Module 01: Elliptic Curve Cryptography

Week-03: Cryptanalysis of ECDSA

Benjamin Dowling, Jan Gilcher, Kenny Paterson, and Sikhar Patranabis

benjamin.dowling@inf.ethz.ch, jan.gilcher@inf.ethz.ch
kenny.paterson@inf.ethz.ch, sikhar.patranabis@inf.ethz.ch

September 2020

Hi all! Welcome to the first Information Security Lab for this semester. This lab is part of the Module-01 on *Elliptic Curve Cryptography*, or ECC in short.

Instructions for Online Lab

We begin with the usual pointers and instructions. As you are aware, the lab sessions will be held online via Zoom. We ask that all students join the Zoom session via the following link:

`https://ethz.zoom.us/j/91450896871`

The first 30-ish minutes of the Zoom session will consist of a general presentation and discussion about the lab objectives and evaluation criteria. Afterwards, you may choose to stay on in the lab session to discuss specific problems you might be having with the lab. 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/course/view.php?id=13172`

During the Zoom session, we will be hosting "break-out rooms". These are meant to facilitate one-on-one discussions with one of the TAs. During such discussions, you can share your screen and bring to our attention specific implementation issues that you might be facing. Please be aware that when you are sharing a screen with the session, you are responsible for the content that is presented to any other students in the breakout. In other words, *please be mature*.

Overview

This lab is focused on implementing cryptanalysis of ECDSA [1] based on *nonce-leakages*. In other words, you will be implementing attack algorithms that recover the secret ECDSA signing key based on either complete or partial information about the nonce(s) used in the ECDSA signing algorithm. We will look at three flavors of cryptanalytic attacks, namely *known-nonce attacks*, *repeated-nonce attacks* and *partially-known-nonce attacks*. The first two attacks are relatively straightforward and serve as warm-ups, while the third one is significantly more involved

and will require the use of lattice based techniques.

You have already seen the theoretical descriptions of these attacks during the lectures and in the exercise classes; but we will recap the necessary material in this lab sheet to help with the implementation. Please note that we expect you to use Python-3 for this lab. We will not accept submissions/solutions coded in any other language, including older versions of Python. *So please make sure that your submissions are Python-3 compatible.*

`fpylll`

This lab requires the use of solvers for lattice problems such as the closest vector problem (CVP) and the shortest vector problem (SVP). Fortunately, we can make black-box use of publicly available implementations of these (and many other) lattice algorithms from `fpylll` - a Python library for performing lattice reduction on lattices over the integers, available at:

`https://github.com/fplll/fpylll`

`fpylll` is based on its C++ predecessor `fplll` [1] and provides an easy-to-use Python-based interface for several state-of-the-art algorithms on lattices that rely on floating-point computations. This includes implementations of the reduction algorithms such as LLL [2] and BKZ [3], as well as implementations of solvers for CVP and SVP, all of which will be relevant and extremely useful for this lab. Hence, we expect you to familiarize yourself with `fpylll`.

[1] `https://github.com/fplll/fplll`

Note that installing `fpylll` is a time-consuming process. It requires prior installation of several other open-source libraries for arbitrary precision integer and floating-point arithmetic, as well as libraries for interfacing between Python and C++. While you are welcome to try and install `fpylll` on your own workstations, we are also providing a virtual machine (VM) image that has `fpylll` pre-installed and ready to use.

The link to access and download the VM is available on Moodle. Below we provide you with a simple set of instructions on how to use this VM, and how to run `fpylll` on it:

- Boot the VM using your favorite virtualization software. For example, you can download and use the Oracle VM Virtualbox from the url below:

  `https://www.virtualbox.org/`

- Login using the root password `isl` and navigate to the folder titled `fpylll-master` on the desktop.

- Launch a terminal and execute the following command:

  ```
  source ./activate
  ```

- This will launch a virtual environment named `fpylll-env`. This is an isolated Python build environment with all dependencies for `fpylll` pre-installed and all environment variables appropriately set up.
- At this point, you can run any Python script(s) that invoke in-built functions from `fpylll`.

