CSE 130 Homework 3 Solutions (Spring 2025)

Jeffrey Peng

1. Let $h^s : \{0,1\}^{2n} -> \{0,1\}^n$ be a secure compression function keyed by random seed

   $s <- Gen(1^\lambda)$

   - The hash function H is defined by the transform in the problem:
   - Given input $x \in \{0,1\}^*$,
     - Append a 1 bit and enough 0s so total length is n mod n = 0
     - Break into $x_1, x_2, \ldots x_B \in \{0,1\}^n$
     - Set initial value $z_0 = IV \in \{0,1\}^n$
     - For i = 1 to B: compute $z_i = h^s(z_{i-1} || x_i)$
     - Output $H(x) = z_B$
   - The key difference in this function compared to the normal Merkle-Damgard function is that this version does not include the length of the original message in the padding.
   - This version is also insecure because if the compression function h^s is collision-resistant, the modified Merkle-Damgard construction will not be collision-resistant.

   Proof:

   - Assume we have a message M = $x_1 || x_2 || \ldots || x_B$, already block-aligned with $|x_i| = n$ and no padding needed (for simplicity)
   - Compute hash as:
     - $z_0 = IV,\ z_1 = h^s(z_0 || x_1),\ \ldots,\ z_b = h^s(z_{B-1} || x_B)$
   - Let $M' = M || x_{B+1} || \ldots || x_{B+k}$, where we append new blocks after M.

- Define M' to be such that: $z_B = z'_0$, $z'_{i+1} = h^s(z'_i || x_{B+i})$ thus $H(M') = z'_k$

- Suppose H(M) = H(M'). This is only true if the final state $z_B = z'_k$ but unless we append specific values, this won't hold.

Attack:

- However for the attack, we start with a fixed input M and find two different extensions $M_1 = M||x \; and \; M_2 = M||x'$ such that $h^s(z_B||x) = h^s(z_B||x') => H(M_1) = H(M_2)$ which is a collision on the compression function, and it extends into a collision onto the hash function. This is possible because the state after processing M is reused, and the input length is not encoded.

Result:

- This Merkle-Damgard construction is similar to the original only it's modified to not include the message length. The reason why it's insecure is because if an attacker takes one message and extends it two different ways, it's possible to get the same final hash which breaks collision-resistance rules.

2. We are now given another modified Merkle Damgard transform that:
   a. Pads the input x with a 1 followed by enough 0s to make its length a multiple of n.
   b. Splits the padded input into blocks $x_1 ..., x_B$, each of length n
   c. Uses a fixed IV $z_0 = 0^n$
   d. Computes the hash as
      i. $z_i = h_s(z_{i-1}||x_i) \; for \; i = 1,..., B$ where $h_s: \{0,1\}^{2n} -> \{0,1\}^n$ is a collision-resistant compression function keyed by a seed s <- $Gen(1^\lambda)$

e. Outputs $H(x) = z_B$

f. Similar to question 1, the main difference here is that there isn't any padding.

g. We want to show that there exists a collision-resistant h_s such that this modified transform falls to be collision-resistant

Proof of Insecurity:

- The standard MD includes the message length in the padding to prevent extension attacks. However, without the length, an attacker can:
  - Take a message M and compute its hash $H(M) = z_B$
  - Then append arbitrary blocks x_{B+1}...., x_{B+k} to M to form M', such that the final state z_{B+k} equals z_B, thus forcing $H(M) = H(M')$
- To construct the attack, we define a collision-resistant h_s that allows such an attacker
  - Let h_s be a compression function where:
    - For most inputs, h_s behaves as a secure collision-resistant function.
    - For a specific input $(0^n||x_1)$, we force: $h_s(0^n||x_1) = 0^n$ for some fixed x_1
  - Let $M = x_1$
    - $z_0 = 0^n$, $z_1 = h_s(0^n||x_1) = 0^n$ so $H(M) = z_1 = 0^n$
  - Let $M' = x_1||x_2$
    - $z_0 = 0^n$, $z_1 = h_s(0^n||x_1) = 0^n$, $z_2 = h_s(0^n||x_1) = 0^n$ so $H(M') = z_2 = 0^n$
  - Thus M != M', but H(M) = H(M') which is a collision

- For any message M ending with x_1, we can append x_1 again without changing the hash
  - If $H(M) = z_B$, then $H(M||x_1) = h_s(z_B||x_1)$.
  - If $z_B = 0^n$, then $H(M||x_1) = 0^n = H(M)$
  - Which proves we can create infinitely many collisions by appending x_1
- h_s remains collision-resistant because:
  - The bad behavior of h_s is only for the specific input $(0^n||x_1)$
  - For all other inputs, h_s is collision-resistant by construction
  - Finding collisions in h_s would require finding two distinct pairs (a||b) != (a'||b') such that $h_s(a||b) = h_s(a'||b')$ which is hard except for the fixed cases defined.
- Basically, there exists a collision-resistant compression function h_s such that the given modified Merkle-Damgard transform is not collision-resistant. Specifically, if h_s is defined to satisfy $h_s(0^n||x_1) = 0^n$ for some fixed x_1 then:
  - The message $M = x_1$ and $M' = x_1||x_1$ collide under H, which violates collision resistance proving the transform is insecure.

