Outline

- five standardized modes (operation, properties)
  - Electronic Codebook (ECB) mode
  - Cipher Block Chaining (CBC) mode
  - Cipher Feedback (CFB) mode
  - Output Feedback (OFB) mode
  - Counter (CTR) mode

- attacks on CBC
  - simple attacks (content leak, cut and paste)
  - padding oracle attack by Vaudenay (2002)

- an attack on the CFB variant used in OpenPGP

- some exercises
**ECB mode**

- **Encrypt**
  
  \[
  \begin{align*}
  X_1 & \rightarrow E \rightarrow Y_1 \leftarrow D \rightarrow X_1 \\
  X_2 & \rightarrow E \rightarrow Y_2 \leftarrow D \rightarrow X_2 \\
  \vdots & \rightarrow E \rightarrow Y_N \leftarrow D \rightarrow X_N
  \end{align*}
  \]

- **Decrypt**
  
  \[
  \begin{align*}
  Y_1 & \rightarrow D \rightarrow X_1 \leftarrow E \rightarrow Y_1 \\
  Y_2 & \rightarrow D \rightarrow X_2 \leftarrow E \rightarrow Y_2 \\
  \vdots & \rightarrow D \rightarrow X_N \leftarrow E \rightarrow Y_N
  \end{align*}
  \]

**Properties of the 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.
  - Does not properly hide patterns in the plaintext.
- 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.
  - Additional integrity protection is essential.
- 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.
Illustration of ECB in action

CBC mode

- encrypt

- decrypt
Properties of the CBC mode

- Encrypting the same plaintext 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 $Y_{j-1}$.
  - Proper decryption of a correct ciphertext block needs a correct preceding ciphertext block only (see cut-and-paste attacks later in this slide set).
- 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 (see the padding oracle attack later in this slide set).
- Self-synchronizing property:
  - Automatically recovers from loss of a ciphertext block.
- Parallel computation (only for decryption), random access, no pre-computation.

Requirements on the IV

- The IV need not be secret (although secret IVs have some advantages), but it should be **unpredictable** and **non-manipulable** by the attacker.
- The problem with predictable IVs (in the chosen plaintext attack model):
  - Let $Y_i = E_K(Y_{i-1} + X_i)$ for some $i$ (part of a CBC encrypted message), and let us assume that the attacker suspects that $X_i = X^*$; can he confirm this?
  - The attacker predicts the next IV, submits $X = IV + Y_{i-1} + X^*$ to the oracle, and receives $Y = E_K(IV + X) = E_K(Y_{i-1} + X^*)$; if $Y = Y_i$, then $X_i = X^*$ is confirmed.
- The problem with manipulable IVs:
  - If an attacker can directly manipulate the IV (e.g., flip a selected bit of it), then he can make specific changes to the first plaintext block recovered (e.g., flip a selected bit of it).
Generating unpredictable IVs

- IV = \( E_K(N) \)
  - where \( N \) is a nonce (non-repeating value)
  - \( N \) may be a counter or a message ID (unique to the message)
  - to ensure non-manipulability, the sender should send \( N \) to the receiver (perhaps at the beginning of the CBC encrypted message), who should then compute the IV locally
    - \( N \) may be changed by an attacker, but he cannot control the effects made on the value of the IV

- IV = output of a **cryptographic** random number generator
  - random number generators available in standard programming libraries (e.g., rnd, rand, …) are not unpredictable, therefore they are not appropriate here!
  - to ensure non-manipulability the sender should send the IV in an encrypted form (e.g., \( E_K(IV) \)) to the receiver
    - \( E_K(IV) \) may be changed, but the attacker cannot control the effects made on the recovered IV

- both approaches also ensure the secrecy of the IV, which is advantageous

Padding

- the length of the message may not be a multiple of the cipher’s block size
- we must add some extra bytes to the short end block such that it reaches the correct size – this is called **padding**
- the receiver must be able to unambiguously recognize and remove the padding
- common examples for padding schemes:
  - append a x01 byte and then as many x00 bytes as needed (i.e., 1000…)
  - indicate the length of the padding in the last added byte

- note: padding is always used, even in the case when the length of the original message is a multiple of the block size: in this case, an entire extra block is added to the message
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 message tails: $x00$, $x01x01$, $x02x02x02$, ...
  - 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

---

CFB mode

- encrypt
  - initialized with IV
  - shift register $(n)$
  - $K \rightarrow E$
  - select $s$ MSB bits
  - $m_i \rightarrow c_i$