Note that when automatically evaluating your code, we will use this same VM and the same build environment. So using the VM would additionally allow you to pre-test your submission in an environment resembling our automated testing environment.

Using CoCalc

In case you are facing issues with installing the VM and running `fpylll` on it, you can alternatively test your code online using CoCalc. Please see the relevant instructions below:

- Go to `https://cocalc.com`.
- Click on "Run CoCalc Now" (or sign in with an existing account).
- **(Optional)** Create an account if you want to save your work on the CoCalc servers.
- Once the page has loaded, click on "Files" (upper left).
- On the upper right click "Upload" and select the files containing the input test vectors that are provided to you.
- On the upper left click on "+ New" and then click on "Jupyter Notebook". Under "Suggested Kernels", choose "SageMath". The Jupyter notebook should be created and will automatically open.
- Copy and paste your code into the notebook.
- Select the cell containing your code and hit `shift+enter`.
- Your code will now run (as indicated by filled a green circle). This might take up to 10 minutes (maybe more depending on the amount of load on the Cocalc servers).
- Please note that CoCalc may interrupt and stop your code if you are idle for too long. So make sure to keep an eye on it!
- Once your code executes successfully, click on "Files", select the newly created output files and then click on "Download".
- Compare your output files with the output test vectors provided to you.

Getting Started

You are provided with a skeleton file named `module_1_ECDSA_Cryptanalysis_Skel.py`. This file has several unimplemented functions related to cryptanalysis of ECDSA that you are expected to implement. Since this is the second lab, you are expected to implement any low-level functions that you wish to use on your own. The skeleton file defines a few such functions but does not implement them:

- `egcd`: Computes the gcd of two integers using the Euclidean algorithm.
- `mod_inv`: Computes $a^{-1} \mod p$ for input integers a and p.

ECDSA Cryptanalysis: Single Known Nonce

We now describe the first warm-up task for this lab – implementing ECDSA cryptanalysis based on a single known nonce. The skeleton file has an (incomplete) implementation of `recover_x_known_nonce` – a function that is meant to realize this attack. You are expected implement this function. It takes as input:

- k: the nonce used by the ECDSA signing algorithm (represented as a big integer)
- h: the message on which the signature was produced, post-hashing and mapping to an integer modulo q (represented as a big integer)
- (r, s): the signature produced by the ECDSA signing algorithm (represented as a pair of big integers)
- q: the order of the base point P on the elliptic curve used by the ECDSA scheme (represented as a big integer)

and should output the recovered signing key x (represented as a big integer).

Note: This attack is quite straightforward to implement and involves simple arithmetic modulo q. For the relevant theory for this attack, look at the lecture slides.

ECDSA Cryptanalysis: Repeated Nonces

The second warm-up task for this lab involves implementing ECDSA cryptanalysis based on a pair of signatures on *distinct* messages sharing a *common* (but *unknown*) nonce. In particular, the skeleton file has an (incomplete) implementation of `recover_x_repeated_nonce` – a function that is meant to realize this attack. You are expected implement this function. It takes as input:

- (h_1, h_2): the messages on which the signatures were produced, post-hashing and mapping to integers modulo q (represented as a pair of big integers)
- (r_1, s_1): the signature produced by the ECDSA signing algorithm on the first message (represented as a pair of big integers)
- (r_2, s_2): the signature produced by the ECDSA signing algorithm on the second message (represented as a pair of big integers)
- q: the order of the base point P on the elliptic curve used by the ECDSA scheme (represented as a big integer)

and should output the recovered signing key x (represented as a big integer).

Note: Again, this attack is quite straightforward to implement and involves simple linear algebra modulo q. The relevant theory for this attack can be found in the lecture slides.