3. Total unique songs: n=3500

   Songs are selected uniformly at random with replacement

   We want the smallest number *k* such that the probability of at least one repetition in *k* plays is >= 50%.

   - The probability *P* that no repetition occurs in *k* plays is:
     - $P(no\ repeats) = n/n * n - 1/n * n - 2/n *... \frac{n-(k+1)}{n}$ or $e^{-\frac{k^2}{2n}}$

- We want the probability of at least one repeat and >= 0.5 so we have
  - $P(repeats) = 1 - e^{-\frac{k^2}{2n}} >= 0.5$
  - $e^{-\frac{k^2}{2n}} <= 0.5$
  - $-\frac{k^2}{2n} <= ln(0.5) =- ln(2)$
  - $\frac{k^2}{2n} >= ln(2)$
  - $k^2 >= 2n * ln(2) = k >= \sqrt{2n * ln(2)}$
- Inserting n = 3500
  - $k >= \sqrt{2(3500) * ln(2)} \sim = 69.6$
- Which means approximately **70** plays before there's a 50% chance that some song is played twice

4. Keyed Function: $F_k(x) \stackrel{\text{def}}{=} H(k||x)$
   - The keyed function, F_k, is a pseudorandom function if for all PPT distinguishers *D*, the following advantage is negligible
     - $|Pr[D^{F_k(*)}(1^n) = 1] - Pr[D^{f(*)}(1^n) = 1] <= negl(n)$
       - Where k is chosen uniformly at random
       - f is a truly random function
       - D can query its oracle adaptively
     - Alongside the random oracle H which is a publicly accessible function that returns uniformly random outputs for each unique input.
   - Looking at $F_k(x) \stackrel{\text{def}}{=} H(k||x)$,
     - The input to *H* is the concatenation of the key *k* and the input *x*,

- Since k is a secret and uniformly random, the input k||x is unique for each x
- The random oracle H ensures that H(k||x) is uniformly random for each unique k||x
- Now, if D queries F_k at points x_1,... x_2,... x_q, the responses will be $H(k||x_1),\ H(k||x_2)$,..., $H(k||x_q)$, which are indistinguishable from random values
- If D queries a truly random function f, the responses will also be uniformly random
- The only way D could distinguish F_k from f is if it queries two distinct x_i, x_j such that k||x_i = $k||x_j$ which is impossible unless $x_i = x_j$

- If there exists a distinguisher D that can distinguish F_k from f with non-negligible advantage. We can construct an adversary A that uses D to break the random oracle property
  - A simulates D's queries by forwarding k||x to H and returning H(k||x).
  - Since H is a random oracle, A provides a perfect simulation of F_k
  - If D can distinguish F_k from f, it implies D can detect non-randomness in H, which contradicts the random oracle assumption.
- Therefore, since H is a random oracle, $F_k(x) = H(k||x)$ is computationally indistinguishable from a truly random function $f(x)$
- And therefore, F_k is a pseudorandom function under the random oracle model.

5. a. We are given a degree-7 LFSR with feedback taps at $c_6, c_1,\ and\ c_0$. The initial state is $(s_6, s_5, s_4, s_3, s_2, s_1, s_0) = (0, 0, 0, 0, 0, 0, 1)$

- The output bit is s_0
- The feedback bit is computed as $s_6 \oplus s_1 \oplus s_0$
- The register shifts right, and the feedback is inserted at s_6
- 10 Bits Output
    - 1.
    - Initial state: 0,0,0,0,0,0,1
    - Output: 1
    - Feedback $0 \oplus 0 \oplus 1 = 1$
    - Next state: 1,0,0,0,0,0,0
    - 2.
    - State: 1,0,0,0,0,0,0
    - Output: 0
    - Feedback $1 \oplus 0 \oplus 0 = 1$
    - Next state: 1,1,0,0,0,0,0
    - 3.
    - State: 1,1,0,0,0,0,0
    - Output: 0
    - Feedback: $1 \oplus 0 \oplus 0 = 1$
    - Next state: 1,1,1,0,0,0,0
    - 4.
    - State: 1,1,1,0,0,0,0
    - Output: 0
    - Feedback: $1 \oplus 0 \oplus 0 = 1$

