Security and Cryptography 2022

Mirosław Kutyłowski

XII. CRYPTO &COMMUNICATION SECURITY

part 1 - first lectures

many mistakes in practice:

- **risk of common (standard) groups**
- cryptanalysis: most efficient number field sieve (NFS):
  - complexity is subexponential (for $\mathbb{Z}_p$ it is $\exp(1.93 + o(1))(\log p^{1/3}(\log\log p)^{2/3})$
  - most time consuming: precomputation independent from the target number y ($\log(y)$ to be computed)
  - the time dependant computation on y can be optimized (still subexponential but lower)
  - 512-bit groups can be broken, MitM attack can be mounted

- standard safe primes – seem to be ok, but attacker can amortize the cost over many attacks
- TLS with DH: frequently **"export-grade" DH** with 512 bit primes, about 5% of servers support DHE_EXPORT, most servers (90% and more) use a few primes of a given length, after a precomputation breaking for a given prime: reported as 90 sec
- TLS: client wants DHE, server offers DHE_EXPORT, but one can manipulate the messages exchanged, so that the client treats the (p_{512}, g, g^b) as a response to DHE – it is not an implementation bug!
- sometimes non safe prime used (the number $\frac{p-1}{2}$ is composite), Pohling-Hellman method can be used
- DH-768 breakable on academic level, claims: DH-1024 breakable by state agencies in ...

recommendations:

- **avoid fixed prime groups**
- transition to **EC** (partially withdrawn due to required transition to post-quantum instead)
- deliberately **do not downgrade** security, even if seems to be ok
- follow the progress in computer algebra

Padding attack (Serge Vaudenay)

Attacked scenario:

- the plaintext should consist of some number of blocks of length $=b$
- padding is always applied (even if unnecessary)
- if i positions have to be padded: the padding consists of i bytes, each of them is i.
- So: removing padding is obvious
- encrypt the resulting padded plaintext $x_1, \ldots, x_N$ in the CBC mode with IV (fixed or random):

$$y_1 = \mathrm{Enc}(\mathrm{IV} \oplus a_1), \quad y_i = \mathrm{Enc}(y_{i-1} \oplus x_i)$$

- properties:
  - efficient
  - warning: do not repeat IV. (if IV fixed, then one can check that two plaintexts have the same prefix)

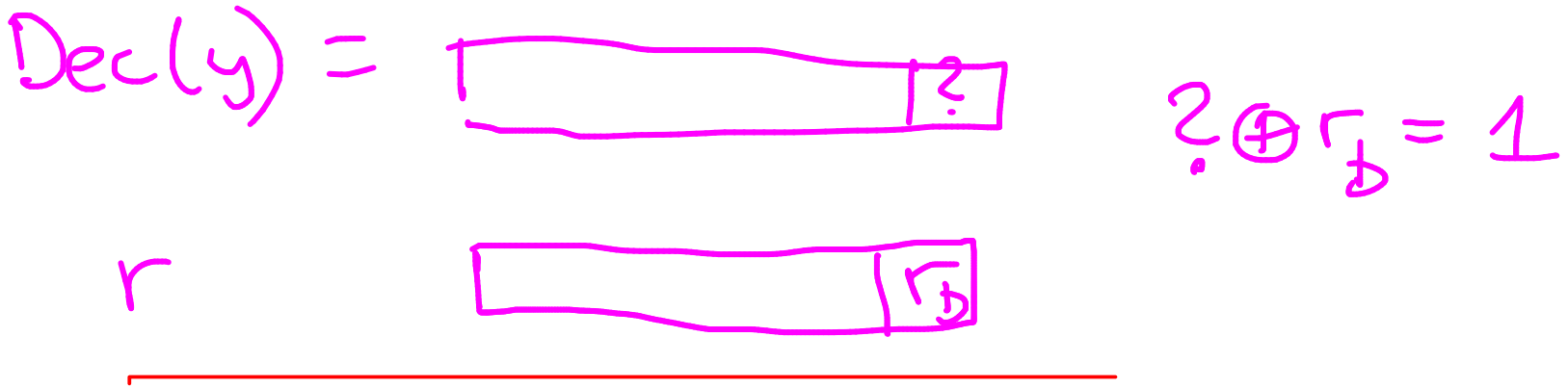
attack:

