Reduction from the DHI assumption to the DY05 VUF (18 points)

The DHI assumption

The q-Diffie-Hellman-inversion (DHI) assumption states the following: Let $\mathbb{G}$ be a group of order p with a generator g. Given the tuple $c := (g,g^\alpha,g^{\alpha^2},...,g^{\alpha^q})$ where α is drawn uniformly from $\mathbb{Z}_p$ it is hard to compute $g^{1/\alpha}$. More precisely, any PPT adversary $\mathcal{A}$ only has negligble probability to compute $g^{1/\alpha}$ given c. That is, $\mathsf{Pr}[\alpha \leftarrow \mathbb{Z}_p : \mathcal{A}(1^\lambda,c) = g^{1/\alpha}] \leq \mathsf{negl}(\lambda)$ where the group order p depends superpolynomially on the security parameter λ.

Verifiable unpredictable functions

A verifiable unpredictable function consists of three algorithms Gen, Eval and Vfy.

Key Generation

The key generation Gen on input a security parameter produces a public *verification key* vk and a secret *evaluation key* sk.

Evaluation

The evaluation Eval on input an evaluation key sk and a preimage $x \in X$ (where X is some input space) produces an image $y \in Y$ (an element of some image space Y) and a proof π. However, for this exercise we will not use the proof and hence omit it.

Verification

The verification Vfy on input a verification key vk, a preimage $x \in X$ and an image $y \in Y$ outputs 1 iff $\mathsf{Eval}(\mathsf{sk},x)=y$, and 0 otherwise. Moreover, the guarantee that Vfy only outputs 1 on the correct preimage/image pair hold even for any verfication key — not only for vks that have been produced by Gen.

Unpredictability of a VUF

We say a VUF is selectively unpredictable, iff any PPT adversary $\mathcal{A} = (\mathcal{A}_1,\mathcal{A}_2)$ (here both $\mathcal{A}_1$ and $\mathcal{A}_2$ are PPT) only has a negligble winning probability in the following game: The adversary $\mathcal{A}_1$ produces a challenge preimage $x^* \in X$ and sends it to the challenger $\mathcal{C}$. The challenger $\mathcal{C}$ generates a key pair $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$ and sends it to the adversary. Let $y^* \leftarrow \mathcal{A}_2^{\mathcal{O}(\cdot)}(1^\lambda,\mathcal{vk},x^*)$ *be the adversary's predicted image. Here the adversary has access to the evaluation oracle $\mathcal{O}$ which on input $x_i \neq x^*$* returns $y_i \leftarrow \mathsf{Eval}(\mathsf{sk},x_i)$ (for $x_i = x^*$ *it returns $\bot$). The adversary wins if $y^* =* $\mathsf{Eval}(\mathsf{sk},x^*)$.

Your task

You will implement a successful reduction from the Diffie-Hellman-inversion assumption to the selective unpredictability of the Dodis-Yampolskiy VUF. That is, you construct a reduction that takes a input a DHI challenge and provides its selective unpredictability adversary with a verification key and an evaluation oracle. Your reduction can then leverage the adversary's power to produce a solution for the DHI challenge. Specifically, you will implement a successful reduction, namely the class `DHI_DY05_Reduction`. In particular, its `run` method takes as input an `DHI_Challenger challenger` with which you can interact, e.g. get the challenger's challenge `challenger.getChallenge()`. Once you find the solution to the challenger's challenge you can simply return it in the `run` method. To help you solve the DHI challenge, you can run the unpredictability adversary `I_Selective_DY05_Adversary adversary` via its `run` method (which you can access via `this.adversary.run(this)` in the class `DHI_DY05_Reduction`). However, the adversary expects a (public) verification key and a correct oracle for evaluation queries, hence you have to implement the methods `getPK()` and `eval(preimage)`. To receive the adversary's challenge preimage (which you need to set up a useful verification key) the adversary will call the reduction's `receiveChallengePreimage(challenge_preimage)` method.
If you don't implement these methods in a formally correct way, the adversary may or may not work, but you may not get full points for the exercise.

SimplePolynomial Class

To help you with your task, we implemented for you a class `SimplePolynomial` whose instances represent univariate polynomials over $\mathbb{Z}_p$. The class offers the following methods:

```
package algebra;

public class SimplePolynomial implements Comparable<SimplePolynomial> {
    public final BigInteger modulus;
    public final int degree;

    public SimplePolynomial(BigInteger modulus);
    public SimplePolynomial(BigInteger modulus, int... coefficients);
    public SimplePolynomial(BigInteger modulus, BigInteger...
coefficients);

    public BigInteger get(int i);

    public SimplePolynomial add(SimplePolynomial other);
    public SimplePolynomial subtract(SimplePolynomial other);
    public SimplePolynomial multiply(SimplePolynomial other);
    public SimplePolynomial multiply(BigInteger number);

    public BigInteger lead();
    public SimplePolynomial div(SimplePolynomial divisor);
    public BigInteger eval(BigInteger input);
    public SimplePolynomial shift(int offset);
    public SimplePolynomial shift(BigInteger offset);

    public boolean isZero();
```

```
    public boolean isOne();
    public boolean isConstant();
}
```

Testing your implementation

The recommended technical setup uses VSCode and Docker.

- Install both on your system according to the instructions on the respective website.
- Download the container.zip file and unzip it.
- Open VSCode, press F1 and select `Extensions: Install Extensions...` and install the extension `Dev Containers` by Microsoft.
- Press F1 and select `Remote Containers: Open Folder in Container...` and select the unzipped folder `{{task}}`.
- Click "Yes, I trust the authors" (optionally also check "Trust the authors of all files on the parent folder 'build'").
- You can run/debug you solution by clicking `Run`/`Debug` directly above the `main` method of the respective `*TestRunner.java` file. (It might take a while until the java extension initialization is completed.)

Alternatively, you can also install a JDK/JRE 17 and run the code locally or on an online service like codio.

Scores and Points

If your implementation is correct, it should succeed in each test case and get full points.

Important Note: The tests which we run in Code Expert and on the TestRunner we gave you are only **preliminary**. After the submission deadline, we will run more exhaustive tests on your solution and review it manually.

Therefore, a solution which is only partially correct may receive full points in the preliminary tests but will get only partial points, eventually. Therefore, make sure that your reductions are correct in the formally theoretic sense of cryptographic reductions!

Time and Memory Restrictions

The resources the TestRunner can use to test your solution are limited. We expect your solution to use less than 1 minute of CPU time and a restriced space of memory when run several times.

Solutions which run into TimeOut- or OutOfMemoryExceptions will be rejected by us and receive 0 points.

Cheater Warning

The purpose of this task is to algorithmically reduce the DHI problem to the unpredictability of the DY05 VUF.

Any solution which tries to solve the DHI problem by cryptoanalytical algorithms or by "tricking" the testing environment is considered to be a cheating attempt and will receive zero points.