- decrypt
  - initialized with IV
  - shift register $(n)$
  - $K \rightarrow E$
  - select $s$ MSB bits
  - $c_i \rightarrow m_i$
Properties of the CFB mode

- encrypting the same plaintexts under the same key, but different IVs results in different ciphertexts
- ciphertext character $c_i$ depends on $m_i$ and all preceding plaintext characters
  - rearranging ciphertext characters affects decryption
  - proper decryption of a correct ciphertext character requires that the preceding n/s ciphertext characters are correct
- error propagation:
  - one bit error in a ciphertext character $c_i$ 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)
  - $m_i$ has bit errors where $C_i$ 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
- parallel computation (only for decryption), random access, no pre-computation

Another view on CFB

- if $s = n$, then...
  - encrypt
    
    \[
    \begin{array}{c}
    \text{IV} \\
    \hline
    K \rightarrow E \\
    x_1 \rightarrow y_1 \\
    \end{array}
    \begin{array}{c}
    K \rightarrow E \\
    x_2 \rightarrow y_2 \\
    \end{array}
    \begin{array}{c}
    K \rightarrow E \\
    x_3 \rightarrow y_3 \\
    \end{array}
    \ldots
    \begin{array}{c}
    K \rightarrow E \\
    x_n \rightarrow y_n \\
    \end{array}
    \]
  - decrypt
    
    \[
    \begin{array}{c}
    \text{IV} \\
    \hline
    K \rightarrow E \\
    y_1 \rightarrow x_1 \\
    \end{array}
    \begin{array}{c}
    K \rightarrow E \\
    y_2 \rightarrow x_2 \\
    \end{array}
    \begin{array}{c}
    K \rightarrow E \\
    y_3 \rightarrow x_3 \\
    \end{array}
    \ldots
    \begin{array}{c}
    K \rightarrow E \\
    y_n \rightarrow x_n \\
    \end{array}
    \]
**OFB mode**

- **encrypt**
  - initialized with IV
  - input register \((n)\)
  - select \(s\) MSB bits
  - \(E\)
  - \(m_i\) \((n)\)

- **decrypt**
  - initialized with IV
  - input register \((n)\)
  - select \(s\) MSB bits
  - \(E\)
  - \(c_i\) \((n)\)

**Properties of the 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 \(c_i\) depends on \(m_i\) only (does not depend on the preceding plaintext characters):
  - However, rearranging ciphertext characters affects decryption.
  - Statistical properties of the plaintext is hidden due to the random output of the block cipher.
- Error propagation:
  - One bit error in a ciphertext character \(c_i\) has an effect on the decryption of only that ciphertext character.
    - \(m_i\) has bit errors where \(c_i\) 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.
- Sequential computation only, no random access, pre-computation is possible.
Another view on OFB

- if \( s = n \), then...
  - encrypt
    \[
    \begin{align*}
    K \implies E & \quad X_1 \implies Y_1 \\
    K \implies E & \quad X_2 \implies Y_2 \\
    K \implies E & \quad X_3 \implies Y_3 \\
    \cdots\end{align*}
    \]
  - decrypt
    \[
    \begin{align*}
    K \implies E & \quad Y_1 \implies X_1 \\
    K \implies E & \quad Y_2 \implies X_2 \\
    K \implies E & \quad Y_3 \implies X_3 \\
    \cdots\end{align*}
    \]

CTR mode

- encrypt
  \[
  \begin{align*}
  \text{ctr}_1 \implies E & \quad X_1 \implies Y_1 \\
  \text{ctr}_2 \implies E & \quad X_2 \implies Y_2 \\
  \text{ctr}_3 \implies E & \quad X_3 \implies Y_3 \\
  \cdots\end{align*}
  \]

- decrypt
  \[
  \begin{align*}
  \text{ctr}_1 \implies E & \quad Y_1 \implies X_1 \\
  \text{ctr}_2 \implies E & \quad Y_2 \implies X_2 \\
  \text{ctr}_3 \implies E & \quad Y_3 \implies X_3 \\
  \cdots\end{align*}
  \]
Properties of the CTR mode

- similar to OFB, but …
- parallel computation and random access (unlike OFB), and pre-computation is possible too

Generating counter blocks

- it is crucial that counter values do not repeat, otherwise…
  - given \( Y = E_K(\text{ctr}) + X \) and \( Y' = E_K(\text{ctr}) + X' \), the attacker can compute \( Y + Y' = X + X' \); if \( X \) (or part of it) is known then \( X' \) (or part of it) is disclosed to the attacker