- manipulate cipertext
- destination node decrypts, padding might be incorrect
- **how to react to incorrect padding**? Each reaction will turn out to be wrong:
  - → reaction “reject”: creates padding oracle (attacker can see that some manipulations result in correct padding)
  - → reaction “proceed”: enables manipulation of the plaintext data

option "reject", last word oracle:

- goal: compute $a = \mathrm{Dec}(y)$ for a block y
- create an input for the padding oracle:
  - create a 2 block ciphertext: $r = r_1 \ldots r_b$ chosen at random, $c := r \,|\, y$
  - oracle call: if $\mathrm{Oracle}(c) = \mathrm{valid}$, then $\mathrm{Dec}(y) \oplus r$ should yield a correct padding.
  - whp this happens if $a_b = r_b \oplus 1$ (that is, if the padding consists of a single "1").
  - other options: suffix "22", "333", "4444",.... are less probable

decryption possibilities



if ok, then the results suffix is:

1
22
333
4444
55555

Probability:

p
p^2
p^3
p^4
p^5

Fishing out the cases of a longer suffix

- it may happen that the oracle says `valid` because of other correct padding.
- Solution (idea: change consequtive words in the padding until invalid):

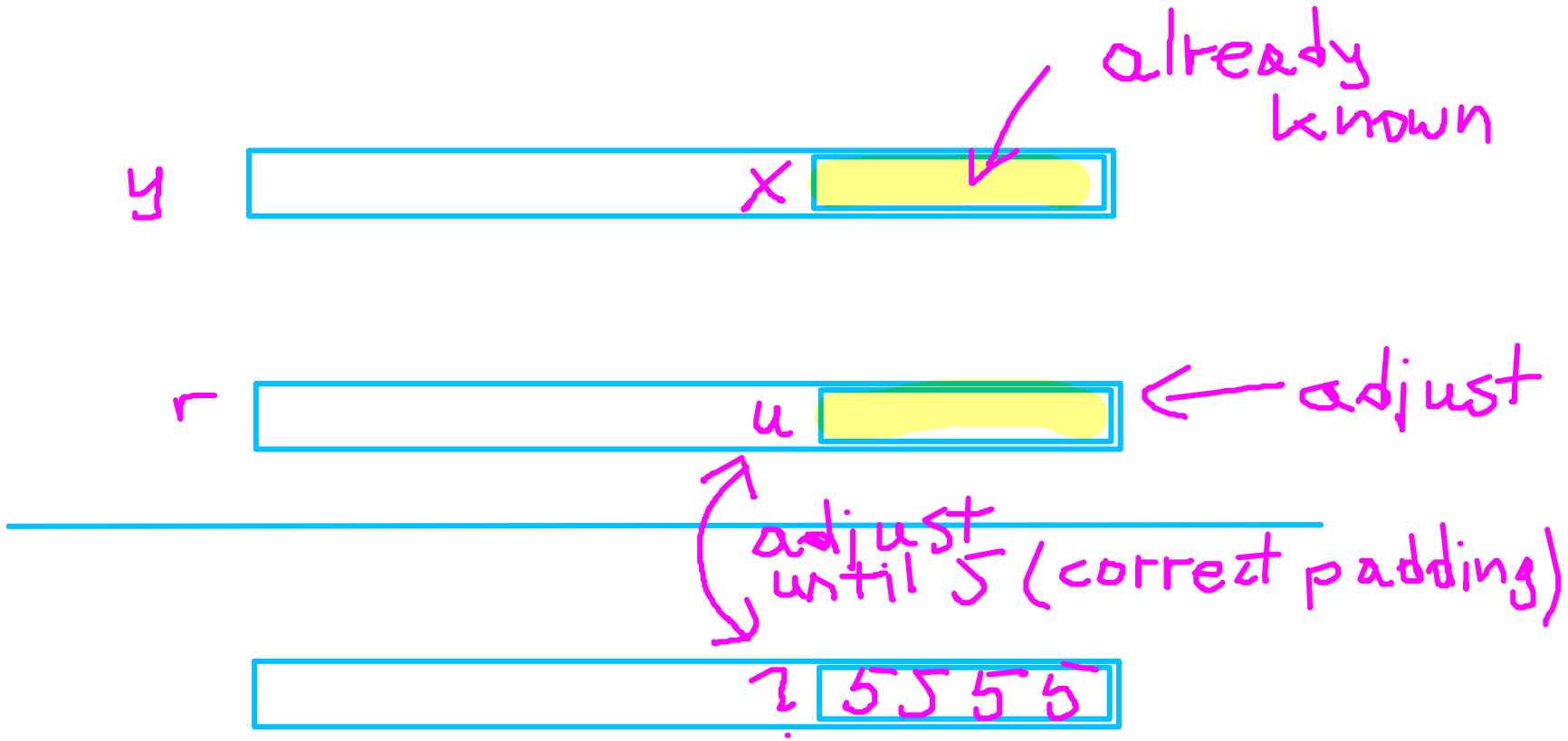
1. pick $r_1, r_2 \ldots, r_b$ at random, take $i = 0$
2. put $r = r_1 r_2 \ldots r_{b-1}(r_b \oplus i)$
3. run padding oracle on $r|y$, if the result `invalid` then increment i and goto (2)
4. $r_b := r_b \oplus i$ /* now we have a correct padding of an unknown length
5. for $j = b$ to 2:

