text{MixColumn}^{-1} & \rightarrow \\
\end{align*}
\]

Output Data

Combined

Input Data

\[
\begin{align*}
\text{ByteSub} & \rightarrow \\
\text{ShiftRow} & \rightarrow \\
\text{MixColumn} & \rightarrow \\
\text{KeyAdd} & \rightarrow \\
\end{align*}
\]

Output Data

$\text{enc}$

$\text{first}$

$\text{regSel}$
Universal AES Processing Sub-circuits

**ByteSub**

- **In** → **perm** → **GF(2^8)-1** → **perm-1** → **Out**
- **enc**

**ShiftRow**

- **In[3:0]** → **Out[3:0]**
- **In[4,7,6,5]** → **Out[7:4]**
- **In[6,5,4,7]** → **Out[11:8]**
- **In[12,15,14,13]** → **Out[15:12]**
- **enc**

**MixColumn**

\[
\text{Out}_{col} = \begin{bmatrix}
2 & 3 & 1 & 1 \\
1 & 2 & 3 & 1 \\
1 & 1 & 2 & 3 \\
3 & 1 & 1 & 2
\end{bmatrix} \cdot \text{In}_{col} + \text{enc} \cdot \begin{bmatrix}
c & 8 & c & 8 \\
c & 8 & c & 8 \\
c & 8 & c & 8 \\
8 & c & 8 & c
\end{bmatrix} \cdot \text{In}_{col}
\]

**KeyAdd is its own inverse**
Universal AES Key Scheduler

- Decryption key = encryption key in reverse order
  \[ \Rightarrow \text{pre-compute keys and store in context (156/208/240B)} \]
  \[ \Rightarrow \text{reversible key generator with first/last keys in context} \]

- Key generation requires at most 8 32-bit variables (e.g., A, B, C, D, E, F, G, and H)
  \[ \Rightarrow \text{set of linear equations in A-H, invert to get decrypt keys} \]

128-bit Encrypt 128-bit Decrypt

<table>
<thead>
<tr>
<th>Initial Round Keys</th>
<th>Final Round Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>W[16] = A_1 = f(D_0) ^ A_0</td>
<td>W[15] = D_0 = C_1 ^ D_1</td>
</tr>
<tr>
<td>W[17] = B_1 = f(D_0) ^ A_0 ^ B_0</td>
<td>W[14] = C_0 = B_1 ^ C_1</td>
</tr>
<tr>
<td>W[18] = C_1 = f(D_0) ^ A_0 ^ B_0 ^ C_0</td>
<td>W[13] = B_0 = A_1 ^ B_1</td>
</tr>
<tr>
<td>W[19] = D_1 = f(D_0) ^ A_0 ^ B_0 ^ C_0 ^ D_0</td>
<td>W[12] = A_0 = f(C_1 ^ D_1) ^ A_1</td>
</tr>
</tbody>
</table>

128-bit Encrypt

<table>
<thead>
<tr>
<th>Initial Round Keys</th>
<th>Final Round Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>W[16] = A_1 = f(D_0) ^ A_0</td>
<td>W[15] = D_0 = C_1 ^ D_1</td>
</tr>
<tr>
<td>W[17] = B_1 = f(D_0) ^ A_0 ^ B_0</td>
<td>W[14] = C_0 = B_1 ^ C_1</td>
</tr>
<tr>
<td>W[18] = C_1 = f(D_0) ^ A_0 ^ B_0 ^ C_0</td>
<td>W[13] = B_0 = A_1 ^ B_1</td>
</tr>
<tr>
<td>W[19] = D_1 = f(D_0) ^ A_0 ^ B_0 ^ C_0 ^ D_0</td>
<td>W[12] = A_0 = f(C_1 ^ D_1) ^ A_1</td>
</tr>
</tbody>
</table>

128-bit Decrypt
Implementing Private Key Algorithms: ARC4

- Stream cipher used extensively in SSL/TLS
  ⇒ 256-byte state array (Key[ ])
  ⇒ two 8-bit pointers (x and y)

