TCSS 487 Cryptography

Practical project – cryptographic library & app

Version: Jan 26, 2020

Your homework in this course consists entirely of programming assignments (the theoretical aspects are covered in the quizzes). Make sure to turn in the source files for the *whole* project completed so far when turning in each individual part: your project is incremental, and all previously completed parts must continue to work when the new parts are added. Also, include all test input files you use to test your compiler, and a short report briefly describing your solution for each part. 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 all project files you turn in must be in one ZIP file.

The project is worth 150 points and is divided in **5 parts** indicated below plus a final **presentation** in the last lectures. Each part is worth 20 points but includes 5 bonus points as indicated below. The presentation is worth 50 points, with 10 bonus points for a live demo.

You must present your project in the last lectures (~5 min per group, each group consisting of 1 or 2 people). Always identify your work. All members of a group must upload their own copy of the project material, clearly identified.

Objective: implement (in **Java**) a library and an app for asymmetric encryption and digital signatures at the 256-bit security level.

Algorithms:

- SHA-3 derived function KMACXOF256;
- ECDHIES encryption and Schnorr signatures;

Symmetric cryptography:

All required symmetric functionality is based on the SHA-3 (Keccak) machinery, except for the external source of randomness.

Specifically, this project requires implementing the KMACXOF256 primitive (and the supporting functions bytepad, encode_string, left_encode, right_encode, and the Keccak algorithm) as specified in the NIST Special Publication 800-185 <https://dx.doi.org/10.6028/NIST.SP.800-185>. Test vectors for all functions derived from SHA-3 (including, but not limited to, KMAXXOF256) can be found

at <https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/cSHAKE_samples.pdf>.

Additional resource: if you clearly and conspicuously provide explicit attribution in the source files and documentation of your project (failing to do so would be plagiarism), you can inspire your Java implementation of SHA3 and the derived function SHAKE256 (both needed to implement cSHAKE256 and then KMACXOF256) on Markku-Juhani Saarinen's very readable C implementation: <https://github.com/mjosaarinen/tiny_sha3/blob/master/sha3.c>. NB: this is just a source of inspiration for your work and does not mean everything you might need is in there, especially if you need to bufferize the input.

Global parameters:

- $p := 2^{521} - 1$, a Mersenne prime.
- $E_{521} : x^2 + y^2 = 1 + dx^2y^2$ with $d = -376014$, an Edwards curve over $\mathbb{F}_p$.
- $G := (x, y)$, a point on E_{521} with $\boldsymbol{x = 4}$ and y even even.

Elliptic curve arithmetic:

Given any two points (x_1, y_1) and (x_2, y_2) on the curve E_{521}, their sum is the point

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

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 on any Edwards curve is always a multiple of 4; for the curve E_{521} that number is $n := 4r$, where:

$$r = 2^{519} - 337554763258501705789107630418782636071\backslash$$
$$904961214051226618635150085779108655765.$$

The Java class implementing points on E_{521} must offer constructors for the neutral element, a constructor for a curve point from its x and y coordinates (both instances of the *BigInteger* class), and a constructor for a curve point from its x coordinate and the least significant bit of y (see details in the appendix). Besides, it must offer methods to compare points for equality, to obtain the opposite of a point, and to compute the sum of two points (specifically, the sum of the current instance and another point).

Services offered by the app:

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):

1. Compute a plain cryptographic hash of a given file (this requires implementing and testing cSHAKE256 and KMACXOF256 first).
   BONUS: Compute a plain cryptographic hash of text input by the user directly to the app instead of having to be read from a file.

2. Encrypt a given data file symmetrically under a given passphrase.
   Decrypt a given symmetric cryptogram under a given passphrase.
   BONUS: Compute an authentication tag of a given file under a given passphrase.

3. Generate an elliptic key pair from a given passphrase and write the public key to a file.
   BONUS: encrypt the private key under the given password and write it to a file as well.

4. Encrypt a data file under a given elliptic public key file.
   Decrypt a given elliptic-encrypted file from a given password.
   BONUS: encrypt/decrypt text input by the user directly to the app instead of having to be read from a file.

5. Sign a given file from a given password and write the signature to a file.
   Verify a given data file and its signature file under a given public key file.
   BONUS: offer the possibility of encrypting a file under the recipient's public key and also signing it under the user's own private key.

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

The numbers above indicate in which part of the project they have to be available, according to the calendar below:

Part 1: Jan 21
Part 2: Feb 04
Part 3: Feb 18
Part 4: Feb 29
Part 5: Mar 09
Presentation: Mar 10 and 12

High-level specification:

We adopt the notation from NIST SP 800-185: KMACXOF256(*k*, *m*, *L*, *S*) is the Keccak message authentication code (KMAC) for key *k*, authenticated data *m*, output bit length *L*, and diversification string *S*. The result, a byte array of bit length *L*, is interpreted as a Java *BigInteger* when it occurs as part of an arithmetic expression (otherwise it is a play byte string). Furthermore, Random(*L*) is assumed to be a strong random number generator yielding a uniformly random binary string of length *L* bits (use the Java *SecureRandom* class).

