Cryptology I (spring 2019) Dominique Unruh

Exercise Sheet 2

Out: 2019-03-23 Due: 2019-04-06

Problem 1: Linear block ciphers

Consider a block cipher E taking $2n$ bits to $2n$ bits that consists of a Feistel network with a round function $F(k, m^{half})$ that has the following property:

For each possible key k, there exists an $n \times n$ matrix $A^{(k)}$ such that for all n-bit messages m^{half}, we have $F(k, m^{half}) = A^{(k)} m^{half}$. (I.e., for a fixed key, F is linear.)

(Reminder: since we operate on bits, matrix multiplication is done modulo 2, i.e., $(A^{(k)} m^{half})_i = \sum_j A^{(k)}_{ij} m^{half}_j \mod 2$.)

We will show that this block cipher cannot be secure under chosen plaintext attacks (by giving an attack).

Note: Even if you do not manage to solve the first parts of this problem, you can still solve the later parts by just assuming that you have solved the first ones.

(a) Let m be a $2n$-bit message within the Feistel network, before swapping the two halves. Construct a $2n \times 2n$ matrix B such that Bm is the result of swapping the two halves. (I.e., the first half of m is the second half of Bm and vice versa.)

(b) Let $m = m_1 \| m_2$ be a $2n$-bit message within the Feistel network. Let $m' = (m_1 \oplus F(k, m_2)) \| m_2$ be the result of applying F.

Construct a matrix $C^{(k)}$ such that $m' = C^{(k)} m$ for all m.

Hint: Try to first write down a formula (involving a sum) for the i-th bit of m'. Then you can read off the definition of $C^{(k)}$. $C^{(k)}$ may depend on $A^{(k)}$.

(c) Give a formula for the matrix $D^{(k)}$ such that $D^{(k)} m = E(k, m)$ for all m. $D^{(k)}$ may depend on B and $C^{(k)}$.

(d) Let m_i be the message $0 \dots 010 \dots 0$ with 1 on the i-th position. What is $D^{(k)} m_i$? How can we reconstruct the whole matrix $D^{(k)}$ given $D^{(k)} m_i$ for all i (efficiently)?

Note: You do not actually need the formulas derived in the previous steps. The only reason for finding those formulas was to show that the matrix $D^{(k)}$ exists at all.

(e) You are given a ciphertext $c = E(k, m)$ and you are allowed to make chosen plaintext queries (i.e., you may ask for $E(k, m')$ for any message m'). How do you find out m?

Hint: Find $D^{(k)}$ first.

(f) Explain why DES is susceptible to the attack described above if the S-boxes are linear.[1]

Problem 2: Security definitions

Your task is to write a security definition in Python (or another language, but we provide a template in Python). The goal of this is to give you a better understanding what security definitions mean, besides just being formulas. We illustrate this by writing the security definition of PRGs in Python:

```
#!/usr/bin/python3

# For simplicity, we fix domain and range of PRGs here:
# The domain is the set of 32-bit integers
# The random is the set of ten-element lists of 32-bit integers
#   (equivalent to 320-bit integers)

import random

# A very bad pseudo-random generator
# Seed is supposed to be a 32-bit integer
# Output is a ten element list of 32-bit integers
def G(seed):
    return [4267243**i*seed % 2**32 for i in range(1,11)]

# The game: it gets a prg and an adversary as arguments
def prg_game(G,adv):
    b = random.randint(0,1) # Random bit
    seed = random.randint(0,2**32-1) # Random seed
    rand = [random.randint(0,2**32-1) for i in range(10)] # Truly random output
    if b==0: badv = adv(G(seed))
    else: badv = adv(rand)
    return b==badv

def adv(rand):
    if rand[1]==4267243*rand[0] % 2**32: return 0
    else: return 1

def test_prg(G,adv):
    num_true = 0
    num_tries = 100000
    for i in range(num_tries):
```

[1] I.e., for each S-box, each output bit is a linear combination of the input bits.

```
        if prg_game(G,adv): num_true += 1
    ratio = float(num_true)/num_tries
    print(ratio)

# An output near 0.5 means no attack
# An output neat 0.0 or 1.0 means a successful attack
test_prg(G,adv)
```

Here `G` is an implementation of a pseudo-random generator (a rather bad one). And `prg_game` is a function that implements the game from the security definition of PRGs. That is, it takes a PRG `G`, and an adversary `adv`, and calls `adv` either with randomness or the output of `G`. If the adversary guesses correctly which of the two was the case, `prg_game` returns `True`, else `False`.

The function `test_prg` tries out whether a given adversary is successful or not by counting how often he guesses right. (Of course, this does not replace a proof: a statistic does not give certainty, and also we cannot know whether other adversaries are successful. But it illustrates the use of the security definition.)

We have also written an example adversary `adv` that breaks the PRG `G`. For simplicity, let both the message and the key space consist of 32-bit integers. But note that you are not supposed to use brute force attacks.