- Encrypt = Decrypt

```
for (# of bytes to be encrypted)
    x = (x + 1) & 0xff;
    Sx = Key[x];
    y = (y + Sx) & 0xff;
    Sy = Key[y];
    Key[x] = Sy;
    Key[y] = Sx;
    k = State[(Sx + Sy) & 0xff];
endfor
outByte = inByte ^ k;
```
ARC4 Implementation issues

- Maps well to smartcard-type 8-bit processing architectures
- Serialized algorithm ⇒ at most 8 bits per 5 cycles
  ⇒ difficult to reach high (Gbps) speeds
- Large key context
  ⇒ read 258 bytes, write 258 bytes
  ⇒ large amount of overhead for small blocks of data

```
0 x = (x + 1)_{256}, Sx = Key[x]
1 y = (y + Sx)_{256}, Sy = Key[y]
2 Key[x] = Sy
3 Key[y] = Sx
4 k = State[(Sx + Sy)_{256}]
5 x = (x + 1)_{256}, Sx = Key[x]
6 y = (y + Sx)_{256}, Sy = Key[y]
7 Key[x] = Sy
8 Key[y] = Sx
9 k = State[(Sx + Sy)_{256}]
```
Outline

- Introduction
- Overview of cryptographic primitives
- Secret key algorithms
  - examples
  - implementation
- Public key algorithms
  - examples
  - implementation
- Security protocols
  - IPsec
  - SSL/TLS
Types of Public Key Cryptography

- Integer Factorization (e.g., RSA)
  \[(M^e)^d \mod N = M^{e \cdot d} \mod N\]
  \[e \cdot d = 1 \mod \phi(N)\]
  \[M^{e \cdot d} \mod N = M^l \mod N = M\]

- Discrete Logarithms (e.g., Diffie-Hellman, DSA)
  \[(X^A)^B = X^{A \cdot B} = (X^B)^A \neq X^A \cdot X^B = X^{A + B}\]

- Elliptic Curve Discrete Logarithms (e.g., EC-DH)
  \[b \cdot (a \cdot P) = (ab) \cdot P = a \cdot (b \cdot P) \neq (a \cdot P) + (b \cdot P) = (a + b) \cdot P\]
Public Key Arithmetic Requirements

- Arithmetic requirements of IEEE 1363-2000:

<table>
<thead>
<tr>
<th></th>
<th>ADD</th>
<th>SUB</th>
<th>MULT</th>
<th>MOD</th>
<th>MOD_ADD</th>
<th>MOD_SUB</th>
<th>MOD_MULT</th>
<th>MOD_INV</th>
<th>MOD_EXP</th>
<th>GF_ADD</th>
<th>GF_MULT</th>
<th>GF_SQR</th>
<th>GF_INV</th>
<th>GF_EXP</th>
<th>EC_ADD</th>
<th>EC_DOUBLE</th>
<th>EC_MULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFEP-RSA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFDP-RSA</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFSP-RSA1</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFVP-RSA1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DLSVDP-DH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DLSVDP-MQV</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DLSP-DSA</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DLVP-DSA</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECSVDP-DH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECSVDP-MQV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECSP-DSA</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECVP-DSA</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Three primary finite fields used:
  - integers modulo-$N$ ($N$ either composite or prime - GF(p))
  - binary Galois Fields - GF($2^m$)
  - points on elliptic curves built upon GF($2^m$) and GF(p)
Modular Arithmetic

- Perform conventional arithmetic and then reduce results by dividing by modulus and returning remainder
  \[ A = \alpha \cdot N + \beta \]
  \(\text{modulus}\)
  \(\text{remainder}\)

- Very large operands
  \(\Rightarrow 512+\) (IF/DL) or 192+ (EC) bit operands

- Modular addition/subtraction simple
  \(\Rightarrow\) use either wide comparators or carry/borrow-outs

- Modular inversion rarely used in IF/DL
  \(\Rightarrow S/W: \) extended euclidean algorithm (mults/divs)
  \(\Rightarrow H/W: \) extended binary euclidean algorithm (adds/shifts), requires fast modular addition/subtraction
Modular Multiplication

