Problem Set 4

Alex Miller

October 28, 2021

Abstract

Collaborators: Elizabeth Coble, Lucy Li

1

Assume that each Attack has access to a (perfect) hash table H of size 2^n. Assume all the indices of H are initialized to $\perp$. Assume that for all block ciphers, $F_1, F_2, F_3, andF_4$, we have t examples of (m, c) plaintext-ciphertext example pairs. Assume t is sufficiently large to mitigate the probabilities of false positive keys.

1.1 $F_1 : \{0,1\}^{3n} \times \{0,1\}^l \to \{0,1\}^l$

$F_1(k_1||k_2||k_3, m) = E(k_3, E(k_2, E(k_1, m)))$

$F_1^{-1}(k_1||k_2||k_3, c) = E^{-1}(k_1, E^{-1}(k_2, E^{-1}(k_3, c)))$

Algorithm 1: *Attack*$((m_1, c_1), (m_2, c_2), ..., (m_t, c_t))$

for $k \in \{0,1\}^n$ **do**
    $x \leftarrow E(k, m_1)||E(k, m_2)||...||E(k, m_t)$
    $H[x] \leftarrow k$
end
for $k_2||k_3 \in \{0,1\}^{2n}$ **do**
    $x \leftarrow= E^{-1}(k_2, E^{-1}(k_3, c_1))||E^{-1}(k_2, E^{-1}(k_3, c_2))||...||E^{-1}(k_2, E^{-1}(k_3, c_t))$
    if $H[x] \neq \perp$ **then**
        $k_1 \leftarrow H[x]$
        return $k_1||k_2||k_3$
    end
end

This attack works because, for a given (m, c) plaintext-ciphertext pair, the following should hold for the correct values of k_1, k_2, and k_3:

$E(k_1, m) = E^{-1}(k_2, E^{-1}(k_3, c))$

This attack utilizes 2^n memory and executes in $2^{2n} + 2^n \approx 2^{2n}$ time

1.2 $F_2 : \{0,1\}^{n+l} \times \{0,1\}^l \to \{0,1\}^l$

$F_2(k_1||a_1, m) = E(k_1, m \oplus a_1)$

$F_2^{-1}(k_1||a_1, c) = E^{-1}(k_1, c) \oplus a_1$

Algorithm 2: *Attack*$((m_1, c_1), (m_2, c_2), ..., (m_t, c_t))$

```
for k ∈ {0,1}^n do
    x ← E^{-1}(k, c_1)||E^{-1}(k, c_2)||...||E^{-1}(k, c_t)
    y ← m_1||m_2||...||m_t
    z_1||z_2||...||z_t ← x ⊕ y
    if ∀i, j ∈ [t], i ≠ j, z_i == z_j then
        return k||z_1
    end
end
```

This attack works because, for a given (m, c) plaintext-ciphertext pair, the following should hold for the correct values of k_1 and a_1:

$m \oplus a_1 = E^{-1}(k_1, c)$, therefore $E^{-1}(k_1, c) \oplus m = a_1$. Therefore if we find a k such that we can replicate some a consistently (i.e. evaluate the same a for all our examples using the same value of k) we know that $k = k_1$ and $a = a_1$.

This attack utilizes minimal memory and takes 2^n time

1.3 $F_3 : \{0,1\}^{n+l} \times \{0,1\}^l \to \{0,1\}^l$

$F_3(k_1||a_1, m) = E(k_1, m) \oplus a_1$

$F_3^{-1}(k_1||a_1, c) = E^{-1}(k_1, c \oplus a_1)$

Algorithm 3: *Attack*$((m_1, c_1), (m_2, c_2), ..., (m_t, c_t))$

```
for k ∈ {0,1}^n do
    x ← E(k, m_1)||E(k, m_2)||...||E(k, m_t)
    H[x] ← k
end
for a ∈ {0,1}^l do
    x ← c_1 ⊕ a||c_2 ⊕ a||...||c_t ⊕ a
    if H[x] ≠ ⊥ then
        k ← H[x]
        return k||a
    end
end
```

This attack works because, for a given (m, c) plaintext-ciphertext pair, the following should for hold the correct values of k_1 and a_1:

$E(k_1, m) = c \oplus a_1$

This attack utilizes 2^n memory and takes $2^n + 2^l \approx 2^l$ time (assuming $n < l$)

1.4 $F_4 : \{0,1\}^{n+2l} \times \{0,1\}^l \to \{0,1\}^l$

$F_4(k_1||a_1||a_2, m) = E(k_1, m \oplus a_1) \oplus a_2$

$F_4^{-1}(k_1||a_1||a_2, c) = E^{-1}(k_1, c \oplus a_2) \oplus a_1$

Algorithm 4: $Attack((m_1, c_1), (m_2, c_2), ..., (m_t, c_t))$

for $a \in \{0,1\}^l$ **do**
  $x \leftarrow m_1 \oplus a || m_2 \oplus a || ... || m_t \oplus a$
  $H[x] \leftarrow a$
end
for $k_1 || a_2 \in \{0,1\}^{n+l}$ **do**
  $x \leftarrow E^{-1}(k_1, c_1 \oplus a_2) || E^{-1}(k_1, c_2 \oplus a_2) || ... || E^{-1}(k_1, c_t \oplus a_2)$
  if $H[x] \neq \perp$ **then**
    $a_1 \leftarrow H[x]$
    return $k_1 || a_1 || a_2$
  end
end

This attack works because, for a given (m, c) plaintext-ciphertext pair, the following should hold the correct values of k_1, a_1, and a_2:

$$m \oplus a_1 = E^{-1}(k_1, c \oplus a_2)$$

This attack utilizes 2^l space and takes $2^l + 2^{n+l} \approx 2^{n+l}$ time

2

The probability that the attack generates a psuedo collision in the Hash table is $\frac{2^{112}}{2^{64t}}$:

- Assume we've already populated our Hash table H with 2^{56} distinct and uniform bit strings $\in \{0,1\}^{64t}$
- Now consider that we're starting our second for-loop, in which we will generate 2^{56} more uniformly random bit strings $\in \{0,1\}^{64t}$. We ask: "what is the probability that a string x I generate in this step is already an index in H ($Pr[H[x] \neq \perp]$)?"
- Well, H contains 2^{56} examples from a uniformly random sample space of size 2^{64t}; since x is also uniformly random, $Pr[H[x] \neq \perp] = \frac{2^{56}}{2^{64t}}$ (the number of examples, divided by the size of the sample space)
- This tells us the probability that a single example x generated by our second for-loop causes a collision in H. Call this event E. We want to know the probability that any x generated by our second for-loop causes a collision in H. Call these set of events $\{E_1, E_2, ..., E_{2^{56}}\}$; we want to know $Pr[E_1 \cup E_2 \cup ... \cup E_{2^{56}}]$
- By applying Union Bound, we can say that $Pr[E_1 \cup E_2 \cup ... \cup E_{2^{56}}] \leq Pr[E_1] + Pr[E_2] + ... + Pr[E_{2^{56}}]$
- Therefore a good bound on the probability that an example of x generated in our second step causes a collision in H is $2^{56} * \frac{2^{56}}{2^{64t}} = \frac{2^{112}}{2^{64t}}$
- This offers itself as a good bound on the probability that our MITM attack generates a psuedo collision in H while trying to determine the keys used in the Encryption scheme we're trying to break

This probability is less than 1 when $t = 2$
This probability is less than 2^{-100} when $t = 4$

3

An SPN scheme on messages of length l with r encryption rounds contains the following:

- A key schedule comprised of $r + 1$ keys $\in \{0,1\}^l$. This set is not known to an adversary trying to break the SPN.
- A $t \times r$ table of S-boxes; $t * 8 = l$ (Each S-Box operates on one byte of data). This set is known to an adversary trying to break the SPN.
- A set of r P-boxes. This set is known to an adversary trying to break the SPN.

