ASSIGNMENT #4 - DEADLINE 15.11. 23:59 (Brno time). UP TO 10 POINTS AWARDED.

This assignment focuses on a real-world timing side-channel attack on an implementation of the ECDSA (Elliptic Curve Digital Signature Algorithm) cryptosystem in Python. The provided implementation is leaking sensitive data via the duration of the `sign` function. The tasks will focus on analyzing this leakage and extracting the private key from provided signatures via a lattice attack. **Even though this assignment involves seemingly complex topics of timing side-channels, ECDSA and lattice attacks, at its core it is a simple data analysis assignment.** The assignment contains some background on ECDSA and timing attacks, and consists of 2 parts with a total of 5 tasks. Information on expected solution format and grading is at the end. It also contains hints to help you along.

➰ ECDSA primer

ECDSA is a digital signature algorithm very similar to DSA that works over elliptic curves. See https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm for some more information, the below is just a short recap of how it works. Another good resource is the "Guide to Elliptic Curve Cryptography" book by Hankerson, Menezes and Vanstone, https://www.springer.com/gp/book/9780387952734.

Let's have an elliptic curve defined over a prime field $\mathbb{F}_p$, with p a large prime. The short Weierstrass equation for the curve is given by:

$$y^2 \equiv x^3 + ax + b$$

where the coefficients a and b are from $\mathbb{F}_p$. Any pair of coordinates (x, y) from $\mathbb{F}_p \times \mathbb{F}_p$ which satisfy the above equation is a point on the curve. These points, together with the point at infinity form an abelian group, i.e., you can add them together to produce another point and you can also negate them to produce another point. Given a point G on the curve, you can perform scalar multiplication by a scalar (any integer) r where you add the point G to itself r-times. This is denoted as $[r]G$.

To perform ECDSA, one needs to fix the domain parameters (the curve and a specific point on it). In the following exercise, the `secp256r1` curve is used. Then the private key d is a random integer from $[1, n-1]$ where n is the order of the generator point G. The public key is the point $P = [d]G$. The notation used in this primer is used throughout the assignment.

Signature generation

1. When signing a message m, the signer first hashes the message producing a hash h using some hash function H, in our case SHA1 is used. If the length of the digest in bits is larger than the bit-length of n, the leftmost L_n bits of the digest are used, where L_n is the bit-length of n (see line 101 of `ecdsa.py`).
2. The signer then picks a random nonce k from the interval $[1, n-1]$ (the same interval as the private key, it can be thought of as a one-time private key and is sensitive). See line 103 of `ecdsa.py`.
3. The signer then performs scalar multiplication ($[k]G$) of the generator G with the nonce k, takes the x-coordinate of the result, reduces it modulo n and stores it as r (

$= ([k]G)_x \mod n$), the first component of the resulting signature. See line 105 of `ecdsa.py`.
4. The signer then computes $s \equiv k^{-1}(h + rd) \mod n$, which constitutes the second component of the signature. See line 107 of `ecdsa.py`.
5. The signature is the tuple (r, s).

Signature verification

Signature verification is not important for this exercise.

⌚ Timing attacks

Timing attacks are a class of side-channel attacks which leverage the duration of some operation in a implementation of a cryptosystem (or a part of it) to obtain information about a secret value used in this operation. An implementation is said to be *leaking information via timing* if there is any dependency of its runtime on some secret value used within it. The leakage itself might or might not be exploitable, it could for example be very small, such that it is thwarted by environmental noise for any realistic attacker.

A classical example of a leaking implementation is the early-abort string compare algorithm (in this case password compare) displayed below. The loop aborts early in case of a wrong character, if the difference in duration resulting from one loop iteration is observable to an attacker (not necessarily using a single run, possibly after applying some statistical methods on many runs), the leakage is exploitable and the password can be extracted iteratively.

```
function checkPasswordVarTime(password) {
    let correctPassword = "hunter2";
    for (let i of correctPassword) {
        if (i >= password.length || password[i] !== correctPassword[i]) {
            return false;
        }
    }
    return password.length === correctPassword.length;
}
```

Timing attacks on cryptosystem implementations were/are quite common, examples include several cryptosystems in the work of Kocher in Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems , or RSA in Remote Timing Attacks are Practical or ECDSA in Remote Timing Attacks are Still Practical. Attacks such as Meltdown or Spectre, or generally cache attacks, can be thought of as timing attacks as well, as they generally use the duration of some operation involving the cache to infer information about the state of the cache.