ECDSA Cryptanalysis: Partially Known Nonces

We now arrive at the main task for this lab – implementing ECDSA cryptanalysis based on partially known nonces using lattice algorithms. For the ease of implementation, we have broken up the attack into a sequence of modular tasks. For each task, the skeleton file describes an incomplete function that you are expected to complete.

A Single HNP Equation

The first sub-task is to implement a function that builds a single equation as part of a larger overall *hidden number problem* (HNP) instance. The equation is to be built from a single ECDSA signature and some partial information about the corresponding nonce used by the ECDSA signing algorithm.

More concretely, assume that you are provided with the L most significant bits of an N-bit nonce used to generate a signature (r, s) on a message h (post-hashing and mapping to an integer modulo q – this is the representation of a message we will use in the rest of the discussion). Recall from the lecture notes that:

$$s = k^{-1} \cdot (h + x \cdot r) \mod q,$$

which after some simple re-arrangement yields:

$$(r \cdot s^{-1}) \cdot x = k - h \cdot s^{-1} \mod q.$$

Setting $t = (r \cdot s^{-1}) \mod q$ and $z = (h \cdot s^{-1}) \mod q$, we have:

$$tx = k - z \mod q.$$

Now, let a be an integer represented by the L most significant bits of k. Then we can write:

$$k = a \cdot 2^{N-L} + 2^{N-L-1} + e,$$

where e is some *error* term bounded as:

$$0 \leq |e| \leq 2^{N-L-1}.$$

Setting $u = a \cdot 2^{N-L} + 2^{N-L-1} - z$, we have:

$$tx = u + e \mod q$$

Here t and u are known, x is the target unknown and e is small but unknown. We can think of computing (t, u) as setting up a single equation as part of a larger overall HNP instance.

The skeleton file has an (incomplete) implementation of `setup_hnp_single_sample` – a function that is meant to realize this sub-task. You are expected to implement this function. It takes as input:

- N: the total number of bits in the nonce k;
- L: the number of known most significant bits of the nonce k;
- $(k_1, \ldots, k_L)$: a Python list object containing the L most significant bits of the nonce k (k_1 being the most significant bit);
- h: the message on which the signature was produced, post-hashing and mapping to an integer modulo q;
- (r, s): the signature produced by the ECDSA signing algorithm;
- q: the order of the base point P on the elliptic curve used by the ECDSA scheme;

and should output the pair (t, u) (to be represented as a pair of big integers).

For ease and modularity of implementation, you can implement and use the subroutine `MSB_to_Padded_Int`, which takes as input N, L and $(k_1, \ldots, k_L)$ as described above, and outputs $a \cdot 2^{N-L} + 2^{N-L-1}$.

We have provided you with some test vectors to test that your implementation of the function `setup_hnp_single_sample` is consistent with our expectation (e.g., with respect to the manner in which the bits of k are ordered in the input). The parameters for the test vectors are $(N, L) = (256, 128)$. They are in the file `testvectors_hnp_single_sample_256_128.txt`.

While we do not separately evaluate your implementation of this function, its correctness is crucial for the overall correctness of your attack implementation; so we strongly encourage you to test this function against the test vectors provided to you.

Building the Overall HNP Instance

The second sub-task is to simply extend the aforementioned approach to build a larger HNP instance consisting of several equations, built from several ECDSA signatures and partial leakages from the corresponding nonces. The skeleton file has an (incomplete) implementation of `setup_hnp_all_samples` – a function that is meant to realize this sub-task. You are expected implement this function. It takes as input:

- N: the total number of bits in each nonce k_i;
- L: the number of known most significant bits of each nonce k_i;
- n: the number of equations to be built (called n in the lecture);
- $\{(k_{i,1}, \ldots, k_{i,L})\}_{i\in[n]}$: a Python nested list object containing L most significant bits of each nonce k_i;
- $\{h_i\}_{i\in[n]}$: a Python list object containing each message h_i post-hashing and mapping to an integer modulo q;
- $\{(r_i, s_i)\}_{i\in[n]}$: a pair of Python list objects containing each signature produced by the ECDSA signing algorithm;
- q: the order of the base point P on the elliptic curve used by the ECDSA scheme;

and should output $\{(t_i, u_i)\}_{i\in[n]}$ (to be represented as a pair of Python list objects). Obviously, you should iterate over each signature and use the `setup_hnp_single_sample` function you implemented earlier to compute the corresponding (t_i, u_i) pair.