   $r := r_1 \ldots r_{b-j}(r_{b-j+1} \oplus 1) r_{b-j} \ldots r_b$

   /* attempting to disturb padding, from left to right

   ask padding oracle for $r|y$, if `invalid` then output $(r_{b-j+1} \oplus j) \ldots (r_b \oplus j)$ and halt
6. output $r_b \oplus 1$ /* last choice, manipulating all positions except the rightmost has not created an error so the padding has length 1, so $y_b \oplus r_b = 1$ or $y_b = r_b \oplus 1$

changing r so that the result should have suffix 1, 22, 333, 4444, ...



once ok: $5 = x \text{ xor } u$

block decryption oracle

let $a_1 ... a_b$ be the plaintext of y

decryption:

- get a_b via the last word oracle
- proceed step by step learning a_{j-1} once $a_j, ..., a_b$ are already known

  1. set $r_k := a_k \oplus (b - j + 2)$ for $k = j, ..., b$ /* preparing the values so that the padding values $(b - j + 2)$ appear at the end)
  2. set $r_1, ..., r_{j-1}$ at random, $i := 0$ /* search for the value that makes a proper padding
  3. $r := r_1 ... r_{j-2}(r_{j-1} \oplus i) r_j ... r_b$
  4. if output on $r|y$ is `invalid`, then $i := i + 1$ and goto 3
  5. output $r_{j-1} \oplus i \oplus (b - j + 2)$

decryption oracle

- block by block, (after decryption we have to XOR with the previous ciphertext block due to CBC construction)
- the only problem is the first block if IV is secret

bomb oracles:

- padding oracle in SSL/TLS breaks the connection if a padding error occurs , so it can be used only once
- bomb oracle: try a longer part at once, execute many trials

other paddings:

easy to adjust the attack in the following cases (the reason is that we KNOW what to expect on a given position of the padding) :

- $00....0n$ instead of $nn....n$
- $12....n$ instead of $nn....n$

the padding where it would not work is a padding with random data on the added positions

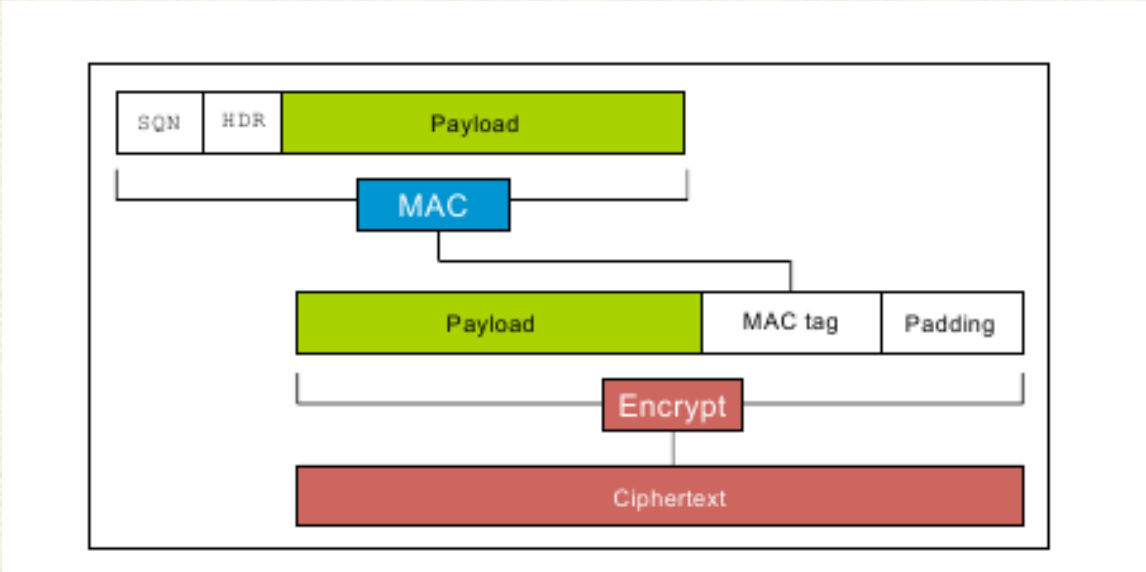
Applications for (old) versions of SSL/TLS, ...