- Critical operation for IF and DL-GF(p)

- Conventional approach has two options
  - multiply and then divide to get remainder
  - interleave multiply and divide during PP accumulation

- Interleaved reduction
  - estimate quotient using $m$ MSB of PP and modulus
  - trade-off error in estimation vs. speed of estimation

- Maps to very regular VLSI architecture
  - n-bit wide PP accumulator w/quotient estimate unit
  - redundant representation for PP (e.g., carry-save)
Montgomery Reduction/Multiplication

- Montgomery’s idea: right shift value $n$ times to turn $2n$-bit number into $n$-bit number
  $\Rightarrow$ add multiples of $N$ to zero $n$ LSB so no information is destroyed
  $\Rightarrow$ adding multiples of $N$ doesn’t change value modulo-$N$

\[
\text{MontMult}(A,B,N) = A \cdot B \cdot 2^{-n} \mod N
\]
\[
\text{MontRed}(A,N) = A \cdot 2^{-n} \mod N
\]

- Requires pre- and post-computation to eliminate constant factors

\[
\text{MontMult}(A,2^{2n},N) = A \cdot 2^{2n} \cdot 2^{-n} \mod N = A \cdot 2^n \mod N
\]
\[
\text{MontMult}(A_0 \cdot 2^n, A_1 \cdot 2^n, N) = A_0 \cdot 2^n \cdot A_1 \cdot 2^n \cdot 2^{-n} \mod N = A_0 A_1 \cdot 2^n \mod N
\]
\[
\text{MontMult}(A \cdot 2^n, 1, N) = A \cdot 2^n \cdot 2^{-n} \mod N = A \mod N
\]
Montgomery Explained

- **Basic algorithm**

  ```
  for (i=0; i<n/k; i++)
    compute q_i
    S_i = (S_{i-1} + a_iB + q_iN)/2^k
  endfor
  ```

- **Need to zero $k$ LSB of partial product**

  - $S_i + a_iB + q_iN = 0 \mod 2^k$
  - $(S_i + a_iB) \mod 2^k = -N \cdot q_i \mod 2^k$
  - $q_i = (S_i + a_iB) \cdot (-N^{-1} \mod 2^k) \mod 2^k$
  - $q_i = (S_i + a_iB)N_{inv} \mod 2^k$

  Where (assuming $k \leq 4$):
  
  $N_{inv} = \{n_3, n_2, n_1, n_0\}$
  
  $= \{!(n_3 \wedge n_2 \wedge n_1), !n_2, !n_1, 1\}$

- **Reduction optimization**

  - $A = \{a_{2n-1}, \ldots, a_n, a_{n-1}, \ldots, a_0\} = A_{High} \cdot 2^n + A_{Low}$
  - $MontRed(A, N) = A \cdot 2^{-n} \mod N$
  - $= (A_{High} + A_{Low} \cdot 2^{-n}) \mod N$
  - $= (A_{High} + MontRed(A_{Low}, N)) \mod N$
High-radix Montgomery Optimizations

for (i=0; i<n; i++)
  qi = (Si + aiB) mod 2
  Si = (Si + aiB + qiN) / 2
endfor

utilize radix-$2^k$ arithmetic

for (i=0; i<n/k; i++)
  qi = (Si + aiB) $N_{inv}$ mod $2^k$
  Si = (Si + aiB + qiN) / $2^k$
endfor

use ($B \cdot 2^k$) instead of $B$
(add iteration to undo scaling)

 defer multiplication by $N_{inv}$ to $S_i$ calculation step ($N_{new} = N_{inv} \cdot N$)

for (i=0; i<n+1; i++)
  qi = $S_i$ mod $2^k$
  $S_i = (S_i + qiN_{new}) / 2^k + aiB$
endfor

