Technische Universität München
Lehrstuhl Informatik VIII
Prof. Dr.-Ing. Georg Carle
Prof. Dr.rer.-nat. Jorge Cuellar
Kilian Glas, M.Sc
Filip Rezabek, M.Sc
Tutors:
Lucas Aschenbach
Gopi Mehta
Yudhistira Wibowo

Applied Cryptography for Decentralized Systems (IN3300) Summer Term 2024
Exercise 1

1 Bonus Programming Assignment

This section provides an overview of the GitLab interaction and the individual tasks.

1.1 Interaction with GitLab

As this the first programming exercise, we also provide details regarding the setup which will be with you for the run of the lecture. Each team will be provided with a GitLab Group in which new project appear for each programming exercise. Of note, a single programming exercise might contain several tasks. The included sheet will provide you the instructions accompanying the corresponding project and tasks. Each project have configured CI/CD pipeline which will deploy the code to a dedicated environment on each commit. Please, try to catch basic compile errors before pushing. To avoid waiting for the results, we encourage you to work on the assignments in advance as closer to the deadline we expect higher load. Similarly, as stated in the Intro slides, shall you face any challenges, we expect to hear from you 2 days prior to the deadline.

The code will be tested against a variety of unit tests, where all the tests must pass for a successful completion of the assignment. To observe the outcome of the tests, navigate to `Build–>Jobs–>Job of a commit`. You are expected to only modify the requested files, otherwise the pipeline might fail. Similarly, the exercises are structured in such a way no additional libraries are needed unless stated differently. As mentioned, during the tutor hours and lectures, we do not provide support on the programming exercise. Nevertheless, we recommend using the lecture discussion forum shall you face any issues.

Task 1 Polynomial Interpolation

In modern day of cryptography, polynomial interpolation, a method of estimating values between data points using polynomials, plays a significant role in multiparty computation. This method is heavily used in secret sharing schemes, which allows parties to divide and distribute shares of a secret – such as a cryptographic key. By dividing the secret, no one can fully reconstruct it without having a minimum number of shares (threshold). This increase the complexity required by an attacker to fully obtain a control of the secret, since it has to compromise multiple instances.

As your (very) first programming exercise, we will write an interpolation method over a field $\mathbb{F}$ with Rust, the most beloved^{TM} programming language for cryptography. Your task is to implement the `interpolate` function in `exercise_1/src/lib.rs`. This function accepts two vectors `xs` and `ys`, which represent $x_i \in \mathbb{F}$ and $y_i \in \mathbb{F}$ for every index i. The function must return either an `OK(Polynomial)` or an error `Err(InterpolateError)`.

We do not want to limit your creativity for this exercise, since there are several algorithms to perform interpolation operation. However, we strongly recommend the most commonly used Lagrange interpolation method. As a refresher, a Lagrange interpolating polynomial $P(x)$ with degree n of a field $\mathbb{F}$ and points $x_1, x_2, \ldots, x_{n+1}, y_1, y_2, \ldots, y_{n+1} \in \mathbb{F}$ where $\nexists 0 \leq i, j \leq n+1, i \neq j : x_i = x_j$ is defined as follows:

$$P(x) = \sum_{i=1}^{n+1} y_i \cdot \left(\prod_{0 \leq j \leq n+1, i \neq j} \frac{x - x_j}{x_i - x_j} \right)$$

For the implementation, we already provided you with skeleton project, which includes the library `math_lib`. The library implements most of the required stuff you need, including field, polynomial, unsigned integer, and others. You can take a look at them inside the `math_lib` directory to understand how the library is built.

It is also important that you return error when the input are invalid. Returning error will prevent program that use your library from having unexpected behavior because they give the function illegal parameters. Try to take a look into the bottom of the file `math_lib/src/polynomial/mod.rs` and try to figure out cases that should be covered in your implementation.