- if MAC applied before padding, then padding oracle techniques can be applied
- wrong MAC and wrong padding create the same error message - from SSL v3.0, debatable whether it is impossible to recognize the situation via side channel (response time)
- TLS attempts to hide the plaintext length by variable padding

  IPSEC: discards message with a wrong padding, no error message, but there might be other activities to process errors (and they may leak information)
- WTLS: *decryption-failed* message in clear (!) session not interrupted
- SSH: MAC after padding (+)

Lucky Thirteen

- concerns DTLS (similar to TLS for UDP connections)
- MAC-Encode-Encrypt paradigm (MEE), MAC is HMAC based



8+5 = lucky 13

- 8-byte SQN, 5-byte HDR (2 byte version field, 1 byte type field, 2 byte length field)
- size of the MAC: 16 bytes (HMAC-MD5), 20 bytes (HMAC-SHA1), 32 bytes (HMAC-SHA-256)
- padding: $p+1$ copies of p, at least one byte must be added
- after receiving: checking the details: padding, MAC, (underflow possible if padding manipulated and padding removed blindly)
- HMAC of M:

  $T := H((K_a \oplus \text{opad}) || H((K_a \oplus \text{ipad}) || M))$ for constants opad and ipad and key K_a

.

– **Distinguishing attack:**

→ M_0 : 32 arbitrary bytes followed by 256 copies of 0xFF (11111111 in binary)

→ M_1: 287 bytes followed by $0x00$

→ both M_0 and M_1 consist of 288 bytes, plaintext consists of 18 16-byte (128-bit) blocks

→ encoded as $M_d||T||\text{pad}$, we aim to guess d

→ $C =$ the resulting ciphertext

M_0 : 32 arbitrary bytes followed by 256 copies of 0xFF

M_1: 287 bytes followed by $0x00$

→ create a ciphertext C' by truncating all parts of C corresponding to $T||\text{pad}$

→ give $\text{HDR}||C'$ for decryption

→ for M_0:

- 256 copies of 0xFF interpreted as padding and removed,
- remaining 32 bytes treated as a short message and MAC,
- calculating MAC: 4 hash block operations, then typically error returned to the attacker

→ if M_1: 8 hash evaluations as HMAC computed over a long message (then typically an error)

Plaintext recovery attacks for CBC encrypted transmission

- C^* – the block of ciphertext to be broken, C' – the ciphertext block preceding it
- we look for P^*, where $P^* = \mathrm{Dec}(C^*) \oplus C'$
- assume CBC with known IV, $b = 16$ (as for AES). $t = 20$ (as for HMAC-SHA-1)
- let Δ be a block of 16 bytes, consider

$$C^{\mathrm{att}}(\Delta) = \mathrm{HDR}||C_0||C_1||C_2||C' \oplus \Delta||C^*$$

it represents 4 non-IV blocks in the plaintext, the last block is:

$$P_4 = \mathrm{Dec}(C^*) \oplus (C' \oplus \Delta) = P^* \oplus \Delta$$

- case 1: P_4 ends with 0x00 byte:
  - 1 byte of padding is removed, the next 20 bytes interpreted as MAC, 43 bytes left - say R. MAC computed on SQN|HDR|R of 43+13=56 bytes
- case 2: P_4 ends with padding pattern of ≥ 2 bytes:
  - at least 2 bytes of padding removed, 20 bytes interpreted as MAC, at most 42 bytes left, MAC over at most 42+13=55 bytes
- case 3: P_4 ends with no valid padding:
  - according to RFC of TLS 1.1, 1.2 treated as with no padding , 20 bytes treated as MAC, verification of MAC over 44+13=57 bytes
    - MAC is computed to avoid other timing attacks!