We have also provided you with some test vectors to test that your implementation of the function `setup_hnp_all_samples` is consistent with our expectation. The parameters for the test vectors are $(N, L, n) = (256, 128, 5)$. They are in the file `testvectors_hnp_all_samples_256_128_5.txt`.

Again, while we do not separately evaluate your implementation of this function, its correctness is crucial for the overall correctness of your attack implementation; so we strongly encourage you to test this function against the test vectors provided to you.

HNP to CVP

The next step is to transform the HNP instance into an instance of the closest vector problem (CVP). Recall from the lecture notes that given $\left(\{(t_i, u_i)\}_{i\in[n]}, q\right)$,

one can construct the following CVP basis matrix:

$$B_{\text{CVP}} = \begin{bmatrix} q & 0 & 0 & \dots & 0 & 0 \\ 0 & q & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & q & 0 \\ t_1 & t_2 & t_3 & \dots & t_n & 1/2^{L+1} \end{bmatrix},$$

and the following CVP target vector:

$$\underline{u}_{\text{CVP}} = \begin{bmatrix} u_1 & u_2 & u_3 & \dots & u_n & 0 \end{bmatrix}.$$

As was formally established during the lecture, for appropriate choices of parameters N, L, n and q, given B_{CVP} and $\underline{u}_{\text{CVP}}$, a CVP solver should produce a vector

$$\underline{v} = \begin{bmatrix} v_1 & v_2 & v_3 & \dots & v_n & v_{n+1} \end{bmatrix}.$$

such that v_{n+1} is of the form:

$$v_{(n+1)} = x/2^{L+1}.$$

Question 1: Suppose that the CVP output vector contains $(x - q)/2^{L+1}$ instead of $x/2^{L+1}$. Could this occur?

Hint: Recall that our CVP instance $(B_{\text{CVP}}, \underline{u}_{\text{CVP}})$ looks like the following:

$$B_{\text{CVP}} = \begin{bmatrix} q & 0 & 0 & \dots & 0 & 0 \\ 0 & q & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & q & 0 \\ t_1 & t_2 & t_3 & \dots & t_n & 1/2^{L+1} \end{bmatrix}, \quad \underline{u}_{\text{CVP}} = \begin{bmatrix} u_1 & u_2 & u_3 & \dots & u_n & 0 \end{bmatrix},$$

As already discussed, we expect that given this instance, the CVP solver should output a vector $\underline{v}$ in the lattice generated by B_{CVP} that is "close" to $\underline{u}_{\text{CVP}}$ and is of the form:

$$\underline{v} = \begin{bmatrix} v_1 & v_2 & v_3 & \dots & v_n & x/2^{L+1} \end{bmatrix}.$$

Note that we must have:

$$\underline{v} = \begin{bmatrix} a_1 & a_2 & a_3 & \dots & a_n & x \end{bmatrix} B_{\text{CVP}}.$$

for some $a_1, \dots, a_n \in \mathbb{Z}$ (in fact $a_i = -\ell_i$ where the ℓ_i satisfy $t_i x = u_i + e_i + \ell_i q$ as in the lecture). Now, consider the following modified vector

$$\underline{v}' = \begin{bmatrix} (t_1 + a_1) & (t_2 + a_2) & (t_3 + a_3) & \dots & (t_n + a_n) & (x - q) \end{bmatrix} B_{\text{CVP}}$$

