Project 3/Homework 8

Project 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.

Background

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

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

```
        for $i = 1, i <= \ell, i \leftarrow i + 1$ do
            tot = $\texttt{tot}^2$
            tot = tot mod mod
            if $\texttt{exponent}_i == 1$ then
                tot = tot × base
                tot = tot mod mod
            end if
        end for
    return 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.

1 Problem 1 (10 points)

Suppose you knew exactly how long every operation on the server takes, as well as the exact network delay between you and the server. Additionally, suppose you knew the exact bit-length of d. Explain how to find the number of 1s in d.

2 Problem 2 (30 points)

Unfortunately, we do not know the exact network delay. We also do not know what architecture the server is running on, so we cannot possibly know the exact time each operation takes, and we don't know what other operations the server is performing after decryption. We don't necessarily know the bit-length of d. Moreover, we don't just care about the number of 1s in d, but actually want to figure out what d is. So 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 $x \bmod y = x$, 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 now 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.

Suppose 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 second most significant bit of d. That is, knowing nothing about the running times of the various components, but only the fact that a modular reduction takes a noticeable amount of time, the decryption times for the two ciphertexts should reveal the desired bit of d.

In order to devise your attack, consider the case where c_0 is a random value in $\mathbb{Z}_q^*$. How many modular reductions do you expect will be performed? Try to find an interval $[a, b]$ such that a random c_0 in $[a, b]$ will either have the same number of modular reductions or a different number of reductions, depending on the 2nd bit of d. You may assume you know both the length of q, as well as several of the most significant digits. You should not need to know q exactly.

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.

3 Problem 3 (10 points)

Explain how to account for noise in the running time of the computation or variations in network delays. To do so, you will need to send many ciphertexts to the server. Note that, at the beginning, you don't know how long a modular reduction takes, or the magnitude of the timing variations. Your attack should account for this. Note that the number of queries in your attack will depend on the time for a modular reduction and the magnitude of timing variations, so you will not know ahead of time how long the attack takes. You do not need to work out the exact number of queries you will need, but you should explain how you will know when you have made enough queries.

4 Problem 4 (10 points)

Explain how to extend the attack from Problem 2 to compute the next few bits of d. Again, your attack will work by carefully selecting various intervals $[a, b]$ and looking at the time it takes to decrypt random messages from those intervals. Note that your attack will need to compute the bits in order: computing the 3rd bit requires the 2nd to already be known, computing the 4th bit requires the 3rd, etc.

5 Problem 5 (30 points)

The most straightforward way to solve Problem 3 will fail after computing relatively few bits of d. Explain why. Devise a more sophisticated attack — one that chooses c_0 not from an interval but from a more sophisticated distribution. Your attack should be able to recover all the bits of d. For this part, you may assume the modulus q is known exactly.

6 Problem 6 (10 points)

When TLA goes to carry out your attack, the notice something odd: some ciphertexts take a very short time to get a response, whereas others take much, much longer. This of course cannot be explained by the minute difference in running time cause by the modular reduction. You suspect that the quick ciphertexts are simply being rejected by the server, without going through decryption. Can you come up with some reasons why?

Homework Problems

After successfully learning the decryption key, TLA has asked you to solve the following additional problems.

7 Problem 7 (25 points)

Recall that a graph G is 3-colorable if each node of the graph can be painted one of three colors (say, red, green, and blue, which we will associate with the numbers 1,2,3) such that, for every edge in G, the endpoints of that edge are painted different colors. Notice that it is easy to verify that a 3-coloring is valid by just checking all of the edges and making sure the endpoints are different. Therefore, 3-coloring is in NP, where the witness for G being 3-colorable is just the 3-coloring.

