TCSS 487 Cryptography

Practical project – cryptographic library & app – part 2

Version: Nov 21, 2023

Your homework in this course consists of a programming project developed in two parts. Make sure to turn in the Java source files, and ***only*** the source files, for each part of the project. Note that the second part depends on, extends, and includes, the first part of the project.

You must include a report describing your solution for each part, including any user instructions and known bugs. Your report must be typeset in PDF (***scans of manually written text or other file formats are not acceptable and will not be graded, and you will be docked 20 points if a suitable report is missing for each part of the project***). For each part of the project, all source files and the report must be in a single ZIP file (***executable/bytecode files are not acceptable: you will be docked 5 points for each such file you submit with your homework***).

Each part of the project will be graded out of 40 points as detailed below, but there will be a total of 10 bonus points for each part as well.

You can do your project either individually or in a group of up to 3 (but no more) students. Always identify your work in all files you turn in. If you are working in a group, both group members must upload their own copy of the project material to Canvas, clearly identified.

Remember to cite all materials you use that is not your own work (e.g. implementations in other programming languages that you inspired your work on). *Failing to do so, as well as copying (with or without modifications) existing Java implementations of the algorithms described herein, constitutes plagiarism and will be reported to the Office of Student Conduct & Academic Integrity*.

Objective: implement (in **Java**) a library and an app for asymmetric encryption and digital signatures at the 256-bit security level (***NB: other programming languages are not acceptable and will not be graded***).

Algorithms:

- SHA-3 derived function KMACXOF256;
- DHIES encryption and Schnorr signatures with elliptic curves;

PART 2: Elliptic curve arithmetic

In what follows, keep in mind that all arithmetic will be modular (it is *not* plain integer arithmetic: you must take the remainder of the division by the appropriate modulus after each BigInteger operation, and any other apparent division operation actually stands for a multiplication of the numerator by the modular inverse of the denominator).

The elliptic curve that will be implemented is known as the Ed448-Goldilocks curve (a so-called Edwards curve), defined by the following parameters:

- $p := 2^{448} - 2^{224} - 1$, a prime number defining the finite field $\mathbb{F}_p$.
- curve equation: $x^2 + y^2 = 1 + dx^2y^2$ with $d = -39081$.

A point on Ed448-Goldilocks is represented by a pair (x, y) of integers satisfying the curve equation above. The curve has a special point $G := (x_0, y_0)$ called its public generator, with $y_0 = -3 \pmod p$ and x_0 a certain unique even number (see Appendix A for a method to compute x_0).

Given any two points (x_1, y_1) and (x_2, y_2) on the curve, their sum is the point

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - x_1x_2}{1 - dx_1x_2y_1y_2}\right).$$

This is called the Edwards point addition formula.

The opposite of a point (x, y) is the point $(-x, y)$, and the neutral element of addition is the point $O := (0, 1)$.

The number of points n on any Edwards curve is always a multiple of 4, and for Ed448-Goldilocks that number is $n := 4r$, where:

$r = 2^{446} - 13818066809895115352007386748515426880336692474882178609894547503885$.

The Java class implementing points on Ed448-Goldilocks must offer a constructor for the neutral element, a constructor for a curve point given its x and y coordinates (both instances of the *BigInteger* class), and a constructor for a curve point from its y coordinate and the least significant bit of x (see details in Appendix A). Besides, it must offer a method to compare points for equality, a method to obtain the opposite of a point, and a method to compute the sum of the current point and another point.

Finally, the class must also support multiplication by scalar: given a point $P := (x, y)$ on the curve and a (typically very large) integer k modulo r, you must provide a method to compute the point $k \cdot P := P + P + \cdots + P$ (that is, the sum of P with itself k times) when $k > 0$, with $(-k) \cdot P = k \cdot (-P)$ and $0 \cdot P = O$ when $k \leq 0$. Warning: computing $k \cdot P$ by actually adding P to itself k times will *not* work in practice! You must use the so-called exponentiation (multiplication by scalar) algorithm for this (check out the course slides).

Services the app must offer for part 2:

The app does not need to have a GUI (a command line interface is acceptable), but it must offer the following services in a clear and simple fashion (each item below is one of the project parts):

• [***8 points***] Generate an elliptic key pair from a given passphrase and write the public key to a file.

BONUS: [*4 points*] Encrypt the private key from that pair under the given password and write it to a different file as well.

