Information Security Lab - AS21

Reductions

Handout: Oct 22, 2021 12:00 AM

Due: Nov 8, 2021 3:00 PM

RSA-FDH EUF-CMA

Open Task

RSA-FDH signatures (8 + 8 + 9 = 25 points)

RSA-FDH signatures

We briefly explain the RSA-FDH Signature system. The algorithm **keygen** samples two large primes p and q. It sets the RSA modulus $N = p \cdot q$. We note that $\Phi(N) = (p-1) \cdot (q-1)$ where $\Phi(N)$ is Euler's Φ-function. **Gen** samples $e \leftarrow \mathbb{Z}_{\Phi(N)}$ such that $gcd(e, \Phi(N)) = 1$ and sets $d = e^{-1} \mod \Phi(N)$. The public verification key is **vk** $= (N, e)$ and the private signing key is **sk** $= (N, d)$.

The RSA-based full-domain-hash (RSA-FDH) digital signature scheme works as follows: Assume we have a hash function H whose image space is $\mathbb{Z}_N$, i.e. the full domain (hence the name). The algorithm **Sig** takes as input a secret key **sk** $= (N, d)$ and a message m. It then computes the hash of the message raised to d, i.e. $H(m)^d \mod N$.

The algorithm **Ver** takes as input a public key **vk** $= (N, e)$, a message m, and a signature σ and checks that the verification equation $H(m) = \sigma^e \mod N$ holds. It outputs 1 if it holds, 0 otherwise.

The RSA Assumption

The RSA assumption states that for $N = p \cdot q$ (where p and q are primes), given e (with $gcd(e, \Phi(N)) = 1$) and y it is hard to compute x such that $x^e = y \mod N$, i.e. it is difficult for an adversary to compute e-th roots.

In all of the following subtasks you are supposed to solve an RSA Challenge which looks as follows:

```java
public class RSA_Challenge {
    public final RSA_Modulus modulus;
    public final BigInteger y;
    public final BigInteger e;

    public RSA_Challenge(RSA_Modulus modulus, BigInteger y, BigInteger e) {
        this.modulus = modulus;
        this.y = y;
        this.e = e;
    }
}
```

You need to implement the method `public BigInteger run(I_RSA_Challenger challenger)` of the reduction in each subtask. Using the challenger `challenger`, you can obtain your challenge with a call to `challenger.getChallenge()`. The actual RSA modulus N can be obtained by making a call to `modulus.getN()`. Your task is, using the adversary provided to you in each subtask, to solve the challenge and output an RSA solution x as a BigInteger such that $x^e = y \mod N$.

Existential Unforgeability (EUF)

In this task, we consider the existential unforgeability (under various forms of attacks) of the RSA-FDH 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 various 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 RSAFDH_Solution {
    public String message;
    public BigInteger signature;
}
```

where `message` corresponds to m above and `signature` corresponds to σ above. The value $H(m)$ is the return value your implementation of `hash(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)` where the return value needs to be between 0 and N (N is the RSA modulus which is a part of the public key you output to the adversary). We now turn to the specific adversary's expectations.

Where to get randomness

Throughout this exercise, you may need to generate random values. To do this, you are expected to use the field `rnd` (see `private SecureRandom rnd = new SecureRandom()` at the top of each reduction). You may also use `NumberUtils.getRandomBigInteger(Random RNG, BigInteger max)` where you can set `RNG` to `rnd` to generate objects of type `BigInteger` that encode a number between 0 and `max-1`.

Your task

The reduction

Your task is to produce a **reduction** that uses a provided adversary for the existential unforgeability game to win the RSA game. To this end you must implement the methods of the `RSAFDH_EUFCMA_Reduction` (resp. `RSAFDH_EUFnaCMA_Reduction` and `RSAFDH_EUFNMA_Reduction`) class:

```java
public class RSAFDH_EUFCMA_Reduction extends A_RSAFDH_EUFCMA_Reduction{
    private SecureRandom rnd = new SecureRandom();
    // your code here
    public BigInteger run(I_RSA_Challenger challenger) {
        //your code here
    }
    public BigInteger hash(String message) {
        //your code here
    }
    public BigInteger sign(String message) {
        //your code here
    }
    public RSAFDH_PK getPk() {
        //your code here
    }
}
```

In each of the following subtasks you are provided with an adversary which is in the field `adversary` of your reduction class (you can expect this field to be initialized). You may ask your adversary how many distinct hash queries it plans to make via `adversary.numHashQueries()`. At the end of its run (which you need to start by calling `adversary.run(this)`), the adversary outputs a signature of the type `RSAFDH_Solution`. The reduction is in turn expected to output an object of the type `BigInteger` which encodes a value x such that $x^e = y \mod N$ (with `y` and `e` as in your RSA challenge, and N as in `modulus.getN()`).

Further remarks

- We note that the parameter sizes of the RSA keys used in this framework are much smaller than what is secure in practice. This is due to time constraints in CodeExpert, i.e. we choose small parameters so that the key generation does not exceed the CodeExpert time limits. Do not use keys this small for any real-world application!
- Despite the small parameter sizes your task is to write a *reduction* and not to attempt to break the RSA assumption directly. The reduction should use the adversary provided to you by us. Solutions which break the RSA assumption without using the adversary are outside of the scope of this task and may not get points.
- In order to get full points in this task you are expected to make at most one call to `adversary.run(this)`. Your success probability may depend on the amount of messages signed or the amount of hash queries made by the adversary, i.e. it does not need to be tight. Solutions that make multiple calls to the adversary will be awarded at most half the possible points.
- Do not create additional constructors for your reduction classes and do not delete any existing constructors as this may lead to the test runner being unable to test your solution. This may result in 0 points for the task.
- The points visible in CodeExpert before the submission deadline are **preliminary**. We may run additional tests after the deadline and will publish your final score then.

Warm-up: no message attacks (NMA) (8 points)

In this subtask you are expected to implement a reduction that solves the RSA problem using an adversary that makes a so-called no-message attack. That is, the adversary you are provided with is able to produce a signature without ever asking the reduction for signatures. Concretely, you need to provide the following implementations:

- the method `public BigInteger run(I_RSA_Challenger challenger)` which is the "main" method of your reduction. It will be called by the testing framework. In this method you need to call `adversary.run(this)` to run your adversary. The return value of this method is your solution to the RSA challenge you can get from your challenger.
- the method `public BigInteger hash(String message)` which provides the random oracle functionality as above. It will be called by the adversary.
- the method `public RSAFDH_PK getPk()` which the adversary will call after you have started its run with `adversary.run()` to obtain an RSA public key

Non-Adaptive Chosen Message Attacks (naCMA) (8 points)

In this subtask, your adversary will ask you to sign some messages which it has chosen before it sees the public key. You need to provide signatures to these messages in order for the adversary to succeed in providing you with a forgery. To do this, you need to implement the following methods:

- the method `public BigInteger run(I_RSA_Challenger challenger)` whch is the "main" method of your reduction. It will be called by the testing framework. In this method you need to call `adversary.run(this)` to run your adversary. The return value of this method is your solution to the RSA challenge you can get from your challenger.
- the method `public BigInteger hash(String message)` which provides the random oracle functionality as above. It will be called by the adversary.
- the method `public RSAFDH_PK submitMessagesAndGetPK(List<String> messages, String challengeMessage)` with which the adversary will submit the messages, i.e. `messages` is a list of messages that the adversary wants a signature to, and `challengeMessage` is the message the adversary intends to forge a signature for. This method needs to return the public key to the adversary.
- the method `public Map<String, BigInteger> getSignatures()` which will be called after the adversary has submitted his messages. You need to return a map of all the previously requested messages and corresponding signatures which need to be valid with respect to your public key as well as the hash oracle you implement.

Adaptive Chosen Message Attacks (CMA) (9 points)

In this subtask, your adversary will be fully adaptive. It will ask you to sign messages after it has seen the public key. You need to provide signatures to these messages in order for the adversary to succeed in providing you with a forgery. To do this, you need to implement the following methods:

- the method `public BigInteger run(I_RSA_Challenger challenger)` whch is the "main" method of your reduction. It will be called by the testing framework. In this method you need to call `adversary.run(this)` to run your adversary. The return value of this method is your solution to the RSA challenge you can get from your challenger.
- the method `public RSAFDH_PK getPk()` which the adversary will call to obtain a public key.
- the method `public BigInteger hash(String message)` which provides the random oracle functionality as above. It will be called by the adversary.
- the method `public BigInteger sign(String message)` which needs to return a signature on the message `message` that is valid with respect to your public key as well as the hash oracle provided by you.