Module 04: Elliptic Curve Cryptography and Lattice Cryptanalysis

Week-12: Cryptanalysis of Schnorr (40 Points)

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

December 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

In the lecture you have seen the theory behind lattice attacks on ECDSA. In this lab you will apply that knowledge to attack the Schnorr digital signature scheme.

In the first two tasks you will implement the necessary transformation from HNP to SVP for the Schnorr and in the third task you will combine last week's timing analysis with the previous tasks to implement an end-to-end attack.

> **Attention**: We ask you to implement your solutions for this module by using LLL as the basic building block. Even though SageMath ships with a CVP solver (in the `fpylll` package), you are not allowed to use it. We will be checking for it and **you will be penalized** if your solution directly imports anything from the `fpylll` package.
>
> When you need to use LLL, build a matrix `M` and call `M.LLL()` on it. This will internally use `fpylll` and it is the only case in which its usage is allowed.

As a reminder below is is the signing and verification algorithms for Schnorr:

```
Schnorr-Sign$(x, m) \mapsto (h, r)$
$k \leftarrow$ HMAC_DRBG$(x, m)$
$r \leftarrow$ x-coord$([k]P) \mod q$
$h \leftarrow$ 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 $1 \leq s \leq q - 1$
  $r' \leftarrow$ x-coord$([s]P + [h]Q)$
  $h' \leftarrow$ bits2int$(H(m, r')) \mod q$
  if $h = h'$
    return 1
return 0
```

2 Schnorr Cryptanalysis: Partially Known Nonces (40 Points)

Before you can start implementing the lattice attack on Schnorr, you first have to translate it to a shortest vector problem similar to what you have seen in the lecture. We suggest the following steps:

1. From the Schnorr signing equation derive the corresponding hidden number problem, i.e. an equation of the form $t_i x = u_i + e_i \mod q$
2. Encode the hidden number problem as a closest vector problem.
3. Use Kannan's Embedding as seen in the lecture and exercise to convert the CVP to SVP.
4. Solve the resulting SVP using LLL.

2.1 Large Nonce Leakage (12 Points)

This server implements a Schnorr signing oracle. Upon a signature request for a message it will serve you the signature as well as the 128 most significant bits. After 5 signing request it will not accept any further signing request for the same key, and will disconnect if you request an additional signature.

Your task is to obtain the flag by successfully recovering the key using the lattice techniques from the lecture and submit a valid signature for the message *gimme the flag*.

Your hand-in for this task is a single python file, `lab2m0.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:50200`.

2.2 Small Nonce Leakage (12 Points)

This tasks works exactly like the previous one, except the server only leaks 8 bits. To compensate you will be allowed to make a maximum of 60 signing queries.

Your task is to obtain the flag by successfully recovering the key using the lattice techniques from the lecture and submit a valid signature for the message *gimme the flag*.

Your hand-in for this task is a single python file, `lab2m1.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:50210`.

2.3 End-to-End Attack (16 Points)

In this last task for this lab, you will combine the timing analysis you did last week with the lattice attack from today. Like last week, we give you access to a Schnorr server with the same timing side-channel during signature creation. Upon request it will again provide you with a message, its signature, and the time required to create the signature. As in previous tasks, the server will release its flag if you are able to generate a valid signature for the message *gimme the flag*.

Your hand-in for this task is a single python file, `lab2m2.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:50220`.

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 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[1] or AMD.[2] 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.

3 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 `week2` folder. This folder should contain the following three files: `lab2m0.py`, `lab2m1.py`, and `lab2m2.py`.

[1] `https://ark.intel.com/content/www/us/en/ark.html`

[2] `https://www.amd.com/en/products/specifications/processors`

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
|   |-- 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.

4 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. *Factoring polynomials with rational coefficients.* Lenstra, A.K., & Lenstra Jr., H. W., & Lovsz, L. (1982). Mathematische Annalen, vol. 261, pp. 515-534.
   `https://infoscience.epfl.ch/record/164484/files/nscan4.PDF`

3. *Lattice basis reduction: improved practical algorithms and solving subset sum problems.* Schnorr, C.P., Euchner, M. (1994). Math. Programming 66, 181–199.
   `https://link.springer.com/article/10.1007/BF01581144`

4. *Improved algorithms for integer programming and related lattice problems.* Kannan, R. (1983). STOC 1983, pp. 193–206.
   `https://dl.acm.org/doi/10.1145/800061.808749`

5. *Efficient signature generation by smart cards.* Schnorr, C.P (1991). J. Cryptology 4, pp. 161–174.
   `https://doi.org/10.1007/BF00196725`