CSE 130 Homework 2 Solutions (Spring 2025)

Jeffrey Peng

Private Key Encryption:

1) We need to define $G(s) \stackrel{\text{def}}{=} s||s$ and show an attack that would prove G is not a pseudorandom generator.
   - First, $G(s) \stackrel{\text{def}}{=} s||s$, the output of the equation means that the first half of the output is identical to the second half.
   - The function $G(s) = s||s$ is not a pseudorandom generator because its output has a clear and detectable pattern: the first half is always identical to the second half. This structure is highly unlikely in a truly random 2n-bit string, where the probability that both halves are equal is only $1/2^n$, which is negligible. In contrast,G(s) always produces outputs where the two halves are equal, meaning this pattern occurs with probability 1. Since a pseudorandom generator must produce outputs that are computationally indistinguishable from uniform random strings, and G(s) clearly does not, it fails to satisfy the definition and is therefore not a valid pseudorandom generator.
2) We need to define $F_k(x) \stackrel{\text{def}}{=} k\&x$ and show that this is not a pseudorandom function
   - The first issue with this function is that all bits in *k* that are 0 will always output 0 no matter what x is.
   - This means that the key *k*, can be found as if we query something that's all 1s ($F_k(1111)$, we'll get 1&*k* which output is the key *k*.
   - This **fails as a pseudorandom function** as the key is leaked if you query all 1s.
   - Example:
     - Key *k* = 1000
     - Query x = 1111
       (1) $F_k(1111) = k\&1111$
       (2)

k	x	output
1	1	1
0	1	0
0	1	0
0	1	0

       (3) $F_k(1111) = 1000$
     - We can see that the output is immediately just the key which by definition fails as a pseudorandom function
3) Show construction 3.30 and Example 3.26 not being secure.
   - Example 3.26 defines a function, F(k,x)=$k \oplus x$ (or $F_k(x) = k \oplus x$)
     - This function is/looks random for one single input but fails for multiple inputs

- If we query two different inputs x_1 and x_2, their outputs $y_1 = F_k(x_1)$ and $y_2 = F_k(x_2)$ have a predictable relationship
  - ➢ $y_1 \oplus y_2 = (k \oplus x_1) \oplus (k \oplus x_2) = (k \oplus k) \oplus (x_1 \oplus x_2) = 0 \oplus (x_1 \oplus x_2)$ = $x_1 \oplus x_2$
- If this was a truly random function, $y_1 \oplus y_2$ would be random and should have no relationship to $x_1 \oplus x_2$
  (1) A ciphertext XOR ciphertext = plaintext XOR plaintext

- Example:
- Construction 3.30 uses a pseudorandom function to encrypt messages.
  - Breaking it down:
    (1) It picks a random key *k*
    (2) It encrypts by choosing a random *r* and then computes ciphertext via $c =< r, F_k(r) \oplus m >$
    (3) It decrypts by using given $c =< r, s >$, recover $m = F_k(r) \oplus s$
    (4) In this instance, $F_k(r) \oplus m$ hides m perfectly as without *k*, an attacker can't find out *m*.
- Replacing $F_k(r)$ from Construction 3.30 with Example 3.26's $F_k(x) = k \oplus x$
  - We get $c =< r, (k \oplus r) \oplus m >$. We know from seeing Example 3.26 that the encryption isn't a pseudorandom function as it's not secure due to the output and the message having a relationship via XOR. This happens again when we try to encrypt with the combined function.
  - Let $c_1 =< r_1, (k \oplus r_1) \oplus m_1 >$
  - Let $c_2 =< r_2, (k \oplus r_2) \oplus m_2 >$
  - Let $s_1 = (k \oplus r_1) \oplus m_1$
  - Let $s_2 = (k \oplus r_2) \oplus m_2$
  - XOR-ing $s_1 and\ s_2$
    - $(k \oplus r_1 \oplus m_1) \oplus (k \oplus r_2 \oplus m_2) = (r_1 \oplus r_2) \oplus (m_1 \oplus m_2)$
    - $s_1 \oplus s_2 = (r_1 \oplus r_2) \oplus (m_1 \oplus m_2)$
  - This means that there's an relationship between the plaintexts ($m_1 and\ m_2$) and the ciphertext components ($r_1 r_2 s_1 s_2$)
  - The attacker will know $r_1 and\ r_2$ as they are clear in $c_1 and\ c_2$. If the attacker therefore chooses $m_1 and\ m_2$ in a chosen plaintext attack, they will get $m_1 \oplus m_2 = (s_1 \oplus s_2) \oplus (r_1 \oplus r_2)$ which would leak information about the plaintexts. **Therefore showing Example 3.26 cannot be used in Construction 3.30 as it is not CPA-secure.**
- **For Example:**

