Using a block cipher in practice

-- Electronic Codebook (ECB) mode
-- Cipher Block Chaining (CBC) mode
-- Cipher Feedback (CFB) mode
-- Output Feedback (OFB) mode
-- Counter (CTR) mode
-- attacks on CBC

(c) Levente Buttyán (buttyan@crysys.hu)

ECB mode

- **encrypt**
  \[
  \begin{align*}
  K &\rightarrow E \\
  X_1 &\rightarrow Y_1 \\
  X_2 &\rightarrow Y_2 \\
  \ldots &\rightarrow \ldots \\
  X_N &\rightarrow Y_N \\
  \end{align*}
  \]

- **decrypt**
  \[
  \begin{align*}
  K &\rightarrow D \\
  Y_1 &\rightarrow X_1 \\
  Y_2 &\rightarrow X_2 \\
  \ldots &\rightarrow \ldots \\
  Y_N &\rightarrow X_N \\
  \end{align*}
  \]
Properties of ECB mode

- encrypting the same plaintext with the same key results in the same ciphertext
- identical plaintext blocks result in identical ciphertext blocks (under the same key of course)
  - messages to be encrypted often have very regular formats
  - repeating fragments, special headers, string of 0s, etc. are quite common
- blocks are encrypted independently of other blocks
  - reordering ciphertext blocks result in correspondingly reordered plaintext blocks
  - ciphertext blocks can be cut from one message and pasted in another, possibly without detection
- error propagation: one bit error in a ciphertext block affects only the corresponding plaintext block (results in garbage)
- overall: not recommended for messages longer than one block, or if keys are reused for more than one block

CBC mode

- encrypt

- decrypt
General properties of CBC mode

- encrypting the same plaintexts under the same key, but different IVs result in different ciphertexts
- ciphertext block $Y_j$ depends on $X_j$ and all preceding plaintext blocks
  - rearranging ciphertext blocks affects decryption
  - however, dependency on the preceding plaintext blocks is only via the previous ciphertext block $Y_{j-1}$
  $\Rightarrow$ proper decryption of a correct ciphertext block needs a correct preceding ciphertext block only (cut-and-paste attacks!)
- error propagation:
  - one bit error in a ciphertext block $Y_j$ has an effect on the $j$-th and $(j+1)$-st plaintext block
    - $X'_j$ is complete garbage and $X_{j+1}'$ has bit errors where $Y_j$ had
    - an attacker may cause predictable bit changes in the $(j+1)$-st plaintext block!
- self-synchronizing property:
  - automatically recovers from loss of a ciphertext block

An example for the cut-and-paste attack

$C \rightarrow S$: password:kis\text{acsa}$

$S \rightarrow C$: http://www.crysyl\text{ex.html}$

http://www.crysyl\text{ex.html}$

Using a block cipher in practice
### Protection of the IV in CBC mode

- the IV need not be secret (although secret IVs have some advantages), but its integrity should be protected
  - malicious modification of the IV allows an attacker to make predictable changes to the first plaintext block recovered
- one solution is to send the IV in an encrypted form at the beginning of the CBC encrypted message
  - this may be changed by an attacker, but he cannot control the effects of his changes made in the IV

### Padding

- the length of the message may not be a multiple of the block size of the cipher
- one can add some extra bytes to the short end block until it reaches the correct size – this is called padding
- a typical example is when the last byte indicates the number of padding bytes added – this allows the receiver to remove the padding

```
short end block   padding   padding length
| | | | | x04
```

- another approach is to add an x01 byte and then as many x00 bytes as needed
Using a block cipher in practice

**Example: TLS Record Protocol**

- **TLS padding:**
  - last byte is the length n of the padding (not including the last byte)
  - all padding bytes have value n
  - examples for correct padding: x00, x01x01, x02x02x02, ...
  - verification: if the last byte is n, then verify if the last n+1 bytes are all n
  - if verification is successful, remove the last n+1 bytes, and proceed with the verification of the MAC

<table>
<thead>
<tr>
<th>type</th>
<th>version</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A potential problem with CBC encryption

- **padding oracle**
  - assume that a system uses CBC encryption/decryption with MAC and padding (in this order!)
  - the receiver of a CBC encrypted message may respond differently in the case of “incorrect padding” and in the case of “correct padding but incorrect MAC”
  - we get 1 bit of information!

