Module 04: Elliptic Curve Cryptography and Lattice Cryptanalysis

Week-11: ECDSA^2 and Timing Side Channels (20 Points)

Jan Gilcher, Kenny Paterson, and Kien Tuong Truong
jan.gilcher@inf.ethz.ch, kenny.paterson@inf.ethz.ch, kitruong@ethz.ch

November 2023

Hi all! Welcome to the last module for the semester!

This module's lab session will work slightly differently from the previous modules. Instead of splitting up students and TAs over several rooms, we will have one big lab session every week with all the TAs for this modules present. At the beginning of each lab session we will give a short presentation with helpful information. In this first lab for example, that will include how to setup the SageMath environment you'll be using to solve the labs.

The room for the lab sessions is: **ML D 28**.

This lab will also follow a Capture The Flag (CTF) approach, that you might already know from a previous module. For this you will be required to interact with our CTFd instance, where you will find the code and be able to submit flags. You have already received the credentials to for your account on the server via email (if you have not message us ASAP!). The server is available at:

`https://isl.aclabs.ethz.ch`

As with previous modules, you are also welcome to post your questions and to join ongoing discussions on the Moodle forum at the following link:

`https://moodle-app2.let.ethz.ch/mod/forum/view.php?id=911956`

1 Overview

This lab is focused on implementing a custom variant of the Elliptic Curve Digital Signature Algorithm (ECDSA) [1], which we call ECDSA^2, or ECDSA squared.

Please note that **we expect you to use Python 3.11** and the corresponding Python 3.11 bindings of SageMath for this lab.

2 Getting Started

First you will have to install SageMath, Python 3 and the required Python packages. For this we recommend to follow the conda-forge install instruction from the SageMath documentation,[1] which we will summarize here.

Note: on Windows you will have to use WSL. For performance reasons, WSL2 is recommended (which is the default on newer systems).

[1] `https://doc.SageMath.org/html/en/installation/conda.html`.

1. If you do not have a working conda installation (e.g. anaconda) install mambaforge as follows:

```
curl -L -O \
https://github.com/conda-forge/miniforge/releases/latest/download/Mambaforge-$(uname)-$(uname -m).sh
sh Mambaforge-$(uname)-$(uname -m).sh
```

2. If you do already have a working conda make sure you are using conda-forge as default and install mamba:

```
conda config --add channels conda-forge
conda config --set channel_priority strict
conda install mamba
```

3. Create a new conda environment with SageMath and Python 3.11 and the required Python packages[2] as follows:

```
mamba create -n isl_sage sage python=3.11 pycryptodome numpy pandas
```

4. Activate your sage environment :

```
conda activate isl_sage
```

Whenever you want to work on this lab you will have to activate the sage environment via as in step 4 above.

To start solving this lab's tasks, go to the CTF website[3]. In the challenges tab you will find the following tasks for this week under lab01.

1. (warmup) Implement ECDSA^2.
2. Break vulnerable implementations of ECDSA^2.
3. Analyze the timing distribution of a non-constant-time Schnorr signature server and detect small nonces.

For each subtask, you will be able to download the server code. This allows you to test and debug locally without internet access. Simply run

```
python3 server.py
```

to start a server in the current terminal.

3 Elliptic Curve Signature Schemes

3.1 ECDSA^2

ECDSA^2 adds our own little spin to ECDSA signatures. Key generation remains unchanged from ECDSA, however signing and signature verification are modified.

The signing algorithm changes to the following:

$\underline{\mathsf{ECDSA}^2\text{-Sign}(x, m) \mapsto (r, s)}$

$h \leftarrow \mathsf{bits2int}(H(m)) \mod q$

do

$k \leftarrow\!\!\$\ \{1, \ldots, q-1\}$

$r \leftarrow \mathsf{x\text{-}coord}([k]P) \mod q$

$s \leftarrow k^{-1}(h^2 + 1337 \cdot x \cdot r) \mod q$

until $r \neq 0$ and $s \neq 0$

return (r, s)

[2] Only *PyCryptodome* is required but we permit usage of *pandas* and *numpy* and therefore include them here.
[3] `https://isl.aclabs.ethz.ch`

3.2 Schnorr

The Schnorr digital signature scheme is similar to ECDSA, sharing the same key generation algorithm. In this module we will use a slightly altered version, namely a version with deterministic nonces, described below:

Schnorr-Sign$(x, m) \mapsto (h, r)$

$k \leftarrow \mathsf{HMAC\_DRBG}(x, m)$
$r \leftarrow \mathsf{x\text{-}coord}([k]P) \mod q$
$h \leftarrow \mathsf{bits2int}(H(m, r)) \mod q$
$s \leftarrow k - hx \mod q$
return (h, s)

Schnorr-Verify$(Q, m, (h, s)) \mapsto \{0, 1\}$

if $1 \leq r \leq q - 1$ and $1 \leq s \leq q - 1$
  $r' \leftarrow \mathsf{x\text{-}coord}([s]P + [h]Q)$
  $h' \leftarrow \mathsf{bits2int}(H(m, r')) \mod q$
  if $h = h'$
    return 1
return 0

4 Implementing ECDSA^2 (5 Points)

Your first task is implementing ECDSA^2 signature verification and signing. In Section 3.1 we give you the pseudocode for the signing algorithm. Your task is to implement it, infer the verification algorithm and implement it as well.

For this we will give you `ecdsa2.py` skeleton, which already contains the implementations for all the elliptic curve arithmetic and ECDSA^2 key generation. Your task is to implement the following methods of the class ECDSA^2:

- `Verify`.
- `Sign` and `SignFixedNonce`.

Please refer to the header comments of the functions for the concrete API.

You can connect to the challenge server at `isl.aclabs.ethz.ch:40102` and `isl.aclabs.ethz.ch:40103` respectively.

Since for all challenges you have the server code, you can create a local server instance that you can use for debugging. For obvious reasons, we cannot provide you with the correct implementation of `ecdsa2.py`, so the server will automatically import yours if you place it in the same folder where the `server.py` file is located.

In order to test your implementation, you will have to (1) be able to create correct ECDSA^2 signatures (2) be able to distinguish correct ECDSA^2 signatures from incorrect ones. We refer to the server code for further details.

Your hand-in for this task is the finished `ecdsa2.py` as well as two files `lab1m0_2.py` and `lab1m0_3.py` that successfully obtain and print the flags from the server. At the end each should print the flag to standard output (this should be on the last line of the output of your program).

5 ECDSA^2 Cryptanalysis: Part 1 (5 Points)

ECDSA signatures can be vulnerable if nonces are handled incorrectly. These issues are inherited by ECDSA^2 as well.

For this task you have to:

- Find a way to forge a signature for the message "*gimme the flag*".

- Submit the flag on the CTF server.

Your hand-in for this task is a single Python file, `lab1m1.py`, that successfully interacts with our server and receives the flag. At the end it should print the flag to standard output (this should be on the last line of the output of your program). You can connect to the challenge server at `isl.aclabs.ethz.ch:40110`.

We will grade your submission based on the number of signature queries and the achieved success rate of your attack. For full points we expect you to achieve 100% success rate.

6 ECDSA^2 Cryptanalysis: Part 2 (5 Points)

For this task we provide you with a server that is again vulnerable in some way. You will have to:

- Find a way to forge a signature for the message "*gimme the flag*".
- Submit the flag on the CTF server.

Your hand-in for this task is a single Python file, `lab1m2.py`, that successfully interacts with our server and receives the flag. At the end it should print the flag to standard output (this should be on the last line of the output of your program). You can connect to the challenge server at `isl.aclabs.ethz.ch:40120`.

We will grade your submission based on the number of signature queries and the achieved success rate of your attack. For full points we expect you to achieve 100% success rate.

7 Timing Attack on Deterministic Schnorr (5 Points)

This task is in preparation for next weeks lab, where you will build upon this further to create a successful end-to-end attack. When implementing cryptographic operations, we have to take care that we do not accidentally leak any information to an observer. A very common kind of this kind leakage is the timing of our operations, specifically when it depends on our secrets. In the lecture you have seen the double-and-add algorithm, which is famously vulnerable to timing attacks, as you only perform the addition when the current bit is one. This leads to shorter runtime for messages with lots of zeros.

In this task, you will interact with a server that has timing side-channel reminiscent of a real side-channel issue that was discovered in OpenSSL.[4] While care has been taken to ensure the double-and-add always performs the same number of operations independent of the current bit value, it still leaks the number of leading zero bits. The server we give will, upon request, give you a signature on a random message as well as the message and the time it took to generate the signature. Your task is:

- Understand where the timing side-channel in the scalar multiplication is coming from.
- Collect enough signatures and identify 20 signatures where the nonce has at least 6 leading zeros.
- Sent 20 *different* messages to the server, which will lead to nonces with at least 6 leading zeros.
- Submit the flag on the CTF Server

Your hand-in for this task is a single Python file, `lab1m3.py`, that successfully interacts with our server and receives the flag. At the end it should print the flag to standard output (this should be on the last line of the output of your program). You can connect to the challenge server at `isl.aclabs.ethz.ch:40130`.

