TCSS 487 Cryptography

Practical project – cryptographic library & app – part 2

Version: Oct 6, 2024

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 128-bit security level (***NB: other programming languages are not acceptable and will not be graded***).

Algorithms:

- SHA-3 and derived SHAKE functions (from part 1 of this project);
- ECIES (elliptic curve integrated encryption scheme) and elliptic Schnorr digital signatures.

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 NUMS-256 curve (more precisely, NUMS ed-256-mers*), a so-called Edwards curve, defined by the following parameters:

- $p := 2^{256} - 189$, a prime number specifying the finite field $\mathbb{F}_p$.
- curve equation: $x^2 + y^2 = 1 + dx^2y^2$ with $d = 15343$.

A point on NUMS-256 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 $y_0 = -4 \pmod p$ 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 NUMS-256 that number is $n := 4r$ where r is the following prime number:

$$r = 2^{254} - 87175310462106073678594642380840586067.$$

The Java class implementing the NUMS-256 curve (and a nested class implementing the arithmetic of points on this curve) must follow the specification in **Appendix D**.

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.

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

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

• [***8 points***] Sign with Schnorr a given file from a password-derived private key and write the signature to a file.

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

BONUS: [***5 points***] Combine encryption and signing as above in a single operation that signs with Schnorr a given data file under the sender's password and encrypts the file and the signature with ECIES under the recipient's public key.

BONUS: [***5 points***] Combine decryption and verification as above in a single operation that decrypts the file and the signature with ECIES under the recipient's password-derived private key and verifies the resulting data file with Schnorr under the sender's public key 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:

NB: check out the hints on obtaining random nonces (e.g. k) in **Appendix E**.

- Generating a (Schnorr/DHIES) key pair (s, V) from a passphrase:
  Init SHAKE-128, absorb the passphrase properly converted to a byte array, squeeze a 256-bit byte array and create a BigInteger from it, then reduce this value mod r. Compute $V \leftarrow s \cdot G$, and if the least significant bit (LSB) of the ***x***-coordinate of V is 1, replace s by $r - s$ and replace V by $-V$ (so that now the LSB of the ***x***-coordinate of V becomes 0). The result is the private key s and the public key $V = s$.

- Encrypting a byte array message under the public key V:
  Pick a random 256-bit byte array via the Java SecureRandom class, convert it to a BigInteger and reduce it mod r. Let k be the resulting value. Compute $W \leftarrow k \cdot V$ and $Z \leftarrow k \cdot G$. Init SHAKE-256 and absorb the ***y***-coordinate of W (converted to a byte array), then squeeze two successive 256-bit byte arrays ka and ke from it. Init SHAKE-128 and absorb ke into it, then squeeze as many bytes as the message length and XOR them with the message, obtaining the symmetric ciphertext c. Init SHA-3-256, absorb ka and then c, and extract the digest t from it. The full cryptogram is the triple (Z, c, t).

- Decrypting a cryptogram (Z, c, t) under a passphrase:
  Recompute the private key s from the passphrase (same as in the key pair generation above) and compute point $W \leftarrow s \cdot Z$. Init SHAKE-256 and absorb the ***y***-coordinate of W (converted to a byte array), then squeeze two successive 256-bit byte arrays ka and ke from it. Init SHA-3-256, absorb ka and then c, and extract the expected digest t' from it. Init SHAKE-128 and absorb ke into it, then squeeze as many bytes as the length of c and XOR them with$c$, obtaining the expected message. Accept the cryptogram and return the message if, and only if, $t' = t$ (otherwise declare decryption error).

- Generating a signature for a byte array message under a passphrase:
  Recompute the private key s from the passphrase (same as in the key pair generation above). Pick a random 256-bit byte array via the Java SecureRandom class, convert it to a BigInteger and reduce it mod r. Let k be the resulting value. Compute point $U \leftarrow k \cdot G$. Init SHA-3-256 and absorb the ***y***-coordinate of U (converted to a byte array) and the message, extract the 256-bit byte array digest, convert it to a BigInteger and reduce it mod r. Let h be the result. Compute $z \leftarrow (k - h \cdot s) \bmod r$. The signature is the pair (h, z).

- Verifying a signature (h, z) for a byte array message under the public key V:
  Compute point $U' \leftarrow z \cdot G + h \cdot V$. Init SHA-3-256 and absorb the ***y***-coordinate of U' (converted to a byte array) and the message, extract the 256-bit byte array digest, convert it to a BigInteger and reduce it mod r. Let h' be the result. Accept the signature if, and only if, $h' = h$.

References: the NUMS-256 elliptic curve and other related curves were introduced in <https://eprint.iacr.org/2014/130.pdf> and <https://link.springer.com/article/10.1007/s13389-015-0097-y>, the current version of ECIES (called DHIES at the time) was introduced and analyzed in <https://web.cs.ucdavis.edu/~rogaway/papers/dhies.pdf>, and Schnorr digital signatures were introduced in <https://link.springer.com/chapter/10.1007/0-387-34805-0_22>.