- Maps well to systolic architectures

S = $A \cdot (B \cdot 2^k) \cdot 2^{-(n/k+1)k}$ mod $N$
= $A \cdot B \cdot 2^{-n}$ mod $N$

trivial $q_i$ computation
requires pre-computation of $N_{new} = N_{inv} \cdot N$
result in range [0, $N \cdot 2^k$), requires post-processing

requires n cycles for n-bit multiplication

requires ~$(n/k)$ cycles for n-bit multiplication
$q_i$ computation requires time-consuming multiply

$S = A \cdot (B \cdot 2^k) \cdot 2^{-(n/k+1)k}$ mod $N$
= $A \cdot B \cdot 2^{-n}$ mod $N$
Modular Exponentiation

- Repeated square-and-multiply

  \[ x^a \cdot x^b = x^{a+b} \]

  \[ x^a \cdot x^a = x^{2a} \]

  \[ \Rightarrow \text{LSB-first} \]

  \[
  \begin{align*}
  A_0 &= A; \\
  P_0 &= 1; \\
  \text{for} \ (i=1; \ i<=n; \ i++) \\
  &\quad A_i = A_{i-1}^2 \mod N \\
  &\quad \text{if} \ (\exp_{i-1}) \\
  &\quad \quad P_i = P_{i-1} \cdot A_{i-1} \mod N \\
  \text{endfor}
  \end{align*}
  \]

  can parallelize operations

  uses two intermediate variables

  \[ \Rightarrow \text{MSB-first} \]

  \[
  \begin{align*}
  P_0 &= 1; \\
  \text{for} \ (i=n-1; \ i>=0; \ i--) \\
  &\quad P_i = P_{i-1}^2 \mod N \\
  &\quad \text{if} \ (\exp_i) \\
  &\quad \quad P_i = P_i \cdot A \mod N \\
  \text{endfor}
  \end{align*}
  \]

  operations must be serialized

  only one intermediate variable

- Radix-2\(^m\) exponent scanning with pre-computation

  \[ \Rightarrow \text{pre-compute} \ \{A^0 \mod N, A^1 \mod N, \ldots\} \]

  \[ \Rightarrow m \text{ squarings} + 1 \text{ multiplication per } m \text{ bits of exponent} \]
Chinese Remainder Theorem

- Perform Modulo-$N$ ($N = pq$) arithmetic using modulo-$p$ and modulo-$q$ arithmetic

\[
\begin{align*}
  y &= p^{-1} \mod q \quad \text{// ~n cycle pre-computation} \\
  ep &= E \mod (p-1) \quad \text{// n/2 cycles} \\
  eq &= E \mod (q-1) \quad \text{// n/2 cycles} \\
  x_p &= A^{ep} \mod p \quad \text{// (n/2)^2 = n^2/4 cycles} \\
  x_q &= A^{eq} \mod q \quad \text{// (n/2)^2 = n^2/4 cycles} \\
  x &= (x_q - x_p) \mod q \quad \text{// 2 cycles} \\
  x &= x \cdot p \mod q \quad \text{// n/2 cycles} \\
  P &= x_p + x \cdot p \quad \text{// total ~ (n^2/2 + 5n/2 + 2) cycles}
\end{align*}
\]

- 2x faster using same hardware
  \implies 4x faster using half-sized parallel units

- Extends precision of existing hardware

- Requires factorization of $N$
Binary Galois Field Arithmetic

- Extend GF(2) into \( m \)-dimensional vector field GF\( (2^m) \) ⇒ carry-free arithmetic (addition = XOR)

- Basis choice dictates implementation
  ⇒ most common: polynomial/standard basis
  ⇒ others: optimal normal basis, dual basis, etc.

