Reduction of CDH to a quadratic self-map (12 points)

In this task, your job is to show that you can solve the Computational Diffie-Hellman problem when given access to an oracle that computes a quadratic map on the exponents of group elements. For this end, you will finish the implementation of a generic reduction (`CDH_Quad_Reduction`) that can solve CDH-challenges when given access to an adversary that computes a quadratic map on the exponents of group elements. The quadratic map is fixed, but unknown to you: i.e., you are guaranteed that the adversary/oracle always computes the same function, however you don't get to know the function directly.

Below, we will explain various parts of this task and various objects, with which your implementation needs to work with.

Generic Group Model

In this task, you will work in the **Generic Group Model** (GGM). In the GGM, it is assumed that there is a cyclic group $\mathbb{G}$ of prime order p (where $p \in \Theta(2^\lambda)$) to which you only have *oracle access*. I.e., each element $g \in \mathbb{G}$ is given to you by a random binary string, a *handle*, which represents this element but does not leak any information about its exponent. Given two handles of group elements g, h in the GGM, you need to ask an oracle to receive a handle of the product $g \cdot h$ or of a power g^e for $e \in \mathbb{Z}$.

For this task, we have already implemented the GGM and will provide you with an interface that encapsulates all necessary functionalities:

```
public interface IGroupElement extends IBasicGroupElement<IGroupElement> {
    BigInteger getGroupOrder();
    IGroupElement multiply(IGroupElement otherElement);
    IGroupElement power(BigInteger exponent);
    IGroupElement invert();
    IGroupElement clone();
    boolean equals(IGroupElement otherElement);
}
```

Each group element that you are given in this task will be of the type `IGroupElement`. Use the methods of this interface to perform generic group operations.

Quadratic Selfmap

In this text, when we speak of a **quadratic selfmap** we mean a map of the type $$\mathbb{Z}_p \times \mathbb{Z}_p \longrightarrow \mathbb{Z}_p,$$ $$ (x,y) \longmapsto axy + bx +cy +d $$ for $a,b,c,d\in \mathbb{Z}_p$ where a is never zero.

However, in this exercise you are not given access to the above function. In fact, you are only given access to an oracle `adversary` that computes the above function on the exponents of group elements. I.e., in this exercise you have access to the function $$\mathbb{G} \times \mathbb{G} \times \mathbb{G} \longrightarrow \mathbb{G},$$ $$ (g, g^x,g^y) \longmapsto g^{axy + bx + cy + d}.$$ In particular, you don't know the values a,b,c,d. However, the adversary that computes the function $(g, g^x,g^y) \longmapsto g^{axy + bx + cy + d}$ for you is *consistent*, i.e., it will always use the same values a,b,c,d to compute this quadratic map.

Computational Diffie-Hellman Assumption

Let us now define the CDH assumption (as in the lecture): Let $g \in \mathbb{G}$ be a fixed generator of the cyclic group $\mathbb{G}$ of order p. Draw $x,y\gets \mathbb{Z}_p$ uniformly at random. A CDH challenge is of the form $$(g, g^x, g^y).$$

The solution to the above CDH challenge is the group element $$g^{xy},$$ which is uniquely determined by the challenge (g, g^x, g^y). I.e., the CDH problem consists in multiplying group elements *in the exponent*.

Formally, the associated CDH challenge game looks at follows:

1. The CDH-challenger $\mathcal{C}$ fixes a generator g of the group $\mathbb{G}$ and draws $x,y\gets \mathbb{Z}_p$ uniformly at random.
2. $\mathcal{C}$ starts a DDH-adversary $\mathcal{A}$ and hands him over the CDH-challenge (g, g^x, g^y) when $\mathcal{A}$ asks for it.
3. $\mathcal{A}$ performs some generic computations on (g, g^x, g^y) and comes up with a group element g^z that it sends back to $\mathcal{C}$.
4. $\mathcal{C}$ checks if $g^z = g^{xy}$. If $g^z = g^{xy}$ then $\mathcal{A}$ wins. Otherwise, $\mathcal{A}$ loses.

The **advantage** of $\mathcal{A}$ in this security game is defined by
$$Adv_{CDH}(\mathcal{A}) := \Pr[\mathcal{A} \text{ wins}] =\Pr_{x,y\gets \mathbb{Z}_p}[\mathcal{A}(g,g^x,g^y) = g^{xy}] .$$
The **CDH-assumption** states that for each ppt adversary $\mathcal{A}$ its advantage in winning the above is negligible (in the security parameter).