- this requires:
  - incrementing function for generating the counter blocks from any initial counter block must ensure that counter blocks do not repeat within a given message
  - the initial counter blocks must be chosen to ensure that counters are unique across all messages that are encrypted under the given key

- a typical approach:
  - divide the counter block into two sub-blocks \( \text{ctr} = \text{ctr}'|\text{ctr}'' \), where \( \text{ctr}'' \) is \( b \) bits long and \( \text{ctr}' \) is \( n-b \) bits long (\( n \) is the block size of the cipher)
  - \( \text{ctr}' \) is a nonce (e.g., a unique message ID) or it is a counter incremented with each new message (\( \Rightarrow \) max number of messages is \( 2^{n-b} \))
  - \( \text{ctr}'' \) is a counter incremented with every block within the message (\( \Rightarrow \) max message length is \( 2^b \) blocks)
Summary of properties

- **ECB:** used to encipher a single plaintext block (e.g., an AES key or an IV)
  - CBC: repeated use of the block cipher to encrypt long messages
    - IV should be changed for every message
    - the unpredictability and the non-manipulability of the IV is important
    - only the decryption can be parallelized, random access, no pre-computation
    - limited error propagation, self-synchronizing property

- **CFB, OFB, CTR:**
  - can be used to convert a block cipher into a stream cipher (s < n)
    - OFB and CTR → synchronous stream ciphers
    - CFB → self-synchronizing stream-cipher
    - only the encryption algorithm is used, that is why some block ciphers (e.g., Rijndael) are optimized for encryption

- **CFB:**
  - IV should be changed for every message
  - only the decryption can be parallelized, random access, no pre-computation
  - extended error propagation, self-synchronizing property

- **OFB:**
  - changing the IV for every message is very important
  - cannot be parallelized, no random access, pre-computation is possible
  - no error propagation, needs synchronization

- **CTR:**
  - non-repeating counters are very important
  - parallelizable, random access, pre-computation
  - no error propagation, needs synchronization

- **none of these modes provide integrity protection!**
  - encrypted message is longer than clear message due to padding (except if $s < n$ in CFB, OFB, and CTR modes)
Ciphertext stealing (CTS) in CBC

**encryption:**
- \( Y_i = E_K(X_i + Y_{i-1}) \) for \( i = 1..n-1 \)
- \( Y_n = E_K(X_n|0^* + Y_{n-1}) \)
- ciphertext: \( Y_1 | Y_2 | … | Y_{n-2} | Y_n | Y_{n-1}^{\text{trunc}(|X_n|)} \)

**decryption:**
- \( X_i = D_K(Y_i) + Y_{i-1} \) for \( i = 1..n-2 \)
- \( X_n = D_K(Y_n^{\text{trunc}(|X_n|)} + Y_{n-1}^{\text{trunc}(|X_n|)} \)
- \( Y_{n-1} = D_K(Y_n) + X_n|0^* \)
- \( X_{n-1} = D_K(Y_{n-1}) + Y_{n-2} \)

Some attacks on CBC

- content leak attack
- cut-and-paste attack
- padding oracle attack
Content leak attack on CBC

- let’s assume that we have two encrypted blocks:
  \[ Y_i = E_K(X_i + Y_{i-1}) \]
  \[ Y_j = E_K(X_j + Y_{j-1}) \]
- that happen to be equal:
  \[ Y_i = Y_j \]
- this means that
  \[ D_K(Y_i) = D_K(Y_j) \]
  \[ X_i + X_j = Y_{i-1} + Y_{j-1} \]

- the attacker knows the difference between \( X_i \) and \( X_j \)
- if \( X_i \) (or part of it) is known to the attacker, then \( X_j \) (or part of it) is also disclosed

Probability of a matching pair

\[ \Pr\{ Y_i = Y_j \} = ? \]

- assume that the block cipher works as a random function
- let \( P_k \) be the probability of having no matching pairs among \( k \) outputs (size of output space is \( N = 2^n \))
  \[ P_1 = 1 \]
  \[ P_2 = \frac{(N-1)}{N} \]
  \[ P_3 = \frac{(N-1)}{N} \frac{(N-2)}{N} \]
  \[ \ldots \]
  \[ P_k = \frac{(N-1)}{N} \frac{(N-2)}{N} \ldots \frac{(N-k+1)}{N} = \frac{1}{N^k} \frac{N!}{(N-k)!} \]

\[ \Pr\{ Y_i = Y_j \} = 1 - P_k \]

\[ k = \sqrt{N} = 2^{n/2} \]
Cut-and-paste attack on CBC

- given two encrypted messages $Y_1 Y_2 \ldots Y_p$ and $Y'_1 Y'_2 \ldots Y'_q$, we can construct $Y'_1 Y'_2 \ldots Y'_q Y_{i+1} \ldots Y_p$
- this will decrypt into $X_1 \ldots X_i \text{RX}'_{2} \ldots \text{RX}'_{q} R'X_{i+2} \ldots X_p$
- R and R* are garbage, but the receiver may actually expect random numbers at those positions of the message