- **time: case 1 and 3:** 5 evaluations of SHA-1, case 2: 4 evaluations of SHA-1, detection of case 2 possible in LAN

- **in case 2:** most probable is the padding 0x01 0x01, all other paddings have probability about $\approx\frac{1}{256}$ of probability of 0x01 0x01, so we may assume that $P_4 = P^* \oplus \Delta$ ends with 0x01 0x01. Then we derive the last two bytes of P^*.

-

repeat the attack with Δ' that has the same last two bytes as Δ to check if the padding has the length bigger than 2 (we are changing the byte 3 and observe whether the case 2 occurs, if it is so, then padding has length 2).

- after recovery of the last two bytes the rest recovered byte by byte from right to left:
  - the original padding attack
  - e.g. to find 3rd rightmost byte set the last two bytes Δ so that P_4 ends with 0x02 0x02, then try different values for the Δ so that Case 2 occurs (meaning that P_4 ends with 3 bytes 0x02
  - average time: $14 \cdot 2^7$ trials

practical issues:

→ for TLS after each trial connection broken, so multi-session scenario

→ timing difference small, so necessary to gather statistical data

→ complexity in fact lower, since the plaintexts not from full domain: e.g. http username and password are encoded Base64

→ partial knowledge may speed up the recovery of the last 2 bytes

→ less efficient configuration of the lengths for HMAC-MD5 and HMAC-SHA-256

BEAST

attack, phase 0:

1. P to be recovered (e.g. a password, cookie, etc), requires ability to force Alice to put secret bits on certain positions
2. force Alice to send a ciphertext of $0...0P_0$ (requires malware on her computer), where P_0 the last byte of P
3. eavesdrop and get $C_p = \mathrm{Enc}(C_{p-1} \oplus 0...0P_0)$
4. guess a byte g
5. force Alice to send encrypted plaintext $C_{i-1} \oplus C_{p-1} \oplus 0...0g$:

   then Alice sends $C_i = \mathrm{Enc}(C_{i-1} \oplus C_{i-1} \oplus C_{p-1} \oplus 0...0g) = \mathrm{Enc}(C_{p-1} \oplus 0...0g)$
6. if $C_i = C_p$ then $P_0 = g$

attack phase 1:

1. P_0 already known
2. force Alice to send $0 \ldots 0 P_0 P_1$ and proceed as in phase 0

phases 2-15 until the whole $P_0 \ldots P_{15}$ learned

protection: browser must be carefully designed, **injecting plaintexts must be prevented** (SOP- Same Origin Protection).

CRIME (2012)

- based on compression algorithm used by some versions of TLS
- compression: LZ77 and then Huffman encoding, LZ77- sliding window approach: instead of a string put a reference to a previous occurence of the same substring
- idea of recovering cookie:

```
POST / HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0) Gecko/20100101 Firefox/14.0.1
Cookie: secretcookie=7xc89f94wa96fd7cb4cb0031ba249ca2
Accept-Language: en-US,en;q=0.8

(... body of the request ...)
```

Listing 1: *HTTP request of the client*

modified POST:

```
POST /secretcookie=0 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0) Gecko/20100101 Firefox/14.0.1
Cookie: secretcookie=7xc89f94wa96fd7cb4cb0031ba249ca2
Accept-Language: en-US,en;q=0.8

( ... body of the request ...)
```

Listing 2: *HTTP request modified by the attacker*

- LZ77 compresses the 2nd occurence of secretcookie= or secretcookie=0.
- We try all options secretcookie=i to find out the case when compression is more effective (secretcookie=7)
- once the 1st character of cookie is recovered, repeat the attack for the 2nd character (trying all "secretcookie=7i" in the preamble)

TIME

- again based on compression but now on the server's side (from the client to the server compression might be disabled and CRIME fails)
- works if the server includes the client's request in the response (most do!)
- works even if SOP is enabled. SOP does not control data with the tag `img`, so the attacker can manipulate the length and therefore influence the number of blocks for block encryption
- the attacker requires malicious Javascript on the client's browser
- the attacker tries to get the secret value sent from the server to the client