- **example padding oracle in practice: a TLS server**
  - send a random message to a TLS server
  - the server will drop the message with overwhelming probability
    - either the padding is incorrect (the server responds with a DECRYPTION_FAILED alert)
    - or the MAC is incorrect with very high probability (the server responds with BAD_RECORD_MAC)

- **how to exploit this?**
  - an attack discovered by Vaudenay in 2002 uses such a padding oracle to decrypt any CBC encrypted message efficiently!
  - vulnerable protocols: SSL/TLS, WTLS, IPsec, …
Attack explained: Last byte(s) oracle

- assume we have an encrypted block \(y_1y_2...y_8 = E_K(x_1x_2...x_8)\)
- we want to compute \(x_8\) (the last byte of \(x\))
- idea:
  1. choose a random block \(r_1r_2...r_8\); let \(i = 0\)
  2. send \(r_1r_2...r_8(r_8 \oplus i)y_1y_2...y_8\) to the server (oracle)
  3. if there’s a padding error, then increment \(i\) and go back to step 2
  4. if there’s no padding error, then \(r_8 \oplus x\) ends with 0 or 11 or 222 ...
     - the most likely is that \((r_8 \oplus i) \oplus x_8 = 0\), and hence \(x_8 = r_8 \oplus i\)

\[
\begin{array}{c}
\text{Assume we have an encrypted block } y_1y_2...y_8 = E_K(x_1x_2...x_8) \\
\text{We want to compute } x_8 \text{ (the last byte of } x) \\
\text{Idea:}
\end{array}
\]

\[
\begin{array}{c}
\text{1. Choose a random block } r_1r_2...r_8; \text{ let } i = 0 \\
\text{2. Send } r_1r_2...r_8(r_8 \oplus i)y_1y_2...y_8 \text{ to the server (oracle)} \\
\text{3. If there's a padding error, then increment } i \text{ and go back to step 2} \\
\text{4. If there's no padding error, then } r_8 \oplus x \text{ ends with 0 or 11 or 222 ...} \\
\text{• The most likely is that } (r_8 \oplus i) \oplus x_8 = 0, \text{ and hence } x_8 = r_8 \oplus i
\end{array}
\]

\[
\begin{array}{c}
\text{Using a block cipher in practice}
\end{array}
\]

\[
\begin{array}{c}
\text{Attack explained: Last byte(s) oracle}
\end{array}
\]

- assume we get that \(r \oplus x\) has a correct padding, but we don’t know if it is 0 or 11 or 222 ...
- algorithm:
  1. let \(j = 1\)
  2. change \(r_j\) and send \(r_1r_2...r_8x_1x_2...x_8\) to the server again
  3. if the padding is still correct then the \(j\)-th byte was not a padding byte; increment \(j\)
     and go back to step 2
  4. if the padding becomes incorrect then the \(j\)-th byte was the first padding byte;
     \(x_j \oplus r_j \oplus x_{j+1} \oplus r_{j+1} \oplus ... \oplus x_8 \oplus r_8 = (8-j) \oplus ... \oplus (8-j)\) and hence
     \(x_j x_{j+1} ... x_8 = r_j \oplus (8-j) r_{j+1} \oplus (8-j) ... r_8 \oplus (8-j)\)

\[
\begin{array}{c}
\text{x = DE AD BE EF DE AD BE EF} \\
r = 01 23 45 67 DD AE BD EC \\
r \oplus x = DF BE FB 88 03 03 03 03 \\
\text{Padding:} \\
\text{1 00 23 45 67 DD AE BD EC} \\
\text{DE BE FB 88 03 03 03 03 OK} \\
\text{2 00 22 44 67 DD AE BD EC} \\
\text{DE 8F FB 88 03 03 03 03 OK} \\
\text{3 00 22 44 67 DD AE BD EC} \\
\text{DE 8F FA 88 03 03 03 03 OK} \\
\text{4 00 22 44 66 DD AE BD EC} \\
\text{DE 8F FA 89 03 03 03 03 OK} \\
\text{5 00 22 44 66 DC AE BD EC} \\
\text{DE 8F FA 89 02 03 03 03 ERROR} \\
\text{x_5 x_6 x_7 x_8 = DD \oplus 03 AD \oplus 03 BD \oplus 03 EC \oplus 03 = DE AD BE EF}
\end{array}
\]
Attack explained: Block decryption oracle

- assume we have an encrypted block $y_1y_2...y_8 = E_k(x_1x_2...x_8)$ and we know the value of $x_{j+1}...x_8$ (using the last byte(s) oracle)
- we want to compute $x_{j-1}$
- algorithm:
  1. choose a random block $r_1r_2...r_8$ such that
     $r_j = x_j \oplus (9-j); r_{j+1} = x_{j+1} \oplus (9-j); ... r_8 = x_8 \oplus (9-j)$;
  2. let $i = 0$
  3. send $r_1r_2...r_{j-2}(r_{j-1} \oplus i)r_{j}...r_8y_1y_2...y_8$ to the server (oracle)
  4. if there’s a padding error then increment $i$ and go back to step 3
  5. if there’s no padding error then $x_{j-1} = r_{j-1} \oplus i = 9-j$ and hence

```
x = DE AD BE EF DE AD BE EF
r = 01 23 45 67 DA A9 BA EB
r \oplus x = DF 8E FB 88 04 04 04 04
```

```
i 0 01 23 45 67 DA A9 BA EB 01 23 45 67 DA A9 BA EB 00 00 00 00 00 ERROR
1 01 23 45 66 DA A9 BA EB 01 23 45 66 DA A9 BA EB 00 00 00 00 00 ERROR
... ... ... ... ... ... ... ... ...
140 01 23 45 EB DA A9 BA EB 01 23 45 EB DA A9 BA EB 00 00 00 00 00 OK
x_4 = EB \oplus 04 = EF
```