$C \rightarrow S$: pass word:kis kacsa
$S \rightarrow C$: http://ww.crysy.s.hu/ind ex.html

The padding oracle attack on CBC

- 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 (chosen ciphertext attack model)
  - the server will drop the message with overwhelming probability
    - either the padding is incorrect (the server responds with a DECRIPTION_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, …
Recovering the last byte(s)

- assume we have an encrypted block $y_1 y_2 \ldots y_8 = E_K(x_1 x_2 \ldots x_8)$
- we want to compute $x_8$ (the last byte of $x$)
- idea:
  1. choose a random block $r_1 r_2 \ldots r_8$; let $i = 0$
  2. send $r_1 r_2 \ldots r_8 (r_8 \oplus i)y_1 y_2 \ldots y_8$ 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 1 or 2 or …
     - the most likely is that $(r_8 \oplus i) \oplus x_8 = 0$, and hence $x_8 = r_8 \oplus i$

```
  r_1 r_2 r_3 (r_8 \oplus i)
  Y_1 Y_2 \ldots Y_8
```

Recovering the last byte(s)

- assume we get that $x \oplus r$ has a correct padding, but we don’t know if it is 0 or 1 or 2 or …
- algorithm:
  1. let $j = 1$
  2. change $r_j$ and send $r_1 r_2 \ldots r_{j-1} r_j (r_j \oplus i) \oplus x_1 x_2 \ldots x_8$ 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; $x_j = r_j \oplus (j \oplus i)$ and hence $x_j x_{j+1} \ldots x_8 = r_j \oplus (j \oplus i) \oplus x_{j+1} x_{j+2} \ldots x_8 = r_j \oplus (j \oplus i) \oplus r_{j+1} \oplus (j \oplus i) \oplus \ldots \oplus r_8 \oplus (j \oplus i)$

```
x = DE AD BE EF DE AD BE EF  
z = 01 23 45 67 DD AE BD EC  
```

<table>
<thead>
<tr>
<th>$i$</th>
<th>$r$</th>
<th>$r \oplus x$</th>
<th>padding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00 23 45 67 DD AE BD EC</td>
<td>DE BE FB 88 03 03 03 03</td>
<td>OK</td>
</tr>
<tr>
<td>2</td>
<td>00 22 45 67 DD AE BD EC</td>
<td>DE RF FB 88 03 03 03 03</td>
<td>OK</td>
</tr>
<tr>
<td>3</td>
<td>00 22 44 67 DD AE BD EC</td>
<td>DE RF FA 89 03 03 03 03</td>
<td>OK</td>
</tr>
<tr>
<td>4</td>
<td>00 22 44 66 DD AE BD EC</td>
<td>DE RF FA 89 02 03 03 03</td>
<td>OK</td>
</tr>
<tr>
<td>5</td>
<td>00 22 44 66 DD AE BD EC</td>
<td>DE RF FA 89 02 03 03 03</td>
<td>ERROR</td>
</tr>
</tbody>
</table>

$x_j x_{j+1} x_8 = DD BE EF BE EF BE EF BE EF BE EF = DE AD BE EF$
Decrypted an entire block

- assume we have an encrypted block \( y_1, y_2, \ldots, y_8 = E_K(x_1, x_2, \ldots, x_8) \) and we know the value of \( x_{8,1}, \ldots, x_8 \) (using the method for recovering the last byte(s))
- we want to compute \( x_j \)
- algorithm:
  1. choose a random block \( r_1, r_2, \ldots, r_8 \) such that
     \[ r_j = x_j \oplus (9-j); r_{j+1} = x_{j+1} \oplus (9-j); \ldots; r_8 = x_8 \oplus (9-j) \]
  2. let \( i = 0 \)
  3. send \( r_1, r_2, \ldots, r_j, (r_j \oplus i)r_{j+1} \ldots r_8, y_1, y_2, \ldots, y_8 \) 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 \( x_j = r_j \oplus i \oplus (9-j) \)