Your task:

(a) Write the security definition for IND-OT-CPA as a Python program. (Recall, in IND-OT-CPA, the adversary is called twice, so you will need two functions `adv1` and `adv2`. Also pay attention to the following: the adversary should not be allowed to output messages that are not in the message space.)

(b) Write an adversary that breaks the encryption scheme *enc* defined in the source code below. (This adversary should have a success probability, as measured by `test_indotcpa` of at least 0.95.)

Here is a template for your solution. You need to fill in code where there are `???`.

```
#!/usr/bin/python

# For simplicity, the message space and the key space consists of 32-bit integers
# And the ciphertext space consists of all integers (possibly longer)

import random

# A bad encryption scheme
def enc(key,msg):
    return key*msg

# The IND-OT-CPA game
def indotcpa_game(enc,adv1,adv2):
```

```
    ???

# The adversary breaking the above encryption scheme
# (Consisting of two functions, since the adversary is invoked twice by the game)
def adv1():
    ???
def adv2(c):
    ???

def test_indotcpa(enc,adv1,adv2):
    num_true = 0
    num_tries = 100000
    for i in range(num_tries):
        if indotcpa_game(enc,adv1,adv2): num_true += 1
    ratio = num_true/num_tries
    print(ratio)

# An output near 0.5 means no attack
# An output neat 0.0 or 1.0 means a successful attack
test_indotcpa(enc,adv1,adv2)
```

Problem 3: Reduction proofs

In the lecture, when we proved that $E(k,m) := G(k) \oplus m$ is $(\tau - O(n), 2\varepsilon)$-IND-OT-CPA secure when G is an (τ,ε)-PRG, we left the following claim unproven:

$$\left|\Pr[b = 1 : Game\ 1] - \Pr[b = 1 : Game\ 2]\right| \leq \varepsilon \tag{1}$$

where

Game 1. $k \xleftarrow{\$} K$, $(m_0, m_1) \leftarrow A()$, $c \leftarrow G(k) \oplus m_0$, $b \leftarrow A(c)$. $\diamond$

and

Game 2. $p \xleftarrow{\$} M$, $(m_0, m_1) \leftarrow A()$, $c \leftarrow p \oplus m_0$, $b \leftarrow A(c)$. $\diamond$

and A is a $(\tau - O(n))$-time adversary and n is the length of $G(k)$.[2]

We said that (1) follows from the fact that G is a (τ,ε)-PRG, and that means that $G(k)$ in Game 1 cannot be distinguished from p in Game 2 by definition of PRGs.

We will now prove this formally:

Proof of (1)**.** We define an algorithm $B(p)$ that performs the following steps: $(m_0, m_1) \leftarrow A()$, $c \leftarrow \boxed{1}$, $b \leftarrow A(c)$, return b.

[2]This is formally not meaningful because $O(n)$ just stands for some function $f \in O(n)$, but we have not specified which. A precise proof would specify concretely which function f it is but we skip this in order not to get too much into machine-model dependent details.

Consider the following game:

Game 3. $k \xleftarrow{\$} K$, $p \leftarrow G(k)$, $b \leftarrow B(p)$. $\diamond$

By unfolding the definition of B, and reordering of some of the program steps, we immediately see that Game 1 and Game 3 do the same, formally

$$\Pr[b = 1 : \mathit{Game}\ 1] = \Pr[b = 1 : \mathit{Game}\ 3] \tag{2}$$

Furthermore, the step $c \leftarrow p \oplus m_0$ takes [2] runtime steps. Since A is a $(\tau - O(n))$-time algorithm, this implies that B is a τ-time algorithm.

Consider the following game:

Game 4. [3], $b \leftarrow B(p)$. $\diamond$

Since B is a τ-time algorithm, and since G is a (τ, ε)-PRG, by definition of PRGs we have

$$\left|\Pr[b = 1 : \mathit{Game}\ 3] - \Pr[b = 1 : \mathit{Game}\ 4]\right| \leq [4]. \tag{3}$$

Finally, by [5], we show

$$\Pr[b = 1 : \mathit{Game}\ 2] = \Pr[b = 1 : \mathit{Game}\ 4]. \tag{4}$$

From [6], we immediate conclude (1). $\square$

Task: Fill in the gaps. (The length of the boxes gives no indication of the length of the missing text.)

Problem 4: One-time-pad in CBC mode (bonus problem)

Assume someone uses the one-time pad in CBC mode. That is, the block cipher is $E(k, m_{block}) := k \oplus m_{block}$, and that block cipher is used in CBC mode.

(a) Assume a message $m = m_1 \| m_2 \| m_3 \| m_4$ is encrypted where all $m_1 = m_2 = m_3 = m_4$ are blocks consisting only of only zeroes.

What is the resulting ciphertext? (That is, give an explicit simple formula for each of the ciphertext blocks.)