• [***8 points***] Encrypt a data file under a given elliptic public key file and write the ciphertext to a file.

BONUS: [*2 points*] Encrypt text input by the user directly to the app instead of having to read it from a file (but write the ciphertext to a file).

• [***8 points***] Decrypt a given elliptic-encrypted file from a given password and write the decrypted data to a file.

• [***8 points***] Sign a given file from a given password and write the signature to a file.

BONUS: [*2 points*] Sign text input by the user directly to the app instead of having to read it from a file (but write the signature to a file).

• [***8 points***] Verify a given data file and its signature file under a given public key file.

BONUS: [*2 points*] Verify text input by the user directly to the app instead of having to read it from a file (but read the signature from a file).

The actual instructions to use the app and obtain the above services must be part of your project report (in PDF).

High-level specification of the items above:

Notation: We adopt the same notation as for part 1, with the following extension: if P is a curve point, then its coordinates are $P = (P_x, P_y)$, and $s*P$ stands for the multiplication of the scalar factor s by the curve point P:

- Generating a (Schnorr/DHIES) key pair from passphrase pw:
  - $s \leftarrow$ KMACXOF256(pw, “”, 448, “SK”); $s \leftarrow 4s \pmod{r}$
  - $V \leftarrow s*G$
  - key pair: (s, V)
- Encrypting a byte array m under the (Schnorr/DHIES) public key V:
  - $k \leftarrow$ Random(448); $k \leftarrow 4k \pmod{r}$
  - $W \leftarrow k*V$; $Z \leftarrow k*G$
  - $(ka \,||\, ke) \leftarrow$ KMACXOF256(W_x, “”, 2×448, “PK”)
  - $c \leftarrow$ KMACXOF256(ke, “”, $|m|$, “PKE”) $\oplus m$

    - $t \leftarrow$ KMACXOF256(ka, m, 448, "PKA")
    - cryptogram: (Z, c, t)

- Decrypting a cryptogram (Z, c, t) under passphrase pw:
    - $s \leftarrow$ KMACXOF256(pw, "", 448, "SK"); $s \leftarrow 4s \bmod r$
    - $W \leftarrow s*Z$
    - (ka || ke) $\leftarrow$ KMACXOF256(W_x, "", 2×448, "PK")
    - $m \leftarrow$ KMACXOF256(ke, "", $|c|$, "PKE") $\oplus c$
    - $t' \leftarrow$ KMACXOF256(ka, m, 448, "PKA")
    - accept if, and only if, $t' = t$

- Generating a signature for a byte array m under passphrase pw:
    - $s \leftarrow$ KMACXOF256(pw, "", 448, "SK"); $s \leftarrow 4s \pmod r$
    - $k \leftarrow$ KMACXOF256(s, m, 448, "N"); $k \leftarrow 4k \pmod r$
    - $U \leftarrow k*G$;
    - $h \leftarrow$ KMACXOF256(U_x, m, 448, "T"); $z \leftarrow (k - hs) \bmod r$
    - signature: (h, z)

- Verifying a signature (h, z) for a byte array m under the (Schnorr/ DHIES) public key V:
    - $U \leftarrow z*G + h*V$
    - accept if, and only if, KMACXOF256(U_x, m, 448, "T") = h

References: the Ed448-Goldilocks elliptic curve was standardized in NIST FIPS 186-5 in <https://csrc.nist.gov/publications/detail/fips/186/5/final>, the current version of DHIES scheme was introduced and analyzed in <https://web.cs.ucdavis.edu/~rogaway/papers/dhies.pdf>, and Schnorr signatures were introduced in <https://link.springer.com/chapter/10.1007/0-387-34805-0_22>.

Grading:

The main class of your project (the one containing the `main()` method) must be called Main and be declared in file Main.java. Also, all input/output file names and passwords must be passed to your program from the command line (retrieved from the `String[] args` argument to the `main()` method in the Main class) . You will be docked 5 points if the main method is missing/malformed, or defined/duplicated in a different class, or if the class containing it is not called Main or defined in a different source, or if you fail to use the `String[] args` argument as required.

A zero will be assigned to any item in the app services that produces wrong or no output (or if the program crashes).

All your classes must be defined *without* a **package** clause (that is, they must be in the default, unnamed package). You will be docked 2 points for each source file containing a **package** clause.

You must include instructions on the use of your application and how to obtain the above services as part of your report. You will be docked 20 points if the report is missing for each project part or if it does not match the observed behavior of your application.

