Cryptology I (spring 2015) Dominique Unruh

Exercise Sheet 2

Out: 2015-03-09 Due: 2015-03-18

Problem 1: One-time-pad in CBC mode

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 2: "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.

(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 3: Security definitions

Your task is to write a security definition in Python (or another language, but we provide a template in Python). We illustrate this by writing the security definition of PRGs in Python:

```
#!/usr/bin/python

# 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):
        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.

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.)

(c) **[Bonus points]** Write the definition of IND-CPA. (For this, note that in Python, you can define functions within the body of other functions, and pass such functions are arguments to other functions. This is very convenient for implementing oracles.)

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 of 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 = 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_indotcpa(enc,adv1,adv2)
```