📂 Contents

All of the Python scripts provided require Python 3. You will find the following files in a ZIP file named after your UCO in the study materials.

- **<UCO>_data.csv**: The signature data, ECDSA signatures with measured signing time. This uses the key in `<UCO>_meta.csv`. In the format (all numbers in base 10): `<time>,<signature component r>,<signature component s>`
- **<UCO>_meta.csv**: The public key used and the message signed in `<UCO>_data.csv`. There is only one line as there was only one keypair and one message signed while generating the

data, the signatures differ as a property of the signature algorithm. In the format (all numbers in base 10):
`<x coordinate of public key>,<y coordinate of public key>,<message in ASCII>`
- **<UCO>_leak.csv**: The simulated leakage data. In the format (all numbers in base 10):
`<time>,<nonce k>,<private key d>`
- **mod.py**: Basic Python class for modular arithmetic, used throughout the implementation.
- **ecdsa.py**: Code of the vulnerable implementation.
- **solver.py**: Code of the lattice attack.
- **setup.sh**: Setup script for the libraries necessary for the lattice attack (works on the aisa server). Run this in an empty directory to create a Python virtual environment with the required libraries.
- **README.{md,pdf}**: This README file.

1️⃣ Part 1

In part 1 you will be working with the ECDSA implementation (`ecdsa.py`) and the simulated leakage data (`<UCO>_leak.csv`) by performing simple data analysis.

Goal: Identify the source and nature of the timing leakage in the ECDSA implementation in `ecdsa.py`.

Note: The supporting code for finite-field arithmetic in `mod.py` might also be variable time and leaking, however, that is not the target of this exercise and you do not need to look at it.

Hint: Use a heatmap to plot the value of a selected byte of some sensitive value used while signing against the resulting duration.

Hint: You can use and run the `ecdsa.py` implementation to help with all of the tasks in this part. You might measure the duration of signing and alter sensitive values for example.

- **(Task 1.a)** Identify the innermost line of code in `ecdsa.py` on which sensitive data are leaking via time when performing ECDSA signing (the `sign` function is called). There is only one line that should be identified as leaking. The number of this line of code should be put into the file `<UCO>_line.txt`. The line numbering of the file is 1-based, meaning that the first line (`#!/usr/bin/env python3`) has number 1 and for example the `scalarmult` function is defined on line 47. **[1 point]**
- **(Task 1.b)** What is the sensitive information that is leaking via timing, describe it via 10 words or less (e.g. "the ... of ..."). Write your answer into the file `<UCO>_what.txt`. **[2 points]**
- **(Task 1.c)** The file `<UCO>_leak.csv` contains simulated leakage data, leaking the same way as the `ecdsa.py` implementation, the first column `time` is the duration of signing of a message using the specified `nonce` k and `private key` d. Your task is to write a one-line Python function that takes as input the nonce and private key and **returns the expected time of signing**. Your aim is basically to compute the leakage (the first column of `<UCO>_leak.csv`) using the sensitive inputs (the other two columns of `<UCO>_leak.csv`). The leakage is noisy, so your function will not return the exact values in the first column. The goal is to minimize the sum of squares of differences between your computed values and the given ones, as well as to express the mathematical relationship between the sensitive value and time, thus your function should respect your answer to **(Task 1.b)**. The function **must** have two inputs of type `int`, the `nonce` and the `private_key`, **must** have the name `leak` and **should** only have one line (e.g. `return ...`). There is no experience with Python necessary, there are no packages necessary, for more information, look at https://docs.pytho

n.org/3/library/stdtypes.html for information on the `int` type and its methods. Put the function into the file `<UCO>_simulate.py`. **[3 points]**

```
def leak(nonce: int, private: int):
    return ...
# Given a line:
127094,189079309736078836929994652155386787625,2757136873503566990886964480249810
14467
# the call is
leak(189079309736078836929994652155386787625,275713687350356699088696448024981014
467)
# and it should return a value close to 127094.
```

2️⃣ Part 2

In part 2 you will be working with the provided leakage data (`<UCO>_data.csv`, `<UCO>_meta.csv`) and the lattice attack code (`solver.py`).

Goal: Perform the lattice attack and get the private key.