- Polynomial/canonical/standard basis
  ⇒ vectors are \((m-1)\)-th degree poly. with binary coefficients
  ⇒ field defined by primitive characteristic polynomial

\[
F(x) = x^m + x^{kn} + \ldots + x^{k1} + 1 = 0
\]

⇒ \( x^m = x^{kn} + \ldots + x^{k1} + 1 \)

- Characteristic polynomial defines complexity
  ⇒ trinomials and pentanomials commonly used
GF$\left(2^m\right)$ Multiplication

- Two basic types derived from basic bit-serial polynomial multiplication algorithm

\[ C(x) = A(x)B(x) \mod F(x) \]

\[
C(x) = \sum x^i [b_{n-i-1}A(x)]_{F(x)} = \sum b_i [x^iA(x)]_{F(x)}
\]
GF($2^m$) Inversion

- Factor group order and exponentiate (normal basis)
  
  \[ A^{-1} = A^{2^m-2}, \quad \forall A \in GF(2^m) \]

- Extended binary Euclidean algorithm (standard basis)
  
  \[ \Rightarrow \text{similar to modular inversion} \]
  \[ \Rightarrow \text{can perform inversion + multiplication concurrently} \]
  \[ \Rightarrow \text{requires shifts, XORs, magnitude comparison} \]

```plaintext
while (A != 0)
  while (A0 = 0)
    A = A/2
    B = (B + B0·N)/2
  endif
  if (A >= C)
    A = A + B
    C = C + D
  else
    B = A + B
    C = C + D
  endif
endwhile
```

```plaintext
while (C0 = 0)
  C = C/2
  D = (D + D0·N)/2
endwhile
```
GF(2^m) Squaring & Exponentiation

- GF(2^m) squaring can be optimized
  ⇒ cross terms drop out due to GF(2) addition
  ⇒ squaring over GF(2) is a NOP

  \[ C(x) = (a_2x^2 + a_1x^1 + a_0x^0)^2 \mod F(x) \]
  \[ = (a_2^2x^4 + 2a_2a_1x^3 + a_1^2x^2 + 2a_1a_0x^1 + a_0^2) \mod F(x) \]
  \[ = (a_2x^4 + a_1x^2 + a_0) \mod F(x) \]

  ⇒ reconfigurable logic enables bit-parallel squaring
  ⇒ LUT-based approach very efficient in S/W

- GF(2^m) exponentiation seldom used
  ⇒ lack of S/W support for GF(2^m) arithmetic
  ⇒ use same basic approaches as modular exponentiation
Unified Field Arithmetic: ALU Support

- Full adder = 2-input GF($2^m$) adder if carry disabled

- $\sim 10x$ performance improvement for multiplication vs. generic RISC
Unified Field Arithmetic: Custom H/W I

- Montgomery Multiplication
  \[ (P_c, P_s)_j = \frac{(P_c, P_s)_{j-1} + b_jA + q_jN}{2} \]

\[ (P_c, P_s)_{i-1} = (P_c, P_s)_i + b_jA_i + q_jN_i \]

- MSB-first GF(2^m) Multiplication
  \[ P_j = 2P_{j-1} + b_jA + q_jN \]

\[ P_{ci} = P_{ci-1} + b_jA_i + q_jN_i \]
Unified Field Arithmetic: Custom H/W II

- GF($2^m$) Inversion
  ⇒ extended binary euclidean algorithm
  ⇒ at completion $A = a/b$

```
Pc = b (field element)
Ps = N (field polynomial in binary form)
A = 0
B = a (field element)
while (Pc != 0)
  while (Pc0 == 0)
    Pc = Pc/2
    B = (B + B0·N)/2
  endif
  if (Pc >= Ps)
    Pc = Pc + Ps
    B = A + B
  else
    Ps = Pc + Ps
    A = A + B
  endif
endwhile
```

Same H/W as unified multiplier
Elliptic Curves

- Given a field $F$, an elliptic curve $E$, is defined to be those points $(x, y)$, $\{x, y\}$ in $F$, satisfying the generalized Weisterass Equation:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

- Depending on the field, you can simplify the equation:
  e.g., $GF(2^m)$: $y^2 + xy = x^3 + a_2x^2 + a_6$

- Points on curve represented in two common forms
  $\Rightarrow$ affine: $(x, y)$
  $\Rightarrow$ projective: $(x, y, z)$

- Much smaller operand sizes (e.g., 177b vs. 1024b)
  $\Rightarrow$ 177b (EC-GF($2^m$)) vs. 1024b (RSA)
  $\Rightarrow$ well-suited to FPGA-based implementations
Affine Elliptic Curve Arithmetic over $GF(2^m)$