- Let the key *k* = 1100
- Let the attacker choose $m_1 = 0110$ and $m_2 = 1111$
  - For $m_1 = 0110$, let $r_1 = 0110$, $F_k(r_1) = k \oplus r_1 =$
  -

k	r_1	$k \oplus r_1$
1	1	0
1	0	1
0	1	1
0	0	0

  - Then, $(k \oplus r_1) \oplus m_1 =$
  -

$k \oplus r_1$	m_1	$(k \oplus r_1) \oplus m_1$
0	0	0
1	1	0
1	1	0
0	0	0

  - We therefore get $c_1 =< 1010, 0000 >$
  - For $m_2 = 1111$, let $r_2 = 0011$, $F_k(r_2) = k \oplus r_2 =$
  -

k	r_2	$k \oplus r_2$
1	0	1
1	0	1
0	1	1
0	1	1

  - Then, $(k \oplus r_2) \oplus m_2 =$
  -

$k \oplus r_2$	m_2	$(k \oplus r_1) \oplus m_2$
1	1	0
1	1	0

1	1	0
1	1	0

- We therefore get $c_2 =< 0011, 0000 >$
- We/the attacker can then $r_1 \oplus r_2 = 1010 \oplus 0011 = 1001$
- Then $s_1 \oplus s_2 = 0000 \oplus 0000 = 0000$
- Then $(r_1 \oplus r_2) \oplus (s_1 \oplus s_2) = 1001$
- If we check against $m_1 \oplus m_2$ we get $0110 \oplus 1111 = 1001$

- We can see here that the attacker can reveal information about the base messages $m_1 \, and \, m_2$. This is because the use of $F_k(x) = k \oplus x$ causes the key *k* to always cancel out.

4) Assuming the underlying stream is secure:

- From the Unsynchronized mode, we have $F_k(IV) \stackrel{\text{def}}{=} G_\infty(k, IV, 1^\ell)$, which is a pseudorandom function when IV is uniform
  - $G_\infty(k, IV, 1^\ell)$ behaves like a pseudorandom function if the stream cipher is secure (which we're assuming it is) and where *IV* is chosen at uniform random; where the same (*k*, IV) generates the same *K* but a different IV generates an output of *K* that looks totally random.
- Theorem 3.31 states "If *F* is a pseudorandom function, then Construction 3.30 is a CPA-secure private-key encryption scheme for messages of length n.
  - However in this case Construction 3.30 would be the unsynchronized stream-cipher mode of operation.
- Since we are assuming the underlying stream cipher is secure, Construction 3.30 defines encryption as $Enc(k, m) =< r, F_k(r) \oplus m >$, where *r* is chosen uniformly at random. When F is a secure pseudorandom function, Theorem 3.31 guarantees that this scheme is CPA-secure. However, in this question, the construction is instantiated with the function $F(k, x) = k \oplus x$ from Example 3.26, which is not pseudorandom. This function outputs $F_k(r) = k \oplus r$, which allows an attacker to exploit linearity. Suppose an attacker chooses two messages: $m_0 = 0^n$ and $m_1 = 1^n$, and receives the challenge ciphertext $c =< r, s >$ where $s = F_k(r) \oplus mb = (k \oplus r) \oplus (mb)$. The attacker can compute $s \oplus r = k \oplus mb$, and since they know both $m_0 \, and \, m_1$, they can test whether $s \oplus r$equals k or $k \oplus 1^n$. This allows them to determine b (the challenge bit) with 100% accuracy, breaking CPA security. Therefore, although Construction 3.30 is CPA-secure when used with a true pseudorandom function, it is **not CPA-secure** when instantiated with the insecure function from Example 3.26