Using a block cipher in practice

13

Attack explained: Decryption oracle

- assume we have a CBC encrypted message $(Y_1, Y_2, ..., Y_N)$ where
  - $Y_1 = E_k(X_1 \oplus IV)$
  - $Y_i = E_k(X_i \oplus Y_{i-1})$ (for $1 < i < N$)
  - $Y_N = E_k([X_N|pad|plen] \oplus Y_{N-1})$
- we want to compute $X_1, X_2, ... X_N$
- algorithm:
  - decrypt $Y_n$ using the block decryption oracle and XOR the result to $Y_{N-1}$; you get $X_n|pad|plen$
  - decrypt $Y_i$ using the block decryption oracle and XOR the result to $Y_{i-1}$; you get $X_i$
  - decrypt $Y_1$ using the block decryption oracle and XOR the result to IV; you get $X_0$ (if IV is secret you cannot get $X_0$)
- complexity of the whole attack:
  - on average we need only $\frac{1}{2}256*8*N = 1024*N$ oracle calls!
Using a block cipher in practice

**CFB mode**

- encrypt

  
  \[
  \text{shift register (n)} \rightarrow E \rightarrow m_i \rightarrow c_i
  \]

  - initialized with IV
  - select s bits

- decrypt

  
  \[
  \text{shift register (n)} \rightarrow E \rightarrow c_i \rightarrow m_i
  \]

  - initialized with IV
  - select s bits

**Properties of CFB mode**

- encrypting the same plaintexts under the same key, but different IVs results in different ciphertexts
- the IV can be sent in clear
- ciphertext character \( c_j \) depends on \( m_j \) and all preceding plaintext characters
  - rearranging ciphertext characters affects decryption
  - proper decryption of a correct ciphertext character needs the preceding n/s ciphertext characters to be correct
- error propagation:
  - one bit error in a ciphertext character \( c_j \) has an effect on the decryption of that and the next n/s ciphertext characters (the error remains in the shift register for n/s steps)
    - \( m_j \) has bit errors where \( C_j \) had, all the other erroneous plaintext characters are garbage
    - an attacker may cause predictable bit changes in the j-th plaintext character
- self-synchronizing property:
  - recovers from loss of a ciphertext character after n/s steps
OFB mode

- encrypt

\[
\begin{array}{c}
\text{shift register (n)} \\
\text{K} \\
\text{select } s \text{ bits} \\
\end{array}
\]
\[
\begin{array}{c}
m_i \\
\oplus \\
c_i \\
\end{array}
\]

initialized with IV

- decrypt

\[
\begin{array}{c}
\text{shift register (n)} \\
\text{K} \\
\text{select } s \text{ bits} \\
\end{array}
\]
\[
\begin{array}{c}
c_i \\
\oplus \\
m_i \\
\end{array}
\]

initialized with IV

Properties of OFB mode

- a different IV should be used for every new message, otherwise messages will be encrypted with the same key stream
- the IV can be sent in clear
  - however, if the IV is modified by the attacker, then the cipher will never recover (unlike CFB)
- ciphertext character \( c_j \) depends on \( m_j \) only (does not depend on the preceding plaintext characters)
  - however, rearranging ciphertext characters affects decryption
- error propagation:
  - one bit error in a ciphertext character \( c_j \) has an effect on the decryption of only that ciphertext character
    - \( m_j' \) has bit errors where \( c_j \) had
    - an attacker may cause predictable bit changes in the \( j \)-th plaintext character !!!
- needs synchronization
  - cannot automatically recover from a loss of a ciphertext character
CTR mode and its properties

- **encrypt**
  
  \[ K \rightarrow E \rightarrow \text{select s bits} \rightarrow m_i \rightarrow \oplus \rightarrow c_i \]

- **decrypt**
  
  \[ c_i \rightarrow E \rightarrow \text{select s bits} \rightarrow m_i \rightarrow \oplus \rightarrow c_i \]

- similar to OFB, but …
- the i-th character can be decrypted independently of the others
  - parallelizable (unlike OFB)
  - random access
- the values to be XORed with the plaintext can be pre-computed

**Summary**

- **ECB**: used to encipher a single plaintext block (e.g., an AES key)
- **CBC**: repeated use of the block cipher to encrypt long messages
  - IV should be changed for every message
  - the integrity of the IV is important
  - needs padding of the plaintext message (padding oracle attacks)
  - some error propagation, self-synchronizing property
  - does not provide integrity protection (cut-and-paste attacks)
- **CFB, OFB, CTR**: essentially convert a block cipher into a stream cipher, although the CTR mode is often used to encrypt long messages (s = n)
  - OFB and CTR → synchronous stream ciphers
  - CFB → self-synchronizing stream-cipher
  - OFB has no error propagation
  - CTR has some advantages over OFB (parallelizable, random access)
  - in CFB, OFB, and CTR mode only the encryption algorithm is used, that is why some block ciphers (e.g., Rijndael) is optimized for encryption
  - do not provide integrity protection!