Remember that you will be docked 5 points for any `.class`, `.jar` or `.exe` file contained in the ZIP file you turn in. Also, a zero will be awarded for this part of the project if any evidence of plagiarism is found.

Appendix A: computing square roots modulo p

To obtain (x, y) from y and the least significant bit of x, one has to compute $x = \pm\sqrt{(1 - y^2)/(1 + 39081y^2)} \bmod p$. Besides the calculation of the modular inverse of $(1 + 39081y^2) \bmod p$, which one obtains with the method `modInverse()` of the `BigInteger` class, this requires the calculation of a modular square root, for which no method is readily available from `BigInteger` class. However, one can get the root with the following method, taking care to test if the root really exists (if no root exists, the method below returns `null`).

```java
/**
 * Compute a square root of v mod p with a specified least-significant bit
 * if such a root exists.
 *
 * @param   v   the radicand.
 * @param   p   the modulus (must satisfy p mod 4 = 3).
 * @param   lsb desired least significant bit (true: 1, false: 0).
 * @return  a square root r of v mod p with r mod 2 = 1 iff lsb = true
 *          if such a root exists, otherwise null.
 */
public static BigInteger sqrt(BigInteger v, BigInteger p, boolean lsb) {
    assert (p.testBit(0) && p.testBit(1)); // p = 3 (mod 4)
    if (v.signum() == 0) {
        return BigInteger.ZERO;
    }
    BigInteger r = v.modPow(p.shiftRight(2).add(BigInteger.ONE), p);
    if (r.testBit(0) != lsb) {
        r = p.subtract(r); // correct the lsb
    }
    return (r.multiply(r).subtract(v).mod(p).signum() == 0) ? r : null;
}
```

Appendix B: multiplication by scalar

The multiplication-by-scalar (aka "exponentiation") algorithms invoke the Edwards point addition formula in a specific fashion to compute $s \cdot P = P + P + \cdots + P$ (s times) efficiently. Doing this naively by repeated addition is completely infeasible for large s because it would take exponential time.

You will be docked points if you compute $s \cdot P$ by plain repeated addition instead of the algorithm below (or Montgomery's version described in the slides), since that approach would absolutely fail work in practice with real-world parameters. In particular, the whole second part of the project will fail to produce meaningful results, and none of the corresponding points will be assigned.

The simplest (not necessarily the most efficient, nor the most secure) version is shown below in pseudocode for ease of reference. See the course slides for a more detailed description and discussion.

```
// $s = (s_k\ s_{k-1} \dots s_1\ s_0)_2$, $s_k = 1$.
V = P;    // initialize with $s_k$*P, which is simply P
for (i = k - 1; i ≥ 0; i--) {    // scan over the k bits of s
    V = V.add(V);    // invoke the Edwards point addition formula
    if ($s_i$ == 1) {    // test the i-th bit of s
        V = V.add(P);    // invoke the Edwards point addition formula
    }
}
return V;    // now finally V = s*P
```

Appendix C: hints for debugging elliptic curve arithmetic

The most essential operation in elliptic curve cryptography is the computation of points of form $P \leftarrow k \cdot G$ given a scalar k and a point G, so it is crucial to make sure the operation of multiplying a point by scalar is correct.

The first and foremost way of promoting correctness is to ensure that the implementation satisfies the arithmetic properties this operation is supposed to satisfy. Thus, be sure to test if:

$0 \cdot G = O$
$1 \cdot G = G$
$G + (-G) = O$ where $-G = (p - x, y)$ for $G = (x, y)$
$2 \cdot G = G + G$
$4 \cdot G = 2 \cdot (2 \cdot G)$
$4 \cdot G \neq O$
$r \cdot G = O$

Also, for random integers k, ℓ, and m, be sure to test if the following properties hold (repeat the test for a large amount of such random integers):

$k \cdot G = (k \bmod r) \cdot G$
$(k + 1) \cdot G = (k \cdot G) + G$
$(k + \ell) \cdot G = (k \cdot G) + (\ell \cdot G)$
$k \cdot (\ell \cdot G) = \ell \cdot (k \cdot G) = (k \cdot \ell \bmod r) \cdot G$
$(k \cdot G) + \big((\ell \cdot G) + (m \cdot G)\big) = \big((k \cdot G) + (\ell \cdot G)\big) + (m \cdot G)$