5) Although *F* is assumed to be a secure pseudorandom function, the overall stream cipher construction fails to maintain security because of the way IVs are handled. Specifically, the IV is incremented deterministically and reused in a way that causes overlap between PRF inputs across keystream blocks. This makes the keystream predictable and distinguishable from random. In both constructions shown (from the initial question and Construction 3.29), an adversary could detect repeated values (e.g., identical halves in adjacent outputs) or infer relationships between them. This violates the requirement for a secure stream cipher, which demands that outputs be computationally indistinguishable from a truly random stream, even if the IV is known. Thus, despite the use of a secure PRF, the predictable and overlapping IV schedule renders the entire stream cipher insecure.

Message Authentication Codes and CCA-Secure Encryption:

6) This padded CBC-MAC is **Not Secure**.
    - The modified CBC-MAC contains:
        - Message length of at most $l * 2^n$ bits
            - Pad the message with 0 if it's the message < $l * 2^n$ bits
        - Then apply basic CBC-MAC.
    - Without needing to apply basic CBC-MAC, we can already see an issue with the instructions
        - Say we have padded messages $m = 101, and\ m' = 1010$
            - Where m=3 bits
        - Following the padding rule, the messages must all become the same length as the original, m, meaning all messages must become $1 * 2^3 = 8$.
        - Now since the messages are being padded with all 0s to length of 8, m becomes "10100000" and m' becomes "10100000", meaning, they have the same tag even before CBC-MAC. This is an issue because by having the same tag, different messages will collide to the same padded form which then results in identical tags for distinct messages.
7) Why do CBC, OFB, and CTR fail as CCA-Secure?
    - What violates CCA (Chosen Ciphertext Attack)-Secure?
        - A major thing that would make something not CCA-secure is if the mode of operation is malleable, therefore meaning that an attacker can easily adjust a ciphertext in a meaningful way that allows them to predict how the plaintext changes without knowing the key.
    - CBC (Cipher Block Chaining):
        - CBC encrypts plaintext blocks by XORing each block with the previous ciphertext block before applying the block cipher. Because each block relies on the previous block, CBC is malleable as an attacker can modify the ciphertext blocks to alter the decrypted plaintext in ways that are easy to predict. These predictions will also allow the attacker to manipulate other blocks potentially revealing information without needing to know the key. Specifically, flipping bits in C[i] will flip corresponding bits to P[i+1]

after decryption. Due to this, CBC is not CCA secure due to the potential information leak.
- Basically:
- C1 = Encrypted(P1 XOR IV)
- C2 = Encrypted(P2 XOR C1)... and so on.
- If a bit is flipped in C1, P1 won't be adjusted but that bit will also be flipped in C2
  - This allows the attacker to learn patterns in the encryption which is why CBC is malleable and therefore not CCA-secure.

- OFB (Output Feedback):
  - OFB turns a block cipher into a stream cipher by repeatedly encrypting an IV (initialization vector) to generate a keystream which is then XORed with the plaintext. Because the keystream is independent of the plaintext or ciphertext, an attacker can flip bits in the ciphertext to flip corresponding bits in the plaintext. This means that OFB is malleable as information can be leaked through these changes, making it not CCA secure.
    - OFB Basically:
    - S1 = Encrypted(IV)
    - S2 = Encrypted(S1)
      - C1 then = P1 XOR S1
      - C2 then = P2 XOR S2… etc
  - Because the Ciphertext is just the Plaintext XOR the stream, Flipping a bit *i* in the Ciphertext will flip the same bit *i* in the Plaintext, creating a pattern an attacker can learn which makes OFB malleable.
- CTR (Counter):
  - Similar to OFB, CTR generates a keystream by encrypting a counter (IV + 0, IV + 1…) and XORs the result with the plaintext. Just like OFB, CTR is malleable and an attacker can manipulate ciphertexts and observe the changes, meaning CTR is malleable.
    - IE: if an attacker flips a bit *i* in a ciphertext, the decryption will flip bit *i* as well. This would allow the attacker to observe and learn how these changes affect the output, which may reveal patterns in the encryption or directly show what the plaintext may be, making CTR malleable. Essentially:
      $Plaintext\ (P)\ =\ Ciphertext\ (C)\ \oplus\ Keystream$ where adjusting bit *i* in C would adjust bit *i* in P
  - Because CTR is malleable meaning that an attacker can modify the ciphertexts and influence the decrypted plaintext in predictable ways, and therefore meaning that the attacker can reveal information on the original text, CTR is not CCA secure.