When run conventionally, the key schedule of an SPN is hard to recover; an exhaustive key schedule search would take 2^{rl} time. However, given the proposed modification, there exists a much fast attack. Let's call the modified encryption scheme SPN'. Like the original SPN, SPN' contains a key schedule K, a table of S-boxes S, and a set of P-boxes P. Unlike SPN, which computes a cipher text from a message by piping it through rounds of the transformations as described in K, S, and P, SPN' works by:

1. Performing all $r+1$ key mixing transformations, as described in K
2. Performing all r S-box transformations, as described in S
3. Performing all r P-box transformations, as described in P

We observe that, even knowing the contents of S and P, SPN is hard to crack because the contents of K are unknown and, if chosen well, will intersperse new bits into cipher texts as they are permuted by the transformations in S and P. Critically, we must realize two properties of SPN', that will let us radically simplify our understanding of how SPN' works:

1. Because we apply all the key mixing steps in K first, without permuting any bits between key mixing steps, we can understand the first step of SPN' as applying a single key mixing step to our original message m with a single key k, such that k is equivalent to all the keys in K xor'ed together.
2. Because SPN' doesn't intersperse bits between applications of transformations described in S and P, we can understand next two steps of SPN' as permuting the bits outputted by the first step of SPN'

Therefore we can say that, given a message m, running m through SPN' (with a key schedule K, S-box table S, and P-Box set P) is equivalent to performing the following transformation on m:

$SPN'(m) = jumble(m \oplus k)$

Where k is the combined key, derived from K, and $jumble(x)$ is a function that permutes the bits in a bit-string x and is equivalent to performing all the transformations described in S and then all the transformations described in P on the string x.

Since S and P are already known by an adversary, we can say that an adversary also knows the defintion of $jumlbe(x)$, as well as $jumble^{-1}(x)$; assume both of these functions can be run efficiently. Therefore, SPN' can be broken by finding a usable key k s.t. k is equivalent to xor'ing together all the keys in the key schedule K. Assuming we have t examples of (m, c) plaintext-ciphertext examples under SPN' with a key schedule K, we can feasibly attack SPN' by performing an exhaustive key-search on the sample space of a single key in K, $\{0,1\}^l$:

Algorithm 5: $Attack((m_1, c_1), (m_2, c_2), ..., (m_t, c_t))$

for $k \in \{0,1\}^l$ **do**
  $x \leftarrow k \oplus m_1 || k \oplus m_2 || ... || k \oplus m_t$
  $y \leftarrow jumble^{-1}(c_1) || jumble^{-1}(c_2) || ... || jumble^{-1}(c_t)$
  if $x == y$ **then**
    return k
  end
end

We can say that our output k is a (likely) correct (if t is large enough), and can be used to decrypt cipher texts encrypted under our instance of SPN'

Our attack therefore breaks SPN' in time 2^l and utilizes minimal memory, which is far too easy given that our original key schedule sample space for SPN was 2^{lr}

4

Here we consider how to break any 2-round SPN with a 64-bit block size. Given such an SPN, let $SP_1(x)$ describe the process of running a 64-bit string x though its first set of S-box and P-box steps, and $SP_2(x)$ describe the process of running a 64-bit string x though its second set of S-box and P-box steps. Let $SP_1^{-1}(x)$ and $SP_2^{-1}(x)$ describe the inverses of these processes, respectively. Assume that we can evaluate these processes efficiently for all $x \in \{0,1\}^{64}$.

With this construction, we can describe the output of running a given 2-round, 64-bit SPN with a key schedule $K = \{k_1, k_2, k_3\}$ on any message m as:

$SPN(m) = SP_2(SP_1(m \oplus k_1) \oplus k_2) \oplus k_3$ and $SPN^{-1}(c) = SP_1^{-1}(SP_2^{-1}(c \oplus k_3) \oplus k_2) \oplus k_1$

We will use this construction to show that the attacks we define below are sound. Assume that in either attack we have t examples of (m, c) plaintext-ciphertext examples under our given 2-round SPN with key schedule K.

4.1

Beginning with the construction of SPN we sketched above, we observe that for a given (m, c) plaintext-ciphertext pair in our example set, the following should hold for the correct values of k_1, k_2, and k_3:

$SP_1(m \oplus k_1) = SP_2^{-1}(c \oplus k_3) \oplus k_2$

Therefore, assuming that the three round keys are uniformly random and independent of one another, we can implement the following Meet-In-The-Middle attack to recover them. Assume that the attack has access to a (perfect) hash table H of size 2^{64}