Quite evidently, $\underline{v}'$ is also a vector in the lattice generated by B_{CVP}. Additionally, convince yourself that the following is true:

$$\underline{v}' = \begin{bmatrix} v_1 & v_2 & v_3 & \dots & v_n & (x - q)/2^{L+1} \end{bmatrix}.$$

Given this, do you think that the following condition could hold?

$$||\underline{v}' - \underline{u}_{\text{CVP}}|| < ||\underline{v} - \underline{u}_{\text{CVP}}||.$$

If yes, the CVP algorithm would output $\underline{v}'$ instead of $\underline{v}$, and your implementation would recover $(x - q)$ instead of q.

Question 2: The basis matrix described above has a non-integral entry. Figure out if the `fpylll` in-built CVP solver supports basis matrices and/or target vectors with non-integral entries.

Question 3: Suppose that the `fpylll` in-built CVP solver *does not* support basis matrices and/or target vectors with non-integral entries. How would you transform the aforementioned CVP instance into one where the basis matrix has only integral entries?

Hint: Think about scaling both the basis matrix and the target vector with an appropriate scalar.

To summarize, in this sub-task, you are expected to implement the incomplete function `hnp_to_cvp` that takes as input:

- N: the total number of bits in each nonce k_i;
- L: the number of known most significant bits of each nonce k_i;
- n: the number of equations in the HNP instance;
- $\{(t_i, u_i)\}_{i \in [n]}$: a pair of Python list objects representing the HNP instance;
- q: the order of the base point P on the elliptic curve used by the ECDSA scheme;

and that outputs $(B_{\text{CVP}}, \underline{u}_{\text{CVP}})$, where B_{CVP} is the CVP basis matrix (to be represented as a Python nested list object) and $\underline{u}_{\text{CVP}}$ is the CVP target vector (to be represented as a Python list object). Both of these should be scaled appropriately to contain only integral entries (if required).

Solving CVP using `fpylll`

The next natural sub-task is to implement a function that solves the (potentially scaled) CVP instance using the `fpylll` in-built CVP solver. So go ahead and implement the function `solve_cvp` in the skeleton file. This function takes as input $(B_{\text{CVP}}, \underline{u}_{\text{CVP}})$ in the format as described above and outputs the solution vector $\underline{v}$ (represented as a Python list object).

Question 4: Do you want to perform some pre-processing on the basis matrix B_{CVP} before passing it as input to the `fpylll` in-built CVP solver? Does the `fpylll` documentation say something about this?

Hint: You might want to pre-process the basis matrix B_{CVP} by reducing it using some lattice reduction algorithms. In particular, the `fpylll` library has in-built functions implementing lattice reduction algorithms such as LLL and BKZ.

Question 5: If you do not immediately find a satisfactory answer to Question 3, here is a more directed question. Would pre-processing B_{CVP} using a lattice reduction algorithm help? If yes, why? Think about what in-built reduction algorithms in the `fpylll` library you could use here.

CVP to SVP via Kannan's Embedding

Alternatively, suppose that you do not have direct access to a CVP solver, but you have access to a solver for the shortest vector problem (SVP). As shown in the lecture, one can further convert the CVP instance into an SVP instance, and then solve the resulting SVP instance. In particular, given the (potentially scaled) CVP basis B_{CVP} and the CVP target vector $\underline{u}_{\text{CVP}}$, once can use Kannan's embedding [4] to prepare the following lattice basis matrix:

$$B_{\text{SVP}} = \begin{bmatrix} B_{\text{CVP}} & \mathbf{0} \\ \underline{u}_{\text{CVP}} & M \end{bmatrix},$$

where $\mathbf{0}$ is an all-zero column vector of appropriate dimension and M is a specially chosen constant.