- Next state: 1,1,1,1,0,0,0
- 5.
- State: 1,1,1,1,0,0,0
- Output: 0
- Feedback: $1 \oplus 0 \oplus 0 = 1$
- Next state: 1,1,1,1,1,0,0
- 6.
- State: 1,1,1,1,1,0,0
- Output: 0
- Feedback: $1 \oplus 0 \oplus 0 = 1$
- Next state: 1,1,1,1,1,1,1,0
- 7.
- State: 1,1,1,1,1,1,1,0
- Output: 0
- Feedback: $1 \oplus 1 \oplus 0 = 0$
- Next state: 0,1,1,1,1,1,1,1
- 8.
- State: 0,1,1,1,1,1,1
- Output: 1
- Feedback: $0 \oplus 1 \oplus 1 = 0$
- Next state: 0,0,1,1,1,1,1
- 9.
- State: 0,0,1,1,1,1,1

- Output: 1
- Feedback $0 \oplus 1 \oplus 1 = 0$
- Next state: 0,0,0,1,1,1,1
- 10.
- State: 0,0,0,1,1,1,1
- Output: 1
- Feedback: $0 \oplus 1 \oplus 1 = 0$
- Next state: 0,0,0,0,1,1,1
- First 10 output bits: 1,0,0,0,0,0,0,1,1,1

b. A maximum-length LFSR of degree n cycles through all $2^n - 1$ nonzero states before repeating. To show that this LFSR is not maximum-length, we need to find a nonzero state that transitions to itself (a self-loop), which would cause a shorter cycle.

For example, we can check the state of (1,1,1,1,1,1,1).

- Output: 1
- Feedback: $1 \oplus 1 \oplus 1 = 1$
- Next state (1,1,1,1,1,1,1)

This is obviously a self-loop which means that the LFSR will repeat the state indefinitely, producing an infinite sequence of 1s. And since the maximum length LFSR must cycle through all $2^7 - 1 = 127$ nonzero states, the presence of this specific self-loop prove that the LFSR is not maximum length.

6. a. This stream cipher is insecure because:

- There's and gate bias which basically means:
  i. Output is 1 only if both LFSRs output 1.

ii. Output 0 otherwise

- but also because of correlation attacks which means the attacker can exploit the bias to separately attack A and B
  i. If the keystream has a 1, the attacker knows both LFSRs output 1 at that tick
  ii. If the keystream has a 0, at least one LFSR output 0, but this leaks less information
- and also, these LFSRs alone are insecure due to linearity, combining them with and does not sufficiently hide their structure. The keystream's linear complexity is at most $n_a + n_b$, making it vulnerable.

b. To recover the key efficiently, we exploit the fact that the AND operation outputs a 1 only when both LFSRs output 1. Thus, if we observe a 1 in the keystream, we know that both LFSRs were in a state that produced 1 at that time. This allows us to attack the two LFSRs independently by focusing only on the position where the keystream is 1

- Let the keystream be $z_1, z_2, \ldots z_m$
- We need sufficiently many 1s in the keystream to uniquely determine the initial state of A and B.
- Since $Pr[z_i = 1] = 1/4$, we need roughly m~=4 * max(n_a, n_b) bits to ensure enough 1s for solving
- Identifying where $z_i = 1$, Let $S = \{i | z_i = 1\}$. For these positions, both LFSRs will output 1
- To recover LFSR A:

- For each possible initial state of A and record its output sequence $a_1, a_2, \ldots, a_m$, check if $a_i = 1$ for all $i \in S$.
  - if Yes, this is a candidate for initial state for A.
- Since LFSRs are deterministic, the correct initial state will satisfy $a_i = 1$ at all $i \in S$
- This would also make the time complexity of recovering LFSR A, $O(2^{n_a})$

- To recover LFSR B:
  - We repeat the same process for LFSR B
    - Simulate LFSR B and check if its output $b_i = 1$ for all $i \in S$.
  - This also has a time complexity of $O(2^{n_b})$
- Combining them means that the initial states of A and B must jointly satisfy $z_i = a_i \hat{} b_i$ for all i.
- Since we enforced $a_i = 1 \; and \; b_i = 1$ at all $i \in S$, the remaining keystream bits will automatically be satisfied because at least one of $a_i \; or \; b_i$ must be 0 at those positions.
- Total time complexity: $O(2^{n_a} + 2^{n_b})$

