TCSS 487 Cryptography

Practical project – cryptographic library & app – part 1

Version: Mar 25, 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 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;

PART 1: 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> and <https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/KMAC_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!

Services the app must offer for part 1:

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). See the detailed specification below:

• [***10 points***] Compute a plain cryptographic hash of a given file (this requires implementing and testing cSHAKE256 and KMACXOF256 first).

BONUS: [*5 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***] Compute an authentication tag (MAC) of a given file under a given passphrase.

BONUS: [*5 points*] Compute an authentication tag (MAC) of text input by the user directly to the app (instead of having to be read from a file) under a given passphrase.

• [***10 points***] Encrypt a given data file symmetrically under a given passphrase.

• [***10 points***] Decrypt a given symmetric cryptogram under a given passphrase.

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 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). 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 $a \mid\mid b$ and their exclusive-or is denoted $a \oplus b$. Additionally, the notation $(a \mid\mid b) \leftarrow$ 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 in two's complement) value is spelled by the byte array a (this will be useful for part 2 of the project). 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)
  - $(ke \mid\mid ka) \leftarrow$ KMACXOF256($z \mid\mid pw$, "", 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 passphrase pw:
  - $(ke \mid\mid ka) \leftarrow$ KMACXOF256($z \mid\mid pw$, "", 1024, "S")
  - $m \leftarrow$ KMACXOF256(ke, "", $|c|$, "SKE") $\oplus\ c$
  - $t' \leftarrow$ KMACXOF256(ka, m, 512, "SKA")
  - accept if, and only if, $t' = t$

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).

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 each `.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.