As was formally established during the lecture, for an appropriate choice of M, the lattice generated by B_{SVP} as designed above should contain a "short" vector of the form

$$\underline{f}' = \begin{bmatrix} \underline{f} & M \end{bmatrix},$$

where $\underline{f} = \underline{u}_{\text{CVP}} - \underline{v}$, and $\underline{u}_{\text{CVP}}$ is the target CVP vector, while $\underline{v}$ is the target CVP solution. Also (hopefully!), the SVP solver should produce the vector $\underline{f}'$ as the shortest vector given B_{SVP} as input. Note that given $\underline{f}'$, one can extract immediately recover $\underline{f}$, and hence the CVP solution $\underline{v}$.

Question 6: From what you have seen in the lecture, what should your choice of M be assuming that B_{CVP} and $\underline{u}_{\text{CVP}}$ are *not* scaled?

Hint: Think about the maximum value that M could take, and how that relates to the maximum distance between the CVP target and solution vectors.

Question 7: Now suppose that B_{CVP} and and $\underline{u}_{\text{CVP}}$ *are* scaled. How should you scale M to preserve SVP correctness?

To summarize, in this sub-task, you are expected to implement the incomplete function `cvp_to_svp` that takes as input a CVP instance $(B_{\text{CVP}}, \underline{u}_{\text{CVP}})$ and outputs the Kannan-embedded basis B_{SVP} (to be represented as a Python nested list object).

Note: If the CVP instance $(B_{\text{CVP}}, \underline{u}_{\text{CVP}})$ is scaled appropriately to contain only integral entries, then B_{SVP} also contains only integral entries.

Solving SVP using `fpylll`

Again, the next natural sub-task is to implement a function that solves the SVP instance using the `fpylll` in-built SVP solver. So go ahead and implement the function `solve_svp` in the skeleton file. This function takes as input B_{SVP} in the format as described above and should output the solution vector $\underline{f}'$ (represented as a Python list object) as described above.

Question 8: Would pre-processing B_{SVP} using a lattice reduction algorithm help? Or is this redundant when solving SVP? What does the `fpylll` in-built SVP solver actually do?

Question 9: For the specially structured basis matrix B_{SVP} prepared using Kannan's embedding, is the desired solution vector $\underline{f}' = \begin{bmatrix} \underline{f} & M \end{bmatrix}$ always the shortest vector in the lattice generated by B_{SVP}? Could the lattice contain a vector shorter than $\underline{f}'$? If yes, what is this vector?

Hint: Revisit the exercises for this week, recalling that the lattice generated by B_{SVP} contains the following vector:

$$\begin{bmatrix} 0 & 0 & 0 & \dots & q/2^{L+1} & 0 \end{bmatrix},$$

or an appropriately scaled version of the same. Is this vector shorter than the desired solution vector $\underline{f}' = \begin{bmatrix} \underline{f} & M \end{bmatrix}$?

Question 10: Suppose that after some experimentation, you managed to convince yourself that in practice, with overwhelmingly large probability, the desired solution vector $\underline{f}'$ is not the shortest vector but the *second* shortest vector in the lattice generated by B_{SVP}. In this case, does the `fpylll` in-built SVP solver allow you to recover the second shortest vector instead of the shortest vector?

Putting Everything Together

Finally, you are expected to put everything together by implementing the partially completed functions `recover_x_partial_nonce_CVP` and `recover_x_partial_nonce_SVP`. These functions are expected to implement partial nonce-based ECDSA cryptanalysis by directly solving CVP and solving CVP via SVP, respectively. Both these functions take as input the following:

- N: the total number of bits in each nonce k_i;
- L: the number of known most significant bits of each nonce k_i;
- n: the number of equations to be built;

- $\{(k_{i,1}, \ldots, k_{i,L})\}_{i \in [n]}$: a Python nested list object containing L most significant bits of each nonce k_i;
- $\{h_i\}_{i \in [n]}$: a Python list object containing each message h_i post-hashing and mapping to an integer modulo q;
- $\{(r_i, s_i)\}_{i \in [n]}$: a pair of Python list objects containing each signature produced by the ECDSA signing algorithm;
- q: the order of the base point P on the elliptic curve used by the ECDSA scheme;

and should output the recovered signing key x (represented as a big integer). The functions should use the sub-routines that you have already implemented before. Some of this is already done for you in the skeleton file. The missing piece is how to finally extract the signing key x, which you should implement, thereby completing the functions.