Note: The lattice attack requires the *fpylll* library which has a somewhat involved installation process. There are several options available for you to run the code:

- Setup a local Python3 virtual environment and install fpylll, which requires the *fplll* library as well as the GMP/MPIR and MPFR libraries.
- Use CoCalc and its support for SageMath (which includes the required *fpylll* library) to run the attack. To do so, register on CoCalc, create a new project, upload the assignment files to it, then create a new "Linux Terminal" file, open it and when you need to run the lattice attack code in `solver.py`, simply do `sage --python solver.py <args>`.
- Use the aisa server. This assignment includes a setup script `setup.sh`, which, when run on the aisa server of the Faculty of Informatics (aisa.fi.muni.cz) sets-up a Python virtual environment that you can use to run the attack code. After you setup the environment it is necessary to activate it. The instructions to do so are printed during setup, also look at the bottom of the `setup.sh` script. If everything is setup correctly, running `python3 solver.py` should result in a message about required arguments and not an import or some other error. If you decide to use the aisa server, please respect its computation guidelines.

Running the attack **locally or on CoCalc is preferred**.

Hint: Think from the attacker's point of view, when they know the signature generation process took some time and produced a certain signature, what information do they have on the sensitive values used in the signature? When do they have the most information? The heatmaps from the first part might be useful as well. You might think of it as a question of conditional entropy, i.e., given a signature with a certain time, what is the entropy of the sensitive values used in it?

- **(Task 2.a)** The file `<UCO>_data.csv` contains signatures produced with a leaking implementation (using the public key and message from the `<UCO>_meta.csv` file), you can think of it as the implementation in `ecdsa.py`. Your task is to choose 100 signatures from the provided signatures for which the amount of leaked (known to the attacker) information is the largest. This definition is intentionally vague, think about how the leakage looks like and when is the largest amount of sensitive information known. These signatures should be sorted by the amount of information leaked (most information known first) and put into the file `<UCO>_signatures.txt` in the same format as they are in `<UCO>_data.csv` (so just select and sort certain lines of that file into your solution file). **[2 points]**

- **(Task 2.b)** Use the lattice attack code in `solver.py` to recover the private key corresponding to the public key in `<UCO>_meta.csv` that was used to produce the signatures in `<UCO>_data.csv`. The expected input to the solver is the `<UCO>_meta.csv` file as the `-m/--meta-file` argument and the signatures with the largest amount of sensitive information known `<UCO>_signatures.txt`, e.g. those computed in the previous task as the `-s/--signatures` argument. If the signatures are chosen correctly, or at least somehow contain enough information to recover the private key, the script will output it. Put the private key into the file `<UCO>_key.txt` . As you have the public key, you can check that the computed private key is correct by using the scalar multiplication function. **[2 points]**

📄 Output

All of your results should be put into a ZIP file with the name `<UCO>.zip`, where `<UCO>` stands for your "Univerzitní Číslo Osoby". The contents of the zip file should be as follows:

- `<UCO>_line.txt`
- `<UCO>_what.txt`
- `<UCO>_simulate.py`
- `<UCO>_signatures.txt`
- `<UCO>_key.txt`

If you do not want to submit a solution for a task, do not include the corresponding results file in the ZIP.

💯 Grading

- **(Task 1.a)** The contents of the `<UCO>_line.txt` file will be compared to the one true solution, 1 point will be awarded for a match.
- **(Task 1.b)** The full 2 points will be awarded to an answer that correctly states what *property* of what *value* is leaking. A point will be awarded to an answer that gets one of those right.
- **(Task 1.c)** The full 3 points will be awarded to a solution that both (1.5 points each):
    - minimizes the *sum of squares of differences* $S = \sum_i (\text{time}_i - \texttt{leak}(\text{nonce}_i, \text{privatekey}_i))^2$ between the given time values and the leak values computed using the function in the answer, such that the sum S is lower than 5 000 000 000 (five billion),
    - computes the time according to the actual leakage mechanism, meaning that it computes the leakage using the correct property of the correct value.
- **(Task 2.a)** There is a list of 100 correct signatures with the largest amount of leaked information. Your solution will be compared to this list and the number of signatures from your answer in this list will determine the amount of points according to the table below.

Number of correctly chosen signatures	Points
100	2
75-99	1.5
50-74	1

Number of correctly chosen signatures	Points
25-49	0.5
0-24	0

- **(Task 2.b)** The contents of the `<UCO>_key.txt` file will be compared to the precomputed private key, 2 points will be awarded for a match.

💬 Miscellaneous

The assignments are personalized, you each have a different private key and input, so sharing answers is detrimental. Cheating is penalized. If you have any questions regarding the assignment, please use the discussion forum.