NB: there is a typo in the encryption algorithm (Figure 1) of the defining DHIES paper cited above, in that the authentication tag computation should read $tag \leftarrow \mathcal{T}(macKey, encM)$ rather than $tag \leftarrow \mathcal{T}(macKey, M)$.

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 *not* use JUnit for your tests (*projects that rely on JUnit will not be graded*).

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 $x = \pm\sqrt{(1-y^2)/(1-dy^2)} \bmod p$. Besides computing the modular inverse of $(1-dy^2) \bmod p$, 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 the `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-1} … s_1 s_0)_2.
    V = O;    // initialize with the neutral element O
    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:

$0 \cdot G = \mathrm{O}$
$1 \cdot G = G$
$G + (-G) = O$ 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):

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

Appendix D: specification of the Edwards class and its nested Point class

```
/**
 * Arithmetic on Edwards elliptic curves.
 */
public class Edwards {

    /**
     * Create an instance of the default curve NUMS-256.
     */
    public Edwards() { /* ... */ }

    /**
     * Determine if a given affine coordinate pair P = (x, y)
     * defines a point on the curve.
     *
     * @param x x-coordinate of presumed point on the curve
     * @param y y-coordinate of presumed point on the curve
     * @return whether P is really a point on the curve
     */
    public boolean isPoint(BigInteger x, BigInteger y) { /* ... */ }
```

```java
/**
 * Find a generator G on the curve with the smallest possible
 * y-coordinate in absolute value.
 *
 * @return G.
 */
public Point gen() { /* ... */ }

/**
 * Create a point from its y-coordinate and
 * the least significant bit (LSB) of its x-coordinate.
 *
 * @param y the y-coordinate of the desired point
 * @param x_lsb the LSB of its x-coordinate
 * @return  point (x, y) if it exists and has order r,
 * otherwise the neutral element O = (0, 1)
 */
public Point getPoint(BigInteger y, boolean x_lsb) { /* ... */ }

/**
 * Display a human-readable representation of this curve.
 *
 * @return a string of form "E: x^2 + y^2 = 1 + d*x^2*y^2 mod p"
 * where E is a suitable curve name (e.g. NUMS ed-256-mers*),
 * d is the actual curve equation coefficient defining this curve,
 * and p is the order of the underlying finite field F_p.
 */
public String toString() { /* ... */ }

/**
 * Edwards curve point in affine coordinates.
 * NB: this is a nested class, enclosed within the Edwards class.
 */
public class Point {

    /**
     * Create a copy of the neutral element on this curve.
     */
    public Point() { /* ... */ }

    /**
     * Create a point from its coordinates (assuming
     * these coordinates really define a point on the curve).
     *
     * @param x the x-coordinate of the desired point
     * @param y the y-coordinate of the desired point
     */
    private Point(BigInteger x, BigInteger y) { /* ... */ }
```

```
        /**
         * Determine if this point is the neutral element O on the curve.
         *
         * @return true iff this point is O
         */
        public boolean isZero() { /* ... */ }

        /**
         * Determine if a given point P stands for
         * the same point on the curve as this.
         *
         * @param P a point (presumably on the same curve as this)
         * @return true iff P stands for the same point as this
         */
        public boolean equals(Point P) { /* ... */ }

        /**
         * Given a point P = (x, y) on the curve,
         * return its opposite -P = (-x, y).
         *
         * @return  -P
         */
        public Point negate() { /* ... */ }

        /**
         * Add two given points on the curve, this and P.
         *
         * @param   P  a point on the curve
         * @return  this + P
         */
        public Point add(Point P) { /* ... */ }

        /**
         * Multiply a point P = (x, y) on the curve by a scalar m.
         *
         * @param   m   a scalar factor (an integer mod the curve order)
         * @return  m*P
         */
        public Point mul(BigInteger m) { /* ... */ }

        /**
         * Display a human-readable representation of this point.
         *
         * @return a string of form "(x, y)" where x and y are
         * the coordinates of this point
         */
        public String toString() { /* ... */ }

    }
}
```

Appendix E: hints on the generation of nonces for ECIES and Schnorr

The ECIES scheme relies on a truly secure source of randomness for encryption. A sensible way of getting a suitable random number k modulo the group order r in Java is this:

```
int rbytes = (E.r.bitLength() + 7) >> 3;
var k = new BigInteger(new SecureRandom().generateSeed(rbytes << 1)).mod(E.r);
```

Notice that twice as many bytes as the size of r are used to make sure the distribution of k is as close to uniform as possible, with any discrepancy being below the security threshold.

By contrast, Schnorr signatures can typically improve on that by including randomness from the signing key s and the signed message m. This requires hashing together s, m, and a randomness seed generated just like above, then squeezing twice as many bytes as the size of r and finally reducing modulo r.