Project 3

Introduction

You are interning at the super secretive TLA (Three Letter Agency) once again. This time, TLA is trying to learn the secret key of the RIA (Rival Intelligence Agency). TLA has discovered a server that processes incoming messages to RIA from their spies abroad. TLA has sent some test ciphertexts to the server, to see what it does. Here is what TLA knows about the server:

- The server only ever appears to respond to an incoming message with 0. Presumably this is a "message received" indicator
- The server is using some sort of ElGamal-based encryption scheme, where the ciphertexts consist of a pair (c_0, c_1) of group elements. The group is the subgroup of order p of $\mathbb{Z}_q^*$, where p, q are primes and $q = 2p + 1$.

  Therefore, the first step of decryption is to compute $(c_0^d \bmod q) \times c_1$, where d is the secret decryption key. The rest of the decryption procedure is irrelevant. Presumably the overall encryption scheme CPA-secure
- There is little hope of figuring out the decryption key just based on the response from the RIA server (since it always returns 0). However, the TLA has noticed some timing fluctuations based on the ciphertexts sent. TLA believes that these variations are due to variations in the amount of time it takes to perform the exponentiation for decryption.

Your goal is to exploit this timing information to learn the secret decryption exponent d.

1 Part 1: Theory

The TLA believes that the RIA server is using the following repeated-squaring procedure for its exponentiation:

function `PowerMod`(`base`, `exponent`, `modulus`)
  `tot` $\leftarrow 1$
  Let ℓ be the bit-length of `exponent`

```
    for i = 1, i <= ℓ, i ← i + 1 do
        tot = tot^2
        tot = tot mod mod
        if exponent_i == 1 then
            tot = tot × base
            tot = tot mod mod
        end if
    end forreturn tot
end function
```

That is, the algorithm stars from the most significant bit of `exponent` and works its way down to the least significant bit.

First, notice that the running time of `PowerMod` depends on both the bit-length of `exponent` (which determines the number of loops) and the number of 1s in `exponent` (which determines how many multiplications are performed). Therefore, if we new exactly how long each operation took, we could learn something about the secret exponent by looking at the overall running time on, say, a random input. Unfortunately, we don't know what architexture the server is using, and there may be additional delays (say due to network latency) that make it hard to predict the running time. Instead, we need some way to learn the exponent, without a priori knowing anything about the running time.

To do so, we will exploit a common optimization. Modular reductions are expensive computations. Therefore, a common approach to saving time when computing x mod y is to first check if $x < y$; if so, then x mod $y = x$, and so no reduction is necessary. The expensive mod operation is then only performed when $x \geq y$. This optimization certainly makes things run faster, but it also means the time is dependent on the data itself. TLA believes the the mod operations in the server's exponentiation algorithm are carried out using this optimized modular reduction. As we will see, this leads to a serious security vulnerability. The key insight is that whether or not a particular mod operation is performed is no dependent on the data itself. By carefully crafting the ciphertexts sent to the server, and observing how long different ciphertexts take, we will be able to determine the bits of the decryption exponent.

We will assume the server, on ciphertext (c_0, c_1), runs `PowerMod`(c_0, d, q). We will assume the exponentiation algorithm above accounts for the majority of the server's decryption time, and that the two optimized modular reductions in the loop account for a significant fraction of the running time. Moreover, we will assume that all operations except these two modular reductions take time independent of the data; the only data-depending timing are the modular reductions.

(a) Suppose first that there is no noise in the running time of the computation or in the network delays. Explain how to find the second most significant bit of d

(the most significant is of course 1) by making two queries to the server. The key here is to craft two ciphertexts $(c_0, c_1), (c'_0, c'_1)$ such that the relative number of modular reductions in both cases reveals the most significant bit of d. You have a choice of which mod to attack: the one after the squaring operation, or the one after the multiplication. Both choices can be used to mount the attack.

(b) Next, extend the above to compute all of the digits of d, and to moreover handle variations in the network delay and running times of the other operations. Note that here, as far as I know, it is only possible to attack one of the mods once you've moved past the first few bits.

2 Part 2: Mounting the Attack

The file "RIAserver.pyc" implements the RIA server in python. We have also provided a file "call_server.py" which shows how to make calls to the server. You will need to make many calls in order to get accurate timing information, so we have also shown how to make parallel calls to "RIAserver.pyc". "call_server.py" also contains q, which TLA has managed to determine. TLA has managed to uncover that d is an integer of about 160 bits. The first two bits are both 1.

Use what you learned in **Part 1** to determine d.

You will notice that not all timing variations are due to d. In particular, if you run "call_server.py" as is, you will find that sometimes, ciphertexts take orders of magnitude longer than other. TLA suspects that the very fast ciphertexts are not being decrypted at all and instead are being rejected. Can you figure out what ciphertexts are being rejected?

3 Part 3: Decrypting

To test if the d you found is correct, try decrypting the contents of the file "ctxt.txt". This file contains 8 ciphertexts intercepted by TLA.

4 Deliverables

Your submission will contain the following files:

- `writeup.pdf`, which will contain a write-up for the theory component (Part 1) above. In a separate section, discuss your actual attack. Did any issues arise? About how long does each modular reduction take? About how many queries

did you need to make to the server to uncover each bit of d? What were the plaintexts that you recovered?

- Any code you used

Submit your files to: https://dropbox.cs.princeton.edu/COS433_S2018/Project_3

5 Grading

Your grade will be determined as follows:

- Writeup: 75 points. This will be based on the thoroughness of your analysis and discussion of your attack
- Cryptanalysis: 25 points, broken down as follows:
  - 5 points for the first 7 bits of d (the first two bits you already know are both 1)
  - 15 points for the remaining digits of d
  - 5 points for decrypting the ciphertexts in "ctxt.txt"

6 Hints

On my machine, I was able to make 512 queries in parallel. Each bit took under 30 seconds, or about 10000 queries to the server. Once I had everything set up, I was able to recover d un under 2 hours.