Consider the following simple proof. Choose a random permutation σ on the colors $\{1, 2, 3\}$. Given a coloring C, let $\sigma(C)$ be the coloring obtained by applying σ to the color of each node. To prove that a graph is 3-colorable, using a witness C, the prover simply sends $\sigma(C)$ to the receiver. The verifier then checks that $\sigma(C)$ is a valid 3-coloring.

(a) This proof system fails in one of the three required properties for a zero knowledge proof system (completeness, soundness, or zero knowledge). Which one fails?

In order to get around the problem above, instead imagine the following physical scheme using locked boxes. The prover will start with one lockable box for each node in G; for each node, she will write down the color of that node in $\sigma(C)$ and put it in the corresponding box. The prover will lock all the boxes, keep the keys, but send the locked boxes to the verifier. Of course, now the verifier has no hope of checking that $\sigma(C)$ is a valid 3-coloring. Instead, the verifier chooses a random edge $e = (u, v) \in G$, and sends e to the prover. The prover then sends the keys for those two nodes back to the verifier. The verifier opens the two boxes, and checks that the colors inside are different

(b) Prove that, if G is not 3-colorable, that a malicious prover has a significant chance of being caught (though it will potentially be $\ll 1$). Thus this scheme has a weak form of soundness.

(c) Prove that the scheme has (malicious verifier) zero knowledge. Assume the simulator can build its own locked boxes

The soundness of this physical scheme can be boosted by repeating the scheme many times sequentially, using a new random σ for each iteration. Thus, we can obtain a physical ZK proof.

(d) What crypto object should we use to implement the locked boxes above, to get a purely digital scheme? Re-prove soundness and zero knowledge for this new protocol.

Recall that 3-coloring is an NP complete language. By composing the scheme above with NP reductions, we can obtain a zero knowledge proof for all of NP

8 Problem 8 (25 Points)

A *random self reduction* is a procedure which turns any instance of a problem into a *random* instance of the problem.

For example, let p a prime, and $\mathbb{G}$ a group of order p. Suppose you are given a discrete log instance $(g, h = g^a)$, and suppose that $g \neq 1$ (so that g is a generator). Choose a random $r, s \in \mathbb{Z}_p$ such that $r \neq 0$ and let $g' = g^r$ and $h' = h^r \times g^s$.

(a) Show that, regardless of the original choice of (g, h), (g', h') is a uniformly random pair of group elements conditioned on $g' \neq 1$. Thus (g', h') is a random discrete log instance, independent of $(g, h,)$.

(b) Suppose you give someone (g', h') and they give you a discrete log b such that $(g')^b = h'$. Explain how to recover the discrete log of h, namely a.

A random self reduction therefore shows that, if there is *any* discrete log instance that is hard, a *random* discrete log instance is also hard. This means that there are no "extra hard" instances, since no instance is harder than the average case. The fact that discrete log admits a random self reduction means we can actually base hardness of the *worst case* version of the problem, rather than an average case problem. It can also be used to amplify the success probability of attacks:

(c) Suppose you have a discrete log adversary A that runs in time t, and solves random discrete log instances with probability ϵ. You know nothing about A except this fact: in particular, maybe A is deterministic, or maybe A is randomized.

Show how to use A to derive an adversary A' which solves discrete log with probability 99/100, but is allowed to run in time about $O(t/\epsilon)$.

Part (c) shows that, for our discrete log assumption, it is actually sufficient to assume that no polynomial time adversary can solve discrete log with high probability. This then implies that no polynomial time adversary can compute discrete logs with inverse polynomial advantage.

(d) Show such a random self reduction for DDH. That is, you are given a tuple $(g, u = g^a, v = g^b, w = g^c)$ where g is a generator, a and b are in $\mathbb{Z}_p$, and c is either $ab \bmod p$ or different than ab. We will call the $c = ab$ case a DDH tuple.

You must come up with a new tuple (g', u', v', w') such that:

- If (g, u, v, w) is a DDH tuple, then (g', u', v', w') is a *random* DDH tuple (it should be random even if (g, u, v, w) is a fixed tuple)
- If (g, u, v, w) is *not* a DDH tuple, then (g', u', v', w') is a truly random tuple of group elements, conditioned on g' being a generator and the tuple being *not* a DDH tuple.

The transformation from (g, u, v, w) to (g, u', v', w') must be efficient: you cannot compute discrete logs as part of the transformation.

Note that for part (d), the following simple transformation will not work: (g, u^r, v, w^r). This is a DDH tuple if (g, u, v, w) was a DDH tuple, and isn't a DDH tuple if (g, u, v, w) isn't. However, for a fixed tuple (g, u, v, w), (g, u^r, v, w^r) is not random: for example, the third component is fixed as v. While this transformation won't work, it is a useful starting point to think about.