Recall from the lecture note that solving the CVP instance should output a vector $\underline{v}$ of length $(n+1)$ such that the last entry of $\underline{v}$ is $x/2^{L+1}$.

Question 11: Recall that you might have already scaled your CVP basis and target vector in order to use the `fpylll` built-in CVP solvers. In this case, what would you expect the solution to the CVP instance to look like?

Question 12: Does solving the CVP instance via SVP using Kannan embedding require you to do some extra work in recovering x? Do the issues referred to in Questions 1 and 2 above also need to be taken into account in this case?

Testing and Evaluation

We have provided in the skeleton file six test modules – all of which involve file reads and writes. You can use these modules to test your implementation. The test modules are summarized as follows (in each case the base point order q is a 256-bit prime):

1. The first module tests the correctness of the `recover_x_known_nonce` function over 100 different randomly generated signing keys, nonces and signatures.
2. The second module tests the correctness of `recover_x_repeated_nonce` function over 100 different randomly generated signing keys, nonces and signature pairs sharing the same nonce.
3. Each of the remaining modules test the correctness of two functions, namely the `recover_x_partial_nonce_CVP` and `recover_x_partial_nonce_SVP` functions over 100 different randomly generated signing keys, nonces and signatures. For each signing key, the test vectors are generated using the following parameters:

(a) $(N, L, n) = (256, 128, 5)$

(b) $(N, L, n) = (256, 32, 10)$

(c) $(N, L, n) = (256, 16, 20)$

(d) $(N, L, n) = (256, 8, 60)$

In other words, the four test modules offer increasingly smaller amounts of leakage on the nonce (smaller L) but compensate by providing you with more instances per signing key (larger n).

For each of these test modules, you are provided with the input test vectors and the corresponding output vectors in separate input and output files. When you execute each test module, you will generate an output file. Your implementation is correct if the contents of this file *exactly* match the contents of the output file provided to you.

Note: You are free to test your code using your own custom-designed test modules. However, we will be using the *same* test modules as in the skeleton file to evaluate your submissions under an automated evaluation framework. Your final submission should have these test modules as is. *Please do not tamper with the test modules in any way to avoid interfering with our automated evaluation frameworks*.

Evaluation

When we evaluate your submissions, we will run the exact same test modules, albeit with respect to our own privately generated input and output files, which will not be made public prior to evaluation. This is why we ask you to leave the test module codes as is in your final submissions. *Failure to adhere to this instruction will incur heavy penalties*.

Summary of Evaluation Criteria. To summarize, you will be evaluated based on the correctness of your implementation of the individual functions:

1. `recover_x_known_nonce` (5 points)
2. `recover_x_repeated_nonce` (5 points)
3. `recover_x_partial_nonce_CVP` (7.5 points for each of the four test modules)
4. `recover_x_partial_nonce_SVP` (7.5 points for each of the four test modules)

So a total of 70 points is available for this week's lab.

Submission Format

Your completed submission for this week should consist of a *single* Python file, and should be named "`module_1_ECDSA_Cryptanalysis.py`".

You are expected to upload your submission to Moodle. The submissions for weeks 2 and 3 should be bundled into a single archive file named "module_1_submission_[insert LegiNo].zip". Submission instructions for the solution to week 2 have already been provided separately in the lab sheet for week 2.

In conclusion, *happy coding*!

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, 181âĂŞ199. https://link.springer.com/article/10.1007/BF01581144

4. *Improved algorithms for integer programming and related lattice problems.* Kannan, R. (1983). STOC 1983, pp. 193âĂŞ206. https://dl.acm.org/doi/10.1145/800061.808749