(b) Assume a message $m = m_1 \| m_2 \| m_3 \| m_4$ is encrypted. What is the resulting ciphertext? (Give a formula in terms of the m_1, m_2, m_3, m_4, simplified as much as possible.)

(c) Explain how to compute $m_3 \oplus m_4$ from the resulting ciphertext. (Without using the key.)

(d) Explain why the above implies that the one-time pad in CBC mode is not IND-CPA secure (not even IND-OT-CPA).

Problem 5: "Inverse" CBC

Consider the following mode of operation (which I call "inverse CBC"):

To encrypt a message m consisting of blocks $m_1, \ldots, m_n$ with key k, pick a random initialization vector iv and then compute $c_1 := E_0(k, m_1) \oplus iv$ and $c_i := E_0(k, m_i) \oplus m_{i-1}$ for $i = 2, \ldots, n$. Here E_0 is the block cipher. And $E(k, m) := iv \| c_1 \| \ldots \| c_n$.

The adversary has intercepted a ciphertext $c = E(k, m)$. He happens to know the last block m_n of m (e.g., because that one is prescribed by the protocol).

(a) Explain how the adversary can completely decrypt m. He can make chosen plaintext queries (i.e., he can ask for encryptions of arbitrary message m'). He cannot make decryption queries.

Hint: First think how you can, e.g., find out $E_0(k, m_n)$ by performing an encryption query $E(k, m_n)$.

(b) Suggest how to fix the mode of operation so that it becomes secure at least again this attack (and simple modifications thereof). You do not need to prove security.

Problem 6: Breaking ECB (Bonus points)

In the lecture we have seen that encrypting a file with ECB mode is not very secure. For example, if an uncompressed image file is encrypted, the result may still reveal much of the picture to the naked eye.

In this exercise, we consider the task of distinguishing the encryption of two given messages m_0, m_1 automatically. That is, assume that two messages m_0, m_1 (English texts) of the same length are given and known to the adversary. Furthermore, the adversary learns c, which is the ECB encryption of m_0 or m_1 (using a random and unknown key k). The adversary is now supposed to guess which message was encrypted. (I.e., we have a known plaintext attack, not a chosen plaintext attack.)

(a) Describe an algorithm that finds out (given m_0, m_1, c) whether m_0 or m_1 was encrypted. It should work on "typical" text files. (That is, it should not require, e.g., one of the text files to contain only spaces or similar.)

Example of "typical" text files are `ecb-distinguish-1.txt` and `ecb-distinguish-2.txt` from the lecture webpage.

(b) Implement the algorithm. That is, fill in the missing code for the function `adv` in the code below (also available on the lecture webpage):

```
#!/usr/bin/python3

# "Crypto" might need "pip install pycrypto" if it's not installed
```

```
import Crypto, random
from Crypto.Cipher import AES

def int_to_bytes(i,len): # Not optimized
    res = []
    for j in range(len):
        res.append(i%256)
        i = i>>8
    return bytes(res)

def aes_ecb_enc(k,m):
    from Crypto import Random
    assert isinstance(m,bytes)
    assert len(m)%AES.block_size == 0, len(m)%AES.block_size
    k = int_to_bytes(k,AES.block_size)
    cipher = AES.new(k, AES.MODE_ECB)
    return cipher.encrypt(m)

def aes_ecb_dec(k,m):
    from Crypto import Random
    assert isinstance(m,bytes)
    k = int_to_bytes(k,AES.block_size)
    cipher = AES.new(k, AES.MODE_ECB)
    return cipher.decrypt(m)

# Just a test
assert aes_ecb_dec(2123414234,aes_ecb_enc(2123414234,b'hello there test')) == b'hello t


# The game: it gets a prg and an adversary as arguments,
# as well as the messages to be distinguished
def guessing_game(adv,m0,m1):
    b = random.randint(0,1) # Random bit
    k = random.getrandbits(256) # Random AES key
    seed = random.randint(0,2**32-1) # Random seed
    rand = [random.randint(0,2**32-1) for i in range(10)] # Truly random output
    msg = (m0,m1)[b]
    ciph = aes_ecb_enc(k,msg)
    badv = adv(ciph)
    return b==badv


def adv(ciph):
```

```
    # blocks contains the ciphertext as a list of blocks
    blocks = [ciph[i*AES.block_size:(i+1)*AES.block_size] for i in range(len(ciph)//AES

    ???
    return ??? # return 0 or 1


def test_adv(adv):
    num_true = 0
    num_tries = 3000
    m0 = open("ecb-distinguish-1.txt","rb").read()
    m1 = open("ecb-distinguish-2.txt","rb").read()
    for i in range(num_tries):
        #if i%100==0: print(str(i)+"...")
        if guessing_game(adv,m0,m1): num_true += 1
    ratio = float(num_true)/num_tries
    print(ratio)

# An output near 0.5 means no attack
# An output neat 0.0 or 1.0 means a successful attack
test_adv(adv)
```