To encapsulate CDH challenges in this task, we created a container class, which you should use:

```
public class CDH_Challenge<IGroupElement> {
    public final IGroupElement generator;
    public final IGroupElement x;
    public final IGroupElement y;
}
```

The Reduction

Your job is to finish the implementation of the class `CDH_Quad_Reduction`:

```
public class CDH_Quad_Reduction extends A_CDH_Quad_Reduction<IGroupElement>
{

   @Override
   public IGroupElement run(I_CDH_Challenger<IGroupElement> challenger) {
       // Write Code here!
```

```
    }

    @Override
    public CDH_Challenge<IGroupElement> getChallenge() {
        // Write Code here!
    }
}
```

`CDH_Quad_Reduction` reduces the CDH problem to the problem of evaluating any quadratic selfmap on the exponents of group elements. I.e., your job is to solve CDH problems while given access to an oracle that computes a quadratic selfmap. Formally, your reduction needs to solve a `CDH_Challenge`. To get this challenge, you need to ask the `I_CDH_Challenger<IGroupElement> challenger` for a challenge by calling.

```
CDH_Challenge<IGroupElement> challenge = challenger.getChallenge();
```

When you come up with a solution `IGroupElement solution` you need to `return` it in the method `run`.

For solving `challenge`, the class `CDH_Quad_Reduction` has a field `I_Quadratic_Adversary<IGroupElement> adversary;` which contains an adversary that can compute a quadratic selfmap:

```
public interface I_Quadratic_Adversary<IGroupElement> extends
I_CDH_Adversary<IGroupElement> {
    public IGroupElement run(I_CDH_Challenger<IGroupElement> challenger);
}
```

When calling `adversary.run(this)`, `adversary` will try to ask for a CDH challenge from your reduction. For this end, the method `getChallenge()` will be called and will return the value of the field `CDH_Challenge<IGroupElement> cdh_challenge` of your reduction.

When `adversary` can succesfully ask for a `cdh_challenge` it will try to compute the quadratic function $aXY + bX +cY + d$ on its exponents. This means, if `cdh_challenge` contains the group elements (g,g^x, g^y), `adversary` will return the group element $g^{axy + bx +cy+ d}$ in its `adversary.run(this)` call.

Now, given `adversary`, your task is to solve `CDH_Challenge<IGroupElement> challenge`. `challenge` will consist of three group elements `generator, x,y` which can be written as g, g^x,g^y. You need to return a group element g^{z} s.t. $g^{z} = g^{xy}$ (i.e. $z = xy$).

BigInteger

In this task, you need to work with the class `java.math.BigInteger`, which we use to represent big numbers. The class `java.math.BigInteger` is immutable i.e. the values of given instances cannot be changed. When you apply mathematical operations, new instances of `java.math.BigInteger` are generated. Be aware of the following methods of `java.math.BigInteger`:

```java
package java.math;

public class BigInteger extends Number implements Comparable<BigInteger> {
    public static BigInteger valueOf(long val);

    public BigInteger add(BigInteger val);
    public BigInteger subtract(BigInteger val);

    public boolean testBit(int n);
    public int bitLength();
}
```

You will need some of those methods to solve this task.

Complexity

We expect your solution to have a feasible runtime. In fact, your solution should have a runtime complexity of $O(\log p)$ and should make $O(\log p)$ many calls of `adversary`'s run method.

Note, that after the submission deadline, we will test your solutions on our systems. To exclude infeasible brute-force solutions, our testing system will automatically zero points to solutions that need more than a few minutes for all testcases. If your solution for this task is correct, it should complete all testcases in under a minute when you test it.

Tips and Tricks

Try to solve this exercise step for step:

1. First, write a method `IGroupElement f1(IGroupElement g, IGroupElement gX, IGroupElement gY)` that on input g,g^x,g^y outputs the group element $g^{axy + bx +cy +d}$.
2. Then, write a method `IGroupElement f2(IGroupElement g, IGroupElement gX, IGroupElement gY)` that computes the function $(g,g^x,g^y)\mapsto g^{axy + bx +cy}$ using `f1`.
3. Then, write a method `IGroupElement f3(IGroupElement g, IGroupElement gX, IGroupElement gY)` that computes the function $(g,g^x,g^y)\mapsto g^{axy + bx}$.
4. Again, write a method `IGroupElement f4(IGroupElement g, IGroupElement gX, IGroupElement gY)` that computes the function $(g,g^x,g^y)\mapsto g^{axy}$.
5. You now have a function `f4` that can compute g^{axy} from the group elements g, g^x and g^y. Additionally, you can compute the element g^a by using `f4`. The tricky part is now to compute g^{xy} by only having access to the function $(g, g^x,g^y) \mapsto g^{axy}$. To solve this task, you need to fiddle around a bit. **Important:** Note that the order p of $\mathbb{G}$ is a prime number and that a is never zero. Therefore, we always have $$ a^{p-3} \mod p = \frac{1}{a^2}. $$
6. There is also an alternative solution to step 5, which requires only a constant number of calls of `f4`.

Constructors

Do **under no circumstances** change or remove the constructor of `CDH_Quad_Reduction` which we pre-implemented. The TestRunner needs this empty constructor to test your solution. If this constructor does not exist or work, then the TestRunner can not test your solution and you will receive 0 points.

Submission

Please submit your solution on CodeExpert in the corresponding task

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 DLog problem to the CDH problem.

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