Notation: if a is a byte array, its length in bits is denoted $|a|$, and if a and b are byte arrays, their concatenation is denoted $a \,||\, b$. Additionally, the notation $(a \,||\, b) \leftarrow$ KMACXOF256(…, …, $2n$, …) means to extract $2n$ bits from the KMACXOF256 sponge and splitting it sequentially into two pieces a and b of the same length n bits. It is assumed that all files and strings involved are converted to byte strings before they are processed cryptographically. We overload the notation and also represent by a an $|a|$-bit integer whose binary (base 2) value is spelled by the byte array a. If P is a curve point, then its coordinates are $P = (P_x, P_y)$. Apart from this, we follow the notation in the NIST Special Publication 800-185:

- Computing a cryptographic hash digest h of a given byte array m:
  - $h \leftarrow$ KMACXOF256("", m, 512, "D")

- Compute a message authentication code t of a byte array m under the passphrase pw:
  - $t \leftarrow$ KMACXOF256(pw, m, 512, "T")

- Encrypting a byte array m symmetrically under the passphrase pw:
  - $z \leftarrow$ Random(512)
  - $(ke \,||\, ka) \leftarrow$ KMACXOF256($z \,||\, pw$, "", 1024, "S")
  - $c \leftarrow$ KMACXOF256(ke, "", $|m|$, "SKE") $\oplus\, m$
  - $t \leftarrow$ KMACXOF256(ka, m, 512, "SKA")
  - symmetric cryptogram: (z, c, t)

- Decrypting a symmetric cryptogram (z, c, t) under the passphrase pw:
  - $(ke \,||\, ka) \leftarrow$ KMACXOF256($z \,||\, pw$, "", 1024, "S")
  - $m \leftarrow$ KMACXOF256(ke, "", $|c|$, "SKE") $\oplus\, c$
  - $t' \leftarrow$ KMACXOF256(ka, m, 512, "SKA")
  - accept if, and only if, $t' = t$

- Generating a (Schnorr/ECDHIES) key pair (s, V) from a passphrase pw:
  - $s \leftarrow$ KMACXOF256(pw, "", 512, "K"); $s \leftarrow 4s$
  - $V \leftarrow s*G$

- Encrypting a byte array m under the (Schnorr/ECDHIES) public key V:
  - $k \leftarrow$ Random(512); $k \leftarrow 4k$
  - $W \leftarrow k*V$; $Z \leftarrow k*G$
  - $(ke \,||\, ka) \leftarrow$ KMACXOF256(W_x, "", 1024, "P")
  - $c \leftarrow$ KMACXOF256(ke, "", $|m|$, "PKE") $\oplus\, m$
  - $t \leftarrow$ KMACXOF256(ka, m, 512, "PKA")
  - cryptogram: (Z, c, t)

- Decrypting a cryptogram (Z, c, t) under the passphrase pw:
  - $s \leftarrow$ KMACXOF256(pw, "", 512, "K"); $s \leftarrow 4s$
  - $W \leftarrow s*Z$
  - $(ke \,||\, ka) \leftarrow$ KMACXOF256(W_x, "", 1024, "P")

- $m \leftarrow$ KMACXOF256(ke, "", $|c|$, "PKE") $\oplus$ c
- $t' \leftarrow$ KMACXOF256(ka, m, 512, "PKA")
- accept if, and only if, $t' = t$

- Generating a signature σ for a byte array *m* under the passphrase *pw*:
  - $s \leftarrow$ KMACXOF256(*pw*, "", 512, "K"); $s \leftarrow 4s$
  - $k \leftarrow$ KMACXOF256(*s*, *m*, 512, "N"); $k \leftarrow 4k$
  - $U \leftarrow k*G$;
  - $h \leftarrow$ KMACXOF256(U_x, *m*, 512, "T"); $z \leftarrow (k - hs) \bmod r$
  - $\sigma \leftarrow (h, z)$

- Verifying a signature $\sigma = (h, z)$ for a given byte array *m* under the (Schnorr/ECDHIES) public key *V*:
  - $U \leftarrow z*G + h*V$
  - accept if, and only if, KMACXOF256(U_x, *m*, 512, "T") = *h*

Appendix: computing square roots modulo p

To obtain (x, y) from x and the least significant bit of y, one has to compute $y = \pm\sqrt{(1 - x^2)/(1 + 376014x^2)} \bmod p$. Besides the calculation of the modular inverse of $(1 + 376014x^2) \bmod p$, 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*).

```
/**
 * 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;
}
```