Algorithm 6: $Attack((m_1, c_1), (m_2, c_2), ..., (m_t, c_t))$

for $k \in \{0,1\}^{64}$ **do**
    $x \leftarrow SP_1(m_1 \oplus k)||SP_1(m_2 \oplus k)||...||SP_1(m_t \oplus k)$
    $H[x] \leftarrow k$
end
for $k_2||k_3 \in \{0,1\}^{128}$ **do**
    $x \leftarrow SP_2^{-1}(c_1 \oplus k_3) \oplus k_2||SP_2^{-1}(c_2 \oplus k_3) \oplus k_2||...||SP_2^{-1}(c_t \oplus k_3) \oplus k_2$
    if $H[x] \neq \perp$ **then**
        $k_1 \leftarrow H[x]$
        return $k_1||k_2||k_3$
    end
end

This attack utilizes 2^{64} space and takes $2^{64} + 2^{128} \approx 2^{128}$ time

4.2

If we can assume that the round keys are uniformly random but $k_1 = k_3$ (while k_2 remains independent) we can implement a much faster attack on our given 2-round SPN. For any (m, c) plaintext-ciphertext pair in our example set, if we evaluate the following:

$m' = SP_1(m \oplus k_1)$, $c' = SP_2^{-1}(c \oplus k_3)$

We observe that:

$m' \oplus k_2 = c'$, so $m' \oplus c' = k_2$

In other words, if we find a $\hat{k}_1, \hat{k}_3$ such that we can replicate some $\hat{k}_2$ consistently (i.e. evaluate the same $\hat{k}_2$ for all our examples using the same values of $\hat{k}_1$ and $\hat{k}_3$) we know that $\hat{k}_1 = k_1$, $\hat{k}_2 = k_2$, and $\hat{k}_3 = k_3$ (or, at the very least, these are defensible inductive guesses for the round keys, given2 our example set). Since we know that $k_1 = k_3$ we can run the following attack:

Algorithm 7: $Attack((m_1, c_1), (m_2, c_2), ..., (m_t, c_t))$

for $k \in \{0,1\}^{64}$ **do**
    $x \leftarrow SP_1(m_1 \oplus k)||SP_1(m_2 \oplus k)||...||SP_1(m_t \oplus k)$
    $y \leftarrow SP_2^{-1}(c_1 \oplus k)||SP_2^{-1}(c_2 \oplus k)||...||SP_2^{-1}(c_t \oplus k)$
    $z_1||z_2||...||z_t \leftarrow x \oplus y$
    if $\forall i, j \in [t], i \neq j, z_i == z_j$ **then**
        return $k||z_1||k$
    end
end

This attack uses minimal memory and takes 2^{64} time.

5

We define the PRP advantage of an attack A on a block-cipher $E : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l$ as:

$Adv_E^{PRP}(A) = |Pr_K[A^{E(k,\cdot)} = 1] - Pr_\Pi[A^{\pi(\cdot)} = 1]|$

where k is a uniformly random variable on $K = \{0,1\}^n$ and π is a uniformly random permutations on Π = the set of permutations on bit strings of length l

We define the following adversaries on our set of block ciphers, each built from a block cipher $E : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l$. Each adversary has oracle access O to its respective block cipher or a random permutation.

5.1

$F_1 : \{0,1\}^n \times \{0,1\}^{2l} \to \{0,1\}^{2l}$
$F_1(k,x) = E(k, x[1])||x[2]$

Algorithm 8: A_1

$x \leftarrow O(\{1\}^l||\{0\}^l)$
if $x[2] == \{0\}^l$ **then**
  return 1
end
return 0

This adversary makes one query, with a constant runtime, assuming E is efficient

We can evaluate the following:

$Pr_K[A_1^{F_1(k,\cdot)} = 1] = 1$: Because F_1 does not encrypt the second half of its input, we know if we call $O(x)$ s.t. $x \in \{0,1\}^{2l}, x[2] = \{0\}^l$, our oracle should return a y s.t. $y \in \{0,1\}^{2l}, y[2] = \{0\}^l$, if the oracle was indeed submitting our request to F_1

$Pr_\Pi[A_1^{\pi(\cdot)} = 1] = \frac{1}{2^l}$: The probability that submitting any input x to the oracle and receiving an output y s.t. $y \in \{0,1\}^{2l}, y[2] = \{0\}^l$ has a probability of $\frac{1}{2^l}$ (equivalent to the probability that any substring in y is some fixed bit string of length l), if indeed the oracle was submitting our request to a random permutation π