\( \begin{array}{c|c|c|c|c|c|c}
\text{x} & \text{DE} & \text{AD} & \text{BE} & \text{DE} & \text{AD} & \text{BE} \\
\text{r} & \text{01} & 23 & 45 & 67 & \text{DA} & \text{A9} & \text{BA} \\
\text{r} \oplus \text{x} & \text{DF} & \text{8E} & \text{FB} & \text{89} & \text{04} & \text{04} & \text{04} \\
\text{i} & \text{0} & 01 & 23 & 45 & 67 & \text{DA} & \text{A9} & \text{BA} \\
\text{r} \oplus \text{i} & \text{DF} & \text{SE} & \text{FB} & \text{88} & \text{04} & \text{04} & \text{04} & \text{ERROR} \\
\text{r} \oplus \text{x} \oplus \text{i} & \text{DF} & \text{SE} & \text{FB} & \text{89} & \text{04} & \text{04} & \text{04} & \text{ERROR} \\
\text{x}_4 & \text{EB} \oplus \text{04} & \text{EF} \\
\end{array} \)

Decrypted an entire message

- assume we have a CBC encrypted message \( Y_1, Y_2, \ldots, Y_N \) where
  - \( Y_1 = E_K(X_1 \oplus IV) \)
  - \( Y_i = E_K(X_i \oplus Y_{i-1}) \) (for \( 1 < i < N \))
  - \( Y_N = E_K([X_N] \oplus \text{pad}[\text{plen}] \oplus Y_{N-1}) \)
- we want to compute \( X_1, X_2, \ldots, X_N \)
- algorithm:
  - decrypt \( Y_N \) using the block decryption method and XOR the result to \( Y_{N-1} \); you get \( X_N \)\|\text{pad}[\text{plen}] \)
  - decrypt \( Y_i \) using the block decryption method and XOR the result to \( Y_{i-1} \); you get \( X_i \)
  - decrypt \( Y_1 \) using the block decryption method and XOR the result to IV; you get \( X_1 \) (if the IV is secret you cannot get \( X_1 \))

complexity of the whole attack:
  on average we need only \( \frac{1}{2} \cdot 256 \cdot 8 \cdot N = 1024 \cdot N \) oracle calls!
Lessons learned

- content leak attack → use a sufficiently large block size (e.g., 128 bits)
- cut-and-paste attack → use some integrity protection mechanism (e.g., MAC or authenticated encryption (next lecture))
- padding oracle attack → pay attention on how the MAC function is used (e.g., apply it on the encrypted message)

CFB encryption in OpenPGP

- the receiver verifies if he uses the right key for decryption:
  \[ \{ E_K(0) + Y_1 \}_{b-1..b} = \{ E_K(0) \}_{b-1..b} + \{ Y_1 \}_{b-1..b} =? \{ E_K(Y_1) \}_{1..2} + Y_2^* \]
- if the above condition holds, then continue decryption, otherwise stop
A chosen ciphertext attack

\[ [E_K(0)]_{b-1:b} + [Y_1]_{b-1:b} =? [E_K(Y_1)]_{1..2} + Y_2^* \]

- we assume that the attacker knows
  - the ciphertext \( C_1 | C_2^* | C_3 | C_4 | \ldots \)
  - the first two bytes of the corresponding plaintext \( [M_1]_{1..2} \)
    - note that PGP compresses messages before encrypting them, and the compression method is encoded in the first two bytes of the compressed message
- computing \([E_K(0)]_{b-1:b}\):
  - send \( C_3 | D^* | C_3 | C_4 | \ldots \) to the oracle
  - the oracle verifies if
    \[ [E_K(0)]_{b-1:b} + C_2^* =? [E_K(C_3)]_{1..2} + D^* = [M_1]_{1..2} + [C_3]_{1..2} + D^* \]
    - if the oracle accepts the message, then the attacker knows that
      \[ [E_K(0)]_{b-1:b} = C_2^* + [M_1]_{1..2} + [C_3]_{1..2} + D^* \]
    - otherwise try another \( D^* \)
- computing \([M_2]_{1..2}\):
  - send \( C_3 | D^* | C_3 | C_4 | \ldots \) to the oracle
  - the oracle verifies if
    \[ [E_K(C_3)]_{1..2} = [E_K(0)]_{b-1:b} + [C_3]_{b-1:b} + D^* \]
    - if the oracle accepts the message, then the attacker knows that
      \[ [M_2]_{1..2} = [E_K(C_3)]_{1..2} + [C_4]_{1..2} \]
    - otherwise try another \( D^* \)
- computing \([M_3]_{1..2}\):
  - send \( C_4 | D^* | C_3 | C_4 | \ldots \) to the oracle
    - …