Massachusetts Institute of Technology
6.5610: Applied Cryptography and Security
Professors Henry Corrigan-Gibbs and Yael Tauman Kalai

Handout 5
April 11, 2023
**Due:** April 21, 2023

# Problem Set 4

This problem set is due on *Friday, April 21, 2023* at **4:59 PM**. Please note our late submission penalty policy in the course information handout. Please submit your problem set, in PDF format, on Gradescope. *Each problem should be in a separate page.*

You are to work on this problem set in groups. For problem sets 1, 2, and 3, we will randomly assign the groups for the problem set. After problem set 3, you are to work on the following problem sets with groups of your choosing of size three or four. If you need help finding a group, try posting on Piazza. See the course website for our policy on collaboration. Each group member must independently write up and submit their own solutions.

*Homework must be typeset in* LaTeX *and submitted electronically!* Each problem answer must be provided as a separate page. Mark the top of each page with your group member names, the course number (6.5610), the problem set number and question, and the date. We have provided a template for LaTeX on the course website (see the *Psets* tab at the top of the page).

With the authors' permission, we may distribute our favorite solution to each problem as the "official" solution—this is your chance to become famous! If you do not wish for your homework to be used as an official solution, or if you wish that it only be used anonymously, please note this in your profile on your homework submission.

**Problem 4-1. BLS Signature Extension: Aggregation and Batching** Recall the BLS signature scheme shown in class. In this problem, we will explore ways to aggregate these signatures together and batch verification.

First, suppose that Alice and Bob both want to send signed messages to Carol. Alice has $(sk_a, vk_a) = (x_a, g^{x_a})$ and Bob has $(sk_b, vk_b) = (x_b, g^{x_b})$. Alice signs $m_a$ and obtains a signature $\sigma_a = H(m_a)^{x_a}$ and Bob likewise signs $m_b$ and obtains a signature $\sigma_b = H(m_b)^{x_b}$. An aggregate signature allows to combine two signatures into a single signature which has the size of a single signature and can be verified given the two messages and the corresponding two verification keys. Alice and Bob's signatures can be aggregated by simply multiplying them to obtain $\sigma = \sigma_a \cdot \sigma_b$.

**(a)** Describe the verification algorithm that verifies such as aggregate signature.

**(b)** Generalize this scheme to work with $n$ parties. How can $n$ parties combine their signatures into one aggregate signature, and how can the receiver verify this signature? As before, the aggregate signature should be no larger than one regular signature.

You just showed how to shrink the total size for $n$ signatures on $n$ different messages. However, verification is still linear in $n$. Next we will examine how signatures on the same message can be verified more efficiently.

Now, suppose Alice and Bob want to jointly sign the same message $m$. They both sign $m$ to get signatures $\sigma_a$ and $\sigma_b$, and they aggregate these signatures as above, by simply multiplying the two signatures to obtain $\sigma = \sigma_a \cdot \sigma_b$.

**(c)** How can Alice and Bob combine (or aggregate) their verification keys $vk_a$ and $vk_b$ so that Carol can verify the aggregate signature on $m$ with this joint key and the regular BLS verification algorithm applied to their joint (aggregated) key?

**(d)** Again, generalize your scheme to $n$ parties. How can $n$ parties combine their verification keys so their aggregate signature on one message $m$ can be verified using the joint (aggregated) key and the regular BLS verification algorithm?

**Problem 4-2. BLS Signature Extension: Blind Signatures**

There are three parties in a blind signature scheme: a user, a signer, and a verifier. The user has a message that they want the signer to sign, but the user does not want to reveal the message to the signer. Thus, the user disguises the message (blinds it) before sending it to the signer. The signer signs the blinded message. Then, the user unblinds the signature to reveal a signature on the message $m$ and sends this signature to the verifier.

One use case for blind signatures would be voting. A voter writes their vote down on a ballot, blinds it, and brings it to the official. Once the official has confirmed the voter's identity, the official signs the voter's blinded ballot without seeing the voter's specific vote. Then, the voter takes this signed ballot to the voting booth and unblinds it before turning it in.

Consider the following blind signature scheme: Let $G$ be an order $q$ group, and let $g \in G$ be a generator of $G$.

- $\mathsf{Gen}$: Samples a random $x \leftarrow Z_q$ and outputs $\mathsf{pk} = g^x$ and $\mathsf{sk} = x$.
- $\mathsf{Blind}(m)$: Chooses a random $r \leftarrow Z_q^*$ and outputs $\tilde{m} = H(m)^r$.
- $\mathsf{Sign}(\mathsf{sk}, \tilde{m})$: Outputs $\tilde{\sigma} = \tilde{m}^{\mathsf{sk}}$.
- $\mathsf{Unblind}(\tilde{\sigma}, r)$: Takes as input a signature of a blinded message $m$ and the randomness $r$ used to blind $m$ and it outputs $\sigma = \tilde{\sigma}^{r^{-1} \bmod q}$ as the signature for $m$.
- $\mathsf{Verify}(\mathsf{pk}, m, \sigma)$: The verifier accepts if and only if $e(\mathsf{pk}, H(m)) = e(\sigma, g)$.

**(a)** Show correctness for this scheme. That is, explain why the verifier will accept the output of unblind as a signature for $m$.

**(b)** Argue that this scheme is blinding. That is, the verifier who sees $\tilde{m}$ cannot link it to $m$.

**Problem 4-3. RSA attack**

**(a)** We will look at the broadcast attack from lecture (See lecture 14 notes). For this attack, several users encrypt the same message under different public keys. In the file `keys.txt`, you will find a list of RSA public keys. In the file `ciphertexts.txt`, you will find a list of ciphertexts of the same message. Each line of the file is encrypted with the key on the corresponding line `keys.txt`. In this problem, we use $e = 5$. A skeleton code file showing the encryption algorithm and reading these files is provided in `rsa_attack.py`. Find the message each of these ciphertexts encrypts for the answer to this problem part.

*Some hints for implementation:*

- To implement an inverse CRT, find $e = 5$ numbers that have residue $(1, 0, 0, 0, 0)$ mod $(n_1, n_2, n_3, n_4, n_5)$ respectively, $(0, 1, 0, 0, 0)$ mod $(n_1, n_2, n_3, n_4, n_5)$ respectively, etc. Then take the correct linear combination. To find these numbers, first consider what is the smallest positive integer whose residues are $(x, 0, 0, 0, 0)$ mod $(n_1, n_2, n_3, n_4, n_5)$ for some $x$? Or, in English, the first number that is a multiple of $n_2$, $n_3$, $n_4$, and $n_5$.
- Be careful if you use a root-finding method that internally using floating point representation. These numbers are too large to be represented exactly. You may need to use another algorithm (such as Newton's method) to find the root over the integers.

**(b)** The python package `rsa` uses `PKCS #1 v1.5`. Research this scheme as necessary and explain how this padding scheme fixes the broadcast attack.

**(c)** The current version of `PKCS` is 2.2. Explain one problem fixed by version 2.2 that exists in 1.5. (E.g explain an attack on version 1.5 with the level of detail that was used in lecture.)

**(d)** Speculate why the current python package uses 1.5 if the most up to date version is 2.2. Is this secure? Explain why or why not.