Note: The server will disconnect after sending 20000 signatures using the same key. Try to be conservative with the amount of signature queries your attack needs, while still performing a reliable attack. We will grade

[4] `https://link.springer.com/chapter/10.1007/978-3-642-23822-2_20`

your submission based on the number of signature queries and the achieved success rate of your attack. For full points we expect you to achieve 95% success rate. Your attack should also reliable work independent of the absolute timings, i.e. **do not hardcode any timing information you observe from the server or locally**. Your solution should learn how the server you are currently interacting with behaves. For this you might want to analyze the distribution of leading zeros among generated nonces (*hint: assume the nonce generation is random*). Then, use this information to detect nonces with leading zeros.

Using SageMath also introduces several other non-constant-time operations, e.g. division and modular reductions. These are not part of the attack, though they could be fatal in a real-world deployment. As timing measurements are very susceptible to noise, we recommend to close all other programs when developing your attack locally. You can also artificially increase the the time difference, by increasing the time required for a single step of the double-and-add implementation (via inserting sleep statements). You can rest assured that, we have taken the necessary steps to ensure that the timing information provided by the server has an amount of noise that does not inhibit a successful attack.

When testing locally, if you still observe noise large enough to prevent your attack despite the above, you should disable or limit frequency scaling ("Turbo-Boost") on your CPU. The best way to do this is via your computer's BIOS, but many laptops unfortunately do not have a BIOS setting for this. We therefore recommend the following:

On Linux Install *cpufrequtils*. This allows you to set the maximum and minimum CPU frequency. DO NOT set this to your maximum Turbo-Boost frequency unless you know you have sufficient cooling (unlikely on a laptop). Instead DO SET THIS to your *base-clock frequency*, this is often specified as part of the Model Name field in the output of *lscpu*. If not use the model name to look it up in the processor specification database from Intel[5] or AMD.[6] Setting the frequency this way will NOT persist a system reboot.

On Windows Frequency scaling will be mostly deactivated when you switch to power saving mode. If you have a modern Intel processor with both E- and P-Cores this will not disable frequency scaling for the E-cores. This is unlikely to cause issues as the server should run on the P-cores. If you still encounter issues try setting the task affinity of the server in windows task manager such that it prefers P-Cores. If this does not help, there is a set of registry edits that reveals configuration options to completely disable frequency scaling, via the energy management settings. We can assist you with this if you ask in person during a lab session.

On Mac Disabling frequency scaling on Macs requires power saving mode. To enforce power saving mode unplug the power cable from your mac.

8 Hand-in and Grading

Your hand-in will consist of a single zip file, containing three folders, named `week1`, `week2`, and `week3`. Each contains your solution for the corresponding week. This weeks files will go into the `week1` folder. This folder should contain the following six files: `ecdsa2.py`, `lab1m0_2.py`, `lab1m0_3.py`, `lab1m1.py`, `lab1m2.py`, and `lab1m3.py`.

Your final folder structure for the whole submission should look like this:

```
submission.zip
|-- week1
|   |-- ecdsa2.py
|   |-- lab1m0_2.py
|   |-- lab1m0_3.py
|   |-- lab1m1.py
```

[5] `https://ark.intel.com/content/www/us/en/ark.html`
[6] `https://www.amd.com/en/products/specifications/processors`

```
|   |-- lab1m2.py
|   |-- lab1m3.py
|-- week2
|   |-- lab2m0.py
|   |-- lab2m1.py
|   |-- lab2m2.py
|-- week3
    |-- lab3m0.py
    |-- lab3m1.py
    |-- lab3m2.py
```

For each task you will get points as noted in the task description. You'll get 50% of the points when you successfully enter the flag on the CTF server. The remaining 50% will be awarded based on running your submissions repeatedly on our servers.

9 References

1. *Public Key Cryptography For The Financial Services Industry: Agreement Of Symmetric Keys Using Discrete Logarithm Cryptography.* ANSI X9.42-2003 (2003).
   `https://webstore.ansi.org/standards/ascx9/ansix9422003r2013`

2. *Elliptic Curve Cryptography Subject Public Key Information.* Turner *et al.* (2009).
   `https://www.ietf.org/rfc/rfc5480.txt`