- **Three basic operations**
  
  $P_3 = P_1 + P_2 = (x_3,y_3)$
  
  $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a_2$
  
  $y_3 = (x_2 + x_3)\lambda + x_3 + y_2$
  
  $\lambda = \frac{y_1 + y_2}{x_1 + x_2}$

- **Point Addtion**
  
  $P_3 = P_1 + P_2$

- **Point Negation**
  
  $-P = -(x, y)$
  
  $y = \lambda x + \mu$
  
  $-P = (x, y + a_1 x)$

- **Point Doubling**
  
  $P_3 = 2P_1 = (x_3,y_3)$
  
  $x_3 = \lambda^2 + \lambda + a_2$
  
  $y_3 = (x_1 + x_3)\lambda + x_3 + y_1$
  
  $\lambda = x_1 + y_1/x_1$

- **Formulae computed over base field**
  
  $\Rightarrow$ requires $GF(2^m)$ add, multiply, square, divide/invert
Affine vs. Projective Arithmetic over GF(2^m)

### Projective Point Addition

\[
P_3 = P_1 + P_2 = (x_3, y_3, z_3)
\]

\[
\lambda_1 = x_1z_2^2
\]

\[
\lambda_2 = x_2z_1^2
\]

\[
\lambda_3 = \lambda_1 + \lambda_2
\]

\[
\lambda_4 = y_1z_2^3
\]

\[
\lambda_5 = y_2z_1^3
\]

\[
\lambda_6 = \lambda_4 + \lambda_5
\]

\[
\lambda_7 = z_1\lambda_3
\]

\[
\lambda_8 = \lambda_6x_2 + \lambda_7y_2
\]

\[
z_3 = \lambda_7z_2
\]

\[
\lambda_9 = \lambda_6 + z_3
\]

\[
x_3 = a_2z_3^2 + \lambda_6\lambda_9 + \lambda_3^3
\]

\[
y_3 = \lambda_9x_3 + \lambda_9\lambda_7^2
\]

### Projective Point Doubling

\[
P_3 = 2\cdot P_1 = (x_3, y_3, z_3)
\]

\[
z_3 = x_1z_1^2
\]

\[
d_6 = a_6^{2^n-2}
\]

\[
x_3 = (x_1 + d_6z_1^2)^4
\]

\[
\lambda = z_3 + x_1^2 + y_1z_1
\]

\[
y_3 = x_1^4z_3 + \lambda x_3
\]

<table>
<thead>
<tr>
<th></th>
<th>affine</th>
<th>projective</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>I + 2M + S</td>
<td>15M + 5S</td>
</tr>
<tr>
<td>doubling</td>
<td>I + 2M + S</td>
<td>5M + 5S</td>
</tr>
</tbody>
</table>

Inversion cost and amount of available memory dictate which representation should be used.
Elliptic Curve Arithmetic: Multiplication

- Analog of modular exponentiation using repeated point doubling and addition
  \[ \Rightarrow \text{radix-}2^m: \text{extend point doubling to quadrupling, etc.} \]

- Simple to negate points
  \[ \Rightarrow \text{booth encoding of multiplication} \]
  \[ \Rightarrow \text{non-adjacent form (NAF) representation of multiplier} \]

| signed NAF: n doublings + \(-\frac{n}{3}\) additions |
| radix-2 binary: n doublings + \(-\frac{n}{2}\) additions |
Outline

- Introduction
- Overview of cryptographic primitives
- Secret key algorithms
  - examples
  - implementation
- Public key algorithms
  - examples
  - implementation
- Security protocols
  - IPsec
  - SSL/TLS
IPsec Overview

- Datapath-oriented security protocol
  ⇒ connections have “long” lifetime
  ⇒ throughput is the key metric

- Situated at the network level of the protocol stack

- Two modes: tunnel and transport

![Diagram showing two modes of IPsec: tunnel and transport.](image-url)
IPsec: Authentication Header (AH)

