Schnorr Signatures and the Forking Lemma (7 points for EUF-NMA and 10 points for EUF-CMA)

Schnorr Signatures

We briefly explain the Schnorr Signature Scheme. The scheme operates over a group $\mathbb{G}$ using a hash function H that maps from $\mathbb{G} \times {0,1}^*$ to $\mathbb{Z}_{p}$ where p is the prime order of $\mathbb{G}$. The secret key is $x \in \mathbb{Z}_p$ and the public key is $y = g^{x}$ where g is a publicly known generator of $\mathbb{G}$.

To generate a signature on a message m, using the secret key x, we do the following.

- choose $r \gets \mathbb{Z}_p$ uniformly at random
- set $R = g^r$
- compute the hash $c = H(m,R)$
- set $s = r + c\cdot x$
- the output signature is c, s

To verify, we check that $c = H(m, g^s \cdot y^{-c})$ and return true if this equation holds, false otherwise.

Existential Unforgeability (EUF)

In this task, we consider the existential unforgeability (under two forms of attacks) of the Schnorr scheme. In the existential unforgeability game, the adversary's goal is to provide a forged signature on a message of its choice. The different subtasks will consider adversaries of dufferebt capabilities. All of your adversaries in this task are going to output their solution as the return value of `adversary.run()` as an object of the following type:

```
public class Schnorr_Solution<BigInteger> {
    public String message;
    public SchnorrSignature<BigInteger> signature;
}
```

where `message` corresponds to m above and `c` and `s` correspond to c and s above. The value $H(m, R)$ is the return value of your implementation of `hash(R, message)` which is explained in the following.

The random oracle model (ROM)

In cryptographic proofs, hash functions are sometimes modelled by an idealized function called a "random oracle". Such a random oracle returns a truly random value for every new value submitted to it. As the adversary has to query the random oracle in order to compute hashes, security reductions in this ideal model may implement it in a way that helps them - we call this *programming* the random oracle.

In all of the tasks below, you are expected to implement a full-domain hash random oracle as the function `public BigInteger hash(String message, IGroupElement r)` where the return value needs to be between 0 and $p-1$ (p is the order of the group).

No message attacks (NMA)

In a no-message attack, the adversary will have a non-negigible advantage to produce a signature that is valid with respect to a public key and a hash function that your reduction needs to provide. For the NMA subtask, your reduction therefore needs to implement the methods `public Schnorr_PK<IGroupElement> getChallenge()` which should return a public key of the form

```
public class Schnorr_PK<IGroupElement> {
    /**
     * the public key as a group element
     */
    public IGroupElement key;
    /**
     * the group generator
     */
    public IGroupElement base;

}
```

Your reduction furthermore needs to implement the method `public BigInteger hash(String message, IGroupElement r)`. This method should behave like a random oracle, i.e. the outputs should be uniformly random, but at the same time consistent (i.e. if the adversary requests the same mesasge-group element pair twice, it should return the same value).

Interlude: Rewinding Adversaries

Sometimes, a reduction may need to rewind an adversary and re-run it with partially different information (e.g. partially different responses to oracle queries) to learn the information it needs. This is possible as we assume the reduction has access to the adversary as a computer program and can run it several times with different inputs. As adversaries are randomized, we assume here that the adversary has a special input from which it derives all random choices it makes.

You can set the seed of an adversary by using the method `adversary.reset(long seed)`. The adversary is then guaranteed to produce the same behaviour as long as it sees the same input from the reduction. Your reduction may also call the reset method before ever running the adversary so that it uses a known seed that it can be reset to later on.

Chosen-message attacks (CMA)

In the CMA subtask, in addition to seeing the public key, the adversary will make queries to the method `public SchnorrSignature<BigInteger> sign(String message)` which will need to return a valid Schnorr signature as specified above.

The Discrete Logarithm Problem (DLOG)

A common assumption in groups is the discrete logarithm assumption. Informally speaking, the discrete logarithm assumption states that it should be hard for any efficient adversary to compute the discrete logarithm y of a random group element $x = g^y$.

Your reductions will need to solve the discrete logarithm problem. They are given a challenge of the form:

```
public class DLog_Challenge<IGroupElement> {
    /**
     * A generator of the group. Usually an encoding of one.
     */
    public final IGroupElement generator;
    /**
     * A group element of the form g^x, where g is the generator in this
tuple.
     * Usually, g^x was drawn uniformly random from the group.
     */
    public final IGroupElement x;
}
```

The value $y \in \mathbb{Z}_p$ should be returned as a `BigInteger` as the return value of your reduction's `run` method. More specifically, it should hold for the return value `ret` of your reduction's `run` method that `generator.power(ret).equals(x)`. Your reduction can obtain the DLOG challenge through the method `challenger.getChallenge()`.

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 5 minutes 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 show EUF-NMA or EUF-CMA security of the Schnorr Signature scheme assuming DLog is hard.

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.