TCSS 487 Cryptography

Practical project – cryptographic library & app

Version: Mar 28, 2022

Your homework in this course consists of a programming project. Make sure to turn in the Java source files, and *only* the source files (you will be docked one point for each .class file you include in your material) for the whole project completed so far when turning in each part. The project is divided into two parts, but the second part depends on, extends, and includes the first part.

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***), and all project source files you turn in must be in one single ZIP file (***executable/bytecode files are not acceptable: you will be docked 5 points for each such file you submit with your homework***).

The project as a whole will be graded out of 80 points divided as indicated below, but there will be a total of 20 bonus points 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 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;
- 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* core algorithm itself) 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, KMACXOF256) 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 constitute 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!

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 E_{521} curve (a so-called Edwards curve), defined by the following parameters:

- $p := 2^{521} - 1$, a Mersenne prime defining the finite field $\mathbb{F}_p$.
- curve equation: $x^2 + y^2 = 1 + dx^2y^2$ with $d = -376014$.

A point on E_{521} 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 $x_0 = 4$ and y_0 a certain unique even number.

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 E_{521} that number is $n := 4r$, where:

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

(NB: the above backslash \ only means the number is too long to fit the line and continues on the line immediately below).

The Java class implementing points on E_{521} 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 x coordinate and the least significant bit of y (see details in the appendix). 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 n, 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 general! You must use the so-called exponentiation (multiplication by scalar) algorithm for this (check out the course slides).

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. • [***10 points***] Compute a plain cryptographic hash of a given file (this requires implementing and testing cSHAKE256 and KMACXOF256 first).
   BONUS: [*4 points*] Compute a plain cryptographic hash of text input by the user directly to the app instead of having to be read from a file.
   • [***10 points***] Encrypt a given data file symmetrically under a given passphrase.
   • [***10 points***] Decrypt a given symmetric cryptogram under a given passphrase.
   BONUS: [*4 points*] Compute an authentication tag (MAC) of a given file under a given passphrase.

2. • [***10 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 under the given password and write it to a file as well.
   • [***10 points***] Encrypt a data file under a given elliptic public key file.
   • [***10 points***] Decrypt a given elliptic-encrypted file from a given password.

BONUS: [*4 points*] encrypt/decrypt text input by the user directly to the app instead of having to be read from a file.

- [***10 points***] Sign a given file from a given password and write the signature to a file.
- [***10 points***] Verify a given data file and its signature file under a given public key file.

BONUS: [*4 points*] 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). Notice that the project is graded out of 80 points, but there are 100 points available.

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: in what follows, 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 $a \,||\, b$ and their exclusive-or is denoted $a \oplus b$. Additionally, the notation $(a \,||\, b) \leftarrow$ KMACXOF256(…, …, $2n$, …) means squeezing $2n$ bits from the KMACXOF256 sponge and splitting them 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 $P = (P_x, P_y)$., and $s*P$ stands for the multiplication of the scalar factor s by the curve point P. A call to Random(512) is assumed to return 512 uniformly random bits (64 bytes) sampled from the underlying processing platform (check out the Java `Random` and `SecureRandom` classes). Apart from this, we follow the notation in the NIST Special Publication 800-185:

- Computing a cryptographic hash h of a byte array m:
  - $h \leftarrow$ KMACXOF256("", m, 512, "D")
- Compute an authentication tag t of a byte array m under passphrase pw:
  - $t \leftarrow$ KMACXOF256(pw, m, 512, "T")
- Encrypting a byte array m symmetrically under passphrase pw:
  - $z \leftarrow$ Random(512)

    - $(ke \,||\, ka) \leftarrow \text{KMACXOF256}(z \,||\, pw, \text{“”}, 1024, \text{“S”})$
    - $c \leftarrow \text{KMACXOF256}(ke, \text{“”}, |m|, \text{“SKE”}) \oplus m$
    - $t \leftarrow \text{KMACXOF256}(ka, m, 512, \text{“SKA”})$
    - symmetric cryptogram: (z, c, t)

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

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

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

- Decrypting a cryptogram (Z, c, t) under passphrase pw:
    - $s \leftarrow \text{KMACXOF256}(pw, \text{“”}, 512, \text{“K”}); s \leftarrow 4s$
    - $W \leftarrow s*Z$
    - $(ke \,||\, ka) \leftarrow \text{KMACXOF256}(W_x, \text{“”}, 1024, \text{“P”})$
    - $m \leftarrow \text{KMACXOF256}(ke, \text{“”}, |c|, \text{“PKE”}) \oplus c$
    - $t' \leftarrow \text{KMACXOF256}(ka, m, 512, \text{“PKA”})$
    - accept if, and only if, $t' = t$

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

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

A quick observation on common sources of difficulties/errors:

Although this project is about cryptography, the most common difficulties are likely to be *not* in the cryptographic algorithms themselves, but in the low-level programming tasks (aka bit fiddling), specifically the implementation of the functions *bytepad*, *encode_string*, *left_encode*, *right_encode*, from the NIST specification.

I suggest you start reading their definition as soon as possible, follow the examples provided by NIST, then implement them as closely as possible to the description in the NIST document (think simple: do not try and write sophisticated code at first, write code that is easy to read and debug).

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;
}
```

Appendix: multiplication by scalar

The multiplication by scalar (aka "exponentiation") algorithms invoke the Edwards point addition formula in a specific fashion to compute $s * P = P + P +$

$\cdots + P$ (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} … 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: hints for debugging elliptic curve arithmetic

The most essential operation in elliptic curve cryptography is the computation of points of form $P \leftarrow k*G$ 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:

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

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

$k*G = (k \bmod r)*G$
$(k + 1)*G = (k*G) + G$
$(k + t)*G = (k*G) + (t*G)$
$k*(t*P) = t*(k*G) = (k*t \bmod r)*G$