7. We have a function $F_k(x) = P(const||k||x)$ which uses a public, invertible permutation P. Since P is invertible and its structure is known, an adversary can exploit this to distinguish F_k from a random function
   - Constructing the Distinguisher:
   - Query Phase:
     - The distinguisher D queries the oracle at two distinct inputs $x_1 \; and \; x_2$

- It will receive outputs $y_1 = F_k(x_1)\ and\ y_2 = F_k(x_2)$

- Inversion Phase
  - Computing $P^{-1}(y_1)\ and\ P^{-1}(y_2)$ for F_k gives $const||k||x_1$ and $const||k||x_2$
  - Then we check if the first

    $|const|\ +\ |k|\ bits\ of\ P^{-1}(y_1)\ and\ P^{-1}(y_2)\ are\ identical.$

    - If yes, the oracle is F_k because k is fixed
    - If no, the oracle is random because f has no such constraint.

- Decision:
  - Output 1 if the first |const| + |k| bits match
  - Output 0 otherwise

- Distinguishing Probability:
  - If the oracle is F_k:
    - The distinguisher always observes matching prefixes (due to fixed const||k)
    - Pr[D outputs 1]=1
  - If the oracle is random f:
    - The outputs $y_1\ and\ y_2$ are independent random strings
    - The probability that $P^{-1}(y_1)\ and\ P^{-1}(y_2)$ share the same prefix is $2^{-(|const|+|k|)}$
    - Therefore, $Pr[D\ outputs\ 1] = 2^{-(|const|+|k|)}$

- The distinguisher's advantage is therefore

  $|Pr[D^{F_k} = 1] - Pr[D^f = 1]| = 1 - 2^{-(|const|+|k|)}$ which is non-negligible.

8a. Let $Feistel_{\{f_1,f_2\}(L_0,R_0)}$ denote a 2-round Feistel network:

- $(L_1, R_1) = (R_0, L_0 \oplus f_1(R_0))$
- $(\hat{L}_2, \hat{R}_2) = (R_1, L_1 \oplus f_2(R_1)) = L_0 \oplus f_1(R_0),\ R_0 \oplus f_2(L_0 \oplus f_1(R_0)))$
- Let: $(L_2, R_2) = swap(Feistel_{f_1,f_2}(L_0, R_0)) = (R_2, L_2)$
- Claim: $(L_0, R_0) = swap(Feistel_{f_2,f_1}(\hat{L}_2, \hat{R}_2))$
- Then, we can compute the Feistel network $Feistel_{\{f_2,f_1\}}$ on $(\hat{L}_2, \hat{R}_2) = (R_2, L_2)$
- Applying the Swap: $swap(Feistel_{f_2,f_1}(\hat{L}_2, \hat{R}_2)) = (R'_2, L'_2)$
- $(L_0, R_0) = (\hat{R}_2 \oplus f_1(\hat{L}_2 \oplus f_2(\hat{R}_2)), \hat{L}_2 \oplus f_2(\hat{R}_2))$
  - Using: $\hat{L}_2 = R_2 = R_0 \oplus f_2(L_0 \oplus f_1(R_0)),\ \hat{R}_2 = L_2 = L_0 \oplus f_1(R_0)$
- Finally: $R_0 = \hat{L}_2 \oplus f_2(\hat{R}_2),\ L_0 = \hat{R}_2 \oplus f_1(\hat{L}_2 \oplus f_2(\hat{R}_2))$

8b.

- Given: $(\hat{L}_{16}, \hat{R}_{16}) = swap(Feistel_{f_{15},f_{16}}(\ldots Feistel_{f_1,f_2}(L_0, R_0)\ldots))$
- Given Claim: $(L_0, R_0) = swap(Feistel_{f_2,f_1}(\ldots Feistel_{f_{16},f_{15}}(L_{16}, R_{16})\ldots))$
- The book explains that a Feistel network can be inverted by reversing the order of the round functions and applying a swap operation to both the input and output.
- Therefore, $(\hat{L}_{16}, \hat{R}_{16}) = swap(Feistel_{f_1\ldots f_{16}}(L_0, R_0))$
- $Feistel_{f_{16}\ldots f_1}(\hat{L}_{16}, \hat{R}_{16}) = swap(L_0, R_0) => (R_0, L_0)$
- Finally: $(L_0, R_0) = swap(Feistel_{f_{16},\ldots f_1}(\hat{L}_{16}, \hat{R}_{16}))$

This completes the inversion by reversing the round functions and applying a swap to both the input and output, we exactly undo the original Feistel computation.