- mechanism:
  - → as in CRIME, the request sends “secretvalue=x” where x varies
  - → the response is compressed, so it takes either “secretvalue=” or “secretvalue=x”
  - → the length manipulated so that either one or two packets are sent – connection specific data must be used: Maximum Transmission Unit
  - → RTT (round trip time) measured

independent of the browser, it is not an implementation attack!

countermeasure: restrict displaying images

BREACH

Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext

- attack against HTTP compression and not TLS compression as in case of CRIME
- a victim visits attacker-controlled website (phishing etc).
- force victim's computer to send multiple requests to the target website.
- check sizes of responses

```
GET /product/?id=12345&user=CSRFtoken=<guess> HTTP/1.1
Host: example.com
```

Listing 4: *Compromised HTTP request*

```
<form target="https://example.com:443/products/catalogue.aspx?id=12345&user=CSRFtoken=<guess>
...
<td nowrap id="tdErrLgf">
<a href="logoff.aspx?CSRFtoken=4bd634cda846fd7cb4cb0031ba249ca2">Log Off</a></td>
```

Listing 5: *HTTP response*

- requirements: application supports http compression, user's input in the response, sensitive data in the response
- countermeasures:
  - → disabling compression
  - → hiding length (randomizing the length of the output – it makes the attacks only harder if the attack can be repeated many times)
  - → no secrets in the same response as the user's data
  - → masking secret: instead of S send $R||S \oplus R$ for random R (fresh in each response)
  - → trace behaviour of requests and warn the user

POODLE (2014)

in SSL v.3.0 using technique from BEAST:

- padding is not covered by MAC so the attacker can manipulate it
- padding non-deterministic: padding 1 to L bytes (L= block length, say 16), the last byte denotes the number of preceding padding random bytes
- encrypted POST request:

  POST /path Cookie: name=value... $\langle$r\n\r\n$\rangle$ body ||20-byte MAC||padding

- manipulations such that:
  - the padding fills the entire block (encrypted to C_n)
  - the last unknown byte of the cookie appears as the last byte in an earlier block encrypted into C_i
- attack: replace C_n by C_i and forward to the server

  usually reject

  accept if $\mathrm{Dec}_K(C_i)[15] \oplus C_{n-1}[15] = 15$, thereby $P_i[15] = 15 \oplus C_{n-1}[15] \oplus C_{i-1}[15]$

  proceed in this way byte by byte
- downgrade dance: provoke lower level of protection by creating errors say in TLS 1.0, and create connection with SSL v3.0
- the attack does not work with weak (!) RC4 because of no padding

Weaknesses of RC4

- known weaknesses:

  $\rightarrow$ the first 257 bytes of encryption strongly biased, ≈ 200 bytes can be recovered if $\approx$232 encryptions of the same plaintext available

  (likely in case of broadcating the same text to many recipients)

  simply gather statistics as in case of Ceasar cipher

  $\rightarrow$ at some positions (multiplies of 256) if a zero occurs, then the next position more likely to contain a zero

- broadcast attack: force the user to encrypt the same secret repeatedly and close to the beginning

- countermeasure: no secrets in the initial part!

CCM encryption mode (Counter with CBC-MAC)

just to avoid patent threats (triggered by request to patent OCB mode - patented in USA, exempt for general public license for non-commertial use)

Prerequisites: block cipher algorithm; key K; counter generation function; formatting function; MAC length $Tlen$

Input: nonce N; payload P of $Plen$ bits; valid associated data A

Computation: Steps:

1. formatting applied to (N, A, P), result: blocks $B_0, ..., B_r$
2. $Y_0 := \mathrm{Enc}_K(B_0)$
3. for $i = 1$ to r: $Y_i := \mathrm{Enc}_K(B_i \oplus Y_{i-1})$
4. $T := \mathrm{MSB}_{Tlen}(Y_r)$
5. generate the counter blocks $\mathrm{Ctr}_0, \mathrm{Ctr}_1, ..., \mathrm{Ctr}_m$ for $m = Plen/128$
6. for $j = 0$ to m: $S_j := \mathrm{Enc}_K(\mathrm{Ctr}_j)$
7. $S := S_1 || ... || S_m$
8. $C := (P \oplus \mathrm{MSB}_{Plen}(S)) || (T \oplus S_0)$