- Provides authentication across entire IP packet
  ⇒ mutable fields zeroized during authentication

<table>
<thead>
<tr>
<th></th>
<th>Transport Mode</th>
<th>Tunnel Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP Header</td>
<td>Original IP Header with modifications to length, TTL, protocol, and checksum fields</td>
<td>Tunnel header from SADB, formatted to reflect current packet characteristics</td>
</tr>
<tr>
<td>Payload</td>
<td>Original payload</td>
<td>Entire original IP packet</td>
</tr>
</tbody>
</table>
IPsec: Encapsulating Security Payload (ESP)

- Provides inline confidentiality and authentication

![Diagram showing IPsec ESP structure](image)

1) Encrypt
2) Authenticate

<table>
<thead>
<tr>
<th>Transport Mode</th>
<th>Tunnel Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP Header</td>
<td>Tunnel header from SADB, formatted to reflect current packet characteristics</td>
</tr>
<tr>
<td>Original IP Header with modifications to length, TTL, protocol, and checksum fields</td>
<td>Entire original IP packet</td>
</tr>
<tr>
<td>Payload</td>
<td>Original payload</td>
</tr>
</tbody>
</table>

If nested, ESP performed first, followed by AH
IPsec: Internet Key Exchange (IKE)

- Setup is done relatively rarely
  ⇒ “Set and forget.”
  ⇒ primarily stressed during power-up and recovery

- IKE used to establish Security Associations (SA’s)
  ⇒ standard/aggressive/quick modes

- Varied processing requirements
  ⇒ DH over GF(p), EC-GF(p), and EC-GF(2^m) used for secret sharing
  ⇒ DSA and RSA used for authentication
  ⇒ HMAC-SHA-1/MD5 used for key generation
IPsec Processing Flows and Requirements

- **Outbound**

- **Inbound**

- **ESP and AH dataflows @ 1Gbps rate:**

![Graphs showing Encrypt/Hash Rate, Context Bandwidth, and Record Size for different encryption algorithms.]
Outline

- Introduction
- Overview of cryptographic primitives
- Secret key algorithms
  - examples
  - implementation
- Public key algorithms
  - examples
  - implementation
- Security protocols
  - IPsec
  - SSL/TLS
SSL/TLS Overview

- Connection-oriented
  ⇒ setting up and tearing down temporary connections
  ⇒ developed for web applications

- Sits between TCP and HTTP in protocol stack

- TLS = new version of SSL
  ⇒ uses HMAC
  ⇒ slight changes to handshake protocol & key generation

- Two components
  ⇒ handshake protocol: establishes connection and keys
  ⇒ record layer: encrypts/decrypts and authenticates datagrams between parties
SSL/TLS Handshake Protocol

- Frequent setups required (e.g., secure web transfer)
  ⇒ connections tend to be short-lived
  ⇒ simple protocol developed to ensure efficiency

- Varied processing requirements
  ⇒ DH over GF(p) used for shared secret generation
  ⇒ RSA used for encrypting self-generated secrets
  ⇒ SHA-1/MD5 used for key generation
    (0.4 - 1.3 Gbps @ 10,000 setups/second)
SSL/TLS Record Processing

- Outbound: compute MAC first, then encrypt
  ⇒ inbound reverses operations

- Algorithms used:
  ⇒ encrypt: DES, 3DES, AES, ARC4
  ⇒ MAC: SSL-MAC and HMAC (SHA-1, MD5)
SSL/TLS Processing Flows & Requirements

- **Outbound**
  
  - Header Processor → MAC Processor → Enc/Dec Processor → MAC Processor → Header Processor

- **Inbound**
  
  - Header Processor → Enc/Dec Processor → MAC Processor → Header Processor

- **Record protocol @1Gbps rate:**

  - Encryption Rate (Gbps)
    - MD5/SHA-1
    - 3DES
    - AES
    - ARC4

  - Context Bandwidth (Gbps)
    - ARC4
    - AES
    - 3DES
References


References (continued)


References (continued)


References (continued)


References (continued)


References (continued)


References (continued)


References (continued)


References (cont.)