Task 2 BLS Signatures

Background: Some elliptic curves, in addition to allowing for the construction of group representations, close to generic groups, also allow for the efficient computation of bilinear pairings. Pairings allow for quadratic polynomial equations to be tested on the group elements directly without requiring knowledge of the discrete logarithm of either element. This powerful property makes elliptic curves a versatile tool for cryptographic constructions. One such construction is the Boneh-Lynn-Shacham (BLS) signature scheme.
Let G_1, G_2, and G_T be cyclic groups of prime order p with generators g_1, g_2, and g_T, respectively. We define an efficiently computable pairing e:

$$e : G_1 \times G_2 \rightarrow G_T$$

that is

- **Bilinear**: $e(P^a \cdot Q^b, R) = e(P, R)^a \cdot e(Q, R)^b \quad \forall a, b \in \mathbb{Z} \quad P, Q \in G_1 \quad R \in G_2$
- **Non-degenerate**: $e(g_1, g_2) \neq 1$

There are two main ingredients required:

- A **bilinear pairing** $e : G_1 \times G_2 \rightarrow G_T$ which is efficiently computable and non-degenerate.
- A **hash function** $H_1 : \{0,1\}^* \rightarrow G_2$ acting as a random oracle.

The signature protocol consists of a **KeyGen**, **Sign**, and **Verify** function which are defined as follows:

1. **KeyGen**(): Generate public-private key-pair $\rightarrow (pk, sk)$
   Choose a random $\alpha \overset{\$}{\leftarrow} \mathbb{Z}_q$ and set
   $$h \leftarrow g_1^{\alpha}$$
   Output $pk := h$ and $sk := \alpha$.

2. **Sign**(sk, m): Generate signature for message $\rightarrow (\sigma)$
   Output
   $$\sigma \leftarrow H_1(m)^\alpha$$
   which is a single group element of G_1.
3. **Verify**(pk, m, σ): Check correctness of signature $\rightarrow$ (`bool`)
   Accept signature if
   $$e(g_1, \sigma) \overset{!}{=} e(pk, H_1(m))$$
   is satisfied.

BLS signatures are generally shorter than ECDSA signatures as they consist of only one group element from G_2. Unlike ECDSA, they are also deterministic, which means they do not additional randomness during creating a signature. Another favorable property is that they allow for aggregation of multiple signatures into a single one, which can be verified in a single pairing operation. For secure signature aggregation an additional primitive is required

- A multi-valued **hash function** $H_2 : G_2^n \rightarrow R^n$ where $R = \{2^k \mid k = 0, \ldots, 128\}$

Signature aggregation uses the same **KeyGen** and **Sign** functions as the standard BLS scheme and extends the protocol with an **Aggregate** and **VerifyAggregate** routine.

4. **Aggregate**$(\sigma_1, \ldots, \sigma_n, pk_1, \ldots, pk_n)$: Aggregate signatures into single signature $\rightarrow (\sigma)$
   Compute deterministic and non-predictable scalars
   $$(t_1, \ldots, t_n) \leftarrow H_2(g_1^{\alpha_1}, \ldots, g_1^{\alpha_n})$$
   and aggregate signatures to
   $$\sigma \leftarrow \sigma_1^{t_1} \cdots \sigma_n^{t_n}$$
   combining all signatures into σ.
5. **VerifyAggregate**$(\sigma, pk_1, \ldots, pk_n)$: Check correctness of aggregated signature $\rightarrow$ (`bool`)
   Accept the signature if
   $$e(g_1, \sigma) \overset{!}{=} e(g_1^{\alpha_1 t_1}, H_1(m_1)) \cdots e(g_1^{\alpha_n t_n}, H_1(m_n))$$
   which is the same as
   $$e(g_1, \sigma) \overset{!}{=} e(pk_1^{t_1}, H_1(m_1)) \cdots e(pk_n^{t_n}, H_1(m_n))$$
   In the case of $m := m_1 = \cdots = m_n$ this simplifies to
   $$e(g_1, \sigma) \overset{!}{=} e(pk_1^{t_1} \cdots pk_n^{t_n}, H_1(m))$$
   using the bilinearity of $e(\cdot, \cdot)$.

Task: The `bls_lib/` library implements the BLS signature protocol. However, the library differs from the standard BLS aggregation routine as described above significantly, which introduces a severe security flaw. This vulnerability allows an attacker, by carefully choosing their public key, to create a valid aggregate signature for their and another user's public key.

Your task is to complete the `attack(...)` function in `exercise_2/src/attack.rs` to exploit this vulnerability. The function should take any public key of a foreign user and an arbitrary message as input and return the public key and signature of the attacker. The signature should be a valid aggregate signature for the attacker's and the foreign user's public key.

Hint: Rogue Key Attack