Therefore:

$Adv_{F_1}^{PRP}(A_1) = |Pr_K[A_1^{F_1(k,\cdot)} = 1] - Pr_\Pi[A_1^{\pi(\cdot)} = 1]| = |1 - \frac{1}{2^l}| = 1 - \frac{1}{2^l}$

5.2

$F_2 : \{0,1\}^n \times \{0,1\}^{2l} \to \{0,1\}^{2l}$
$F_2(k,x) = E(k, x[1])||E(k, x[2])$

Algorithm 9: A_2

$x \leftarrow O(\{0\}^l||\{0\}^l)$
if $x[1] == x[2]$ **then**
  return 1
end
return 0

This adversary makes one query, with a constant runtime, assuming E is efficient

We can evaluate the following:

$Pr_K[A_2^{F_2(k,\cdot)} = 1] = 1$: Because F_2 encrypts both halves its input using the same block cipher E and key k, we know if we call $O(x)$ s.t. $x \in \{0,1\}^{2l}, x[1] == x[2]$, our oracle should return a y s.t. $y \in \{0,1\}^{2l}, y[1] == y[2]$, if the oracle was indeed submitting our request to F_2

$Pr_\Pi[A_2^{\pi(\cdot)} = 1] = \frac{1}{2^l}$: The probability that submitting any input x to the oracle and receiving an output y s.t. $y \in \{0,1\}^{2l}, y[1] == y[2]$ has a probability of $\frac{1}{2^l}$ (equivalent to the probability that any substring in y is some fixed bit string of length l), if indeed the oracle was submitting our request to a random permutation π

Therefore:

$Adv^{PRP}_{F_2}(A_2) = |Pr_K[A_2^{F_2(k,\cdot)} = 1] - Pr_\Pi[A_2^{\pi(\cdot)} = 1]| = |1 - \frac{1}{2^l}| = 1 - \frac{1}{2^l}$

5.3

$F_3 : \{0,1\}^n \times \{0,1\}^{2l} \rightarrow \{0,1\}^{2l}$
$F_3(k,x) = E(k, x[1] \oplus x[2]) || E(k, E(k, x[2]) \oplus x[1] \oplus x[2])$

Algorithm 10: A_3

```
x ← O({0}^{2l})
y ← O({1}^{2l})
if x[1] == y[1] then
    return 1
end
return 0
```

This adversary makes two queries, with a constant runtime, assuming E is efficient

We can evaluate the following:

$Pr_K[A_3^{F_3(k,\cdot)} = 1] = 1$: The first half of any output y from $F_3(k,x)$ is equivalent to $E(k, x[1] \oplus x[2])$. Therefore if we call $O(x)$ s.t. $x \in \{0,1\}^{2l}, x[1] == x[2]$, our oracle should return $E(k, x[1] \oplus x[2])||z = E(k, \{0\}^l)||z$ (where $z \in \{0,1\}^l$) if it was indeed submitting our request to F_3. Therefore, if we submit two queries too our oracle s.t. for both inputs $x, y \in \{0,1\}^{2l}, x \neq y, x[1] == x[2], y[1] == y[2]$, and receive outputs a and b, then if our oracle submitted our requests to F_3 then $a[1] = b[1] = E(k, \{0\}^l)$

$Pr_\Pi[A_3^{\pi(\cdot)} = 1] = \frac{1}{2^l}$: Say that our oracle was submitting queries to a random π on Π, and that we feed it two valid queries, $x, y \in \{0,1\}^l$. Say we receive an output a from our query $O(x)$, and b from $O(y)$. Both a and b are random permutations. Let $a[1] = z$. The probability that $b[1] = z$ is $\frac{1}{2^l}$ (equivalent to the probability that any substring in y is some fixed bit string of length l).

Therefore:

$Adv^{PRP}_{F_3}(A_3) = |Pr_K[A_3^{F_3(k,\cdot)} = 1] - Pr_\Pi[A_3^{\pi(\cdot)} = 1]| = |1 - \frac{1}{2^l}| = 1 - \frac{1}{2^l}$