8) For int i+1 (starting at second, We skip *i* as *i* is just IV/block 0) to i-1 (last block)
   - PreviousBlock = Previous Block [i-1]

- CurrentBlock = current block we're checking [i]
- PlaintextBlock = empty for later
- For int j (blocksize)-1 to 0
  - For int k (bytes from last to first)
    - Set padding_value for how many bytes we are trying to fake (padding_value = block_size - k)
    - Try all (256) bytes
      - TempBlock = PreviousBlock
      - Modifying TempBlock to try guessing if the byte you're on is correct
      - Fix bytes after the current one to match the padding pattern.
      - If oracle returns "valid padding"
        - We know this is the correct byte
        - Recover the actual plaintext byte
        - Save it to PlaintextBlock and output
        - Break and continue to next byte

9) Describe a padding-oracle attack on CTR-mode encryption, assuming PKCS #7 padding is used to pad messages to a multiple of the block length before encrypting.
   Breaking down this sentence:
   - Padding:
     - When bytes are added so that the end of the message to make its length a multiple of the block size.
   - Padding Oracle:
     - From Lecture Slides:
     - "Even if an error is not explicitly returned, an attacker might be able to detect differences in timing, behavior, etc. After decryption"
     - "If an attacker modifies (only) the *i*-th byte of IV, this cause a predictable change to (only) the *i*-th byte of the encoded data
   - PKCS #7 encoding/padding
     - Assume message is an integral # of bytes
     - Let L be the block length (in bytes) of the cipher
     - Let b>0 be # of bytes that need to be appended to the message to get length "a" multiple of "L"
       - 1 <= b <= L; note b!=0
     - Append b (encoded as a 1-byte string), b times
       - I.e., if 3 bytes of padding are needed, append 0x030303
   - PKCS #7 decryption
     - USE CBC-mode decryption to obtain encoded data
     - Say the final byte of encoded data has value b
       - If b=0 or b>L, return "error"
       - If final b bytes of encoded data are not all equal to b, return "error"
       - Otherwise Strip off final b bytes of the encoded data, and output what remains as the message

- CTR-mode
  - From #7 we know that CTR is malleable where by adjusting bit *i* in a ciphertext, bit *i* is also changed for the plaintext
- Now the question is a PKCS #7 Padding-Oracle on CTR-mode attack if PKCS #7 is used.
  - Assume ciphertext C is encrypted via CTR-mode with PKCS #7 Used.
    - After an initial decryption,
      - C converts back into a plaintext (containing the padding) due to CTR decryption.
      - Then the plaintext is checked with PKCS #7 padding and returns an error (via the oracle) if it's invalid.
    - Since CTR is malleable:
      - The attacker can modify any byte of the ciphertext which would lead to predictable changes in the decrypted plaintext byte (due to $Plaintext\ (P)\ =\ Ciphertext\ (C)\ \oplus\ Keystream$) The attacker would also know if the byte adjustment is proper as the attacker would also receive an error (padding-oracle) if the padding is invalid.
  - Although CTR-mode encryption does not require padding, when PKCS #7 padding is applied and validated after decryption, it introduces a vulnerability due to CTR's malleability. In CTR mode, each ciphertext byte is XORed with a keystream byte, so flipping a bit in the ciphertext leads to a predictable bit flip in the corresponding plaintext. If the system performs padding validation after decryption and reveals whether the padding is valid, an attacker can modify ciphertext bytes to influence the decrypted padding and observe whether it passes the check. This feedback effectively turns the system into a padding oracle. The attacker can then exploit this oracle to iteratively guess and recover the plaintext, one byte at a time. Thus, even though CTR mode itself is secure, the combination of CTR with PKCS #7 padding and error-leaking padding validation makes it vulnerable to a padding oracle attack.