TCSS 487 Cryptography

Practical project – cryptographic library & app – part 1

Version: Sep 24, 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, all 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 hash and SHAKE extendable output functions;
- 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 SHA-3-256 and SHA-3-512 hash functions, as well as the SHAKE-128 and SHAKE-256 extendable output functions as specified in the Federal Information Processing Standard (FIPS) 202 <https://doi.org/10.6028/NIST.FIPS.202>. Test vectors for these functions are available from
https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/sha3/sha-3bytetestvectors.zip
and
<https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/sha3/shakebytetestvectors.zip>.

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 SHA-3 and the SHAKE 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). ***However, you must follow the SHA3SHAKE class interface specification at the end of this document***. See the detailed for each project item below:

• [***10 points***] Compute the SHA-3-256 and SHA-3-512 hashes for a user-specified file.

BONUS: [*2 points*] Compute also the SHA-3-224 and SHA-3-384 hashes of a user-specified file.

• [***10 points***] Compute SHAKE-128 and SHAKE-256 authentication tags (MACs) of user-specified length for a user-specified file under a user-specified passphrase.

BONUS: [*2 points*] Compute also SHAKE-128 and SHAKE-256 authentication tags (MACs) of user-specified length for text input by the user directly to the app (instead of having to be read from a file) under a user-specified passphrase.

• [***10 points***] Encrypt a user-specified data file symmetrically under a user-supplied passphrase. Use the 128-bit SHAKE-128 hash of the passphrase as a symmetric key, sample a random 128-bit nonce, then use SHAKE-128 as a stream cipher by hashing the nonce and the data file (NB: be sure to include with the cryptogram or it will be impossible to decrypt it).

BONUS: [*3 points*] Include a message authentication tag (MAC) in the cryptogram, using SHA-3-256 and the same symmetric key as for encryption.

• [***10 points***] Decrypt the symmetric cryptogram created by the encryption process above under the user-supplied passphrase.

BONUS: [*3 points*] Also verify the MAC from the cryptogram, computed as per the encryption bonus.

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:

<u>Notation</u>: All input and output data must be in the form of a Java byte[]. Data that is not in that format must be converted to-and-from it for the cryptographic algorithms to be applied. 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).

- Computing a cryptographic hash of data: use the static SHA3() method from SHA3SHAKE at the requested level (256 or 512, and the other allowed sizes for the bonus).

  For the bonus: use the suitable SHA-3 suffix for the desired security level.

- Compute an authentication tag of data under a passphrase: initialize an instance of SHA3SHAKE at the requested SHAKE level (128 or 256) via init(), then absorb() the passphrase, the data, and a final tag "T" (all properly converted to byte arrays), and finally squeeze() the desired number of output bytes.

  For the bonus: change the source of the data to authenticate (yes, it's that easy!).

- Encrypting a byte array symmetrically under a passphrase: hash the user-supplied passphrase via the static SHAKE() method with a 128-bit output bitlength to be the symmetric key, sample a random nonce as a byte array of length 128 bits with the Java *SecureRandom* class, init() an instance of SHA3SHAKE for SHAKE-128, absorb() the nonce and the symmetric key, then squeeze as many bytes as required to XOR onto the data to be encrypted. Make sure to include the random nonce as part of the generated cryptogram.

  For the bonus: init() a SHA3SHAKE instance for SHA-3-256 hashing, absorb() the same symmetric key as for encryption, absorb() the ciphertext (NB: not the plaintext data!), then digest() to obtain the 256-bit MAC tag. Make sure to include this tag as part of the cryptogram.

- Decrypting the above symmetric cryptogram: hash the user-supplied passphrase via the static SHAKE() method with a 128-bit output bitlength to be the symmetric key, init() an instance of SHA3SHAKE for SHAKE-128, absorb() the nonce (from the cryptogram) and the symmetric key, then squeeze as many bytes as required to XOR onto the ciphertext to be decrypted. Make sure to check if the output matches the original data file.

  For the bonus: init() a SHA3SHAKE instance for SHA-3-256 hashing, absorb() the same symmetric key as for encryption, absorb() the ciphertext, then digest() to obtain the expected 256-bit MAC tag, then compare it against the actual MAC tag included in the ciphertext (they must match).

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.

You must implement a class named SHA3SHAKE according to the interface specified at the end of this document. You will be docked 2 points for each nonconformity.

A zero will be assigned to any item in the app services that produces wrong or no output (or if the program crashes). A zero will be assigned to this part of the homework as a whole if you use any Java SHA-3 and/or SHAKE implementation apart from your own.

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 <u>not</u> 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.

Specification of the SHA3SHAKE class

Important: this class models a plain non-duplexing sponge (see the course slides). Therefore:

- A call to init() is required before any call to other instance (non-static) methods.
- Calling init() resets the sponge to its initial state (any prior hashing information is erased).
- After an init() call but before any squeeze() or digest() call, as many calls to absorb() are allowed. No calls to absorb() are allowed without at least one prior init() call, or after any squeeze() or digest() call (unless there is an intervening init() call).
- After all (zero or more) calls to absorb(), *either* as many calls to squeeze() *or else* as many calls to digest() are allowed, but squeeze() and digest() calls are mutually exclusive and must not be mixed (unless there is an intervening init() call).
- Each call to squeeze() returns an independent chunk of hashed bytes, but all calls to digest() return the same hash value (unless there is an intervening init() call).

```java
public class SHA3SHAKE {

  public SHA3SHAKE() {}

  /**
   * Initialize the SHA-3/SHAKE sponge.
   * The suffix must be one of 224, 256, 384, or 512 for SHA-3, or one of 128 or 256 for SHAKE.
   * @param suffix  SHA-3/SHAKE suffix (SHA-3 digest bitlength = suffix, SHAKE sec level = suffix)
   */
  public void init(int suffix) { /* … */ }

  /**
   * Update the SHAKE sponge with a byte-oriented data chunk.
   *
   * @param data  byte-oriented data buffer
   * @param pos  initial index to hash from
   * @param len  byte count on the buffer
   */
  public void absorb(byte[] data, int pos, int len) { /* … */ }

  /**
   * Update the SHAKE sponge with a byte-oriented data chunk.
   *
   * @param data  byte-oriented data buffer
   * @param len  byte count on the buffer (starting at index 0)
   */
  public void absorb(byte[] data, int len) { /* … */ }

  /**
   * Update the SHAKE sponge with a byte-oriented data chunk.
   *
   * @param data  byte-oriented data buffer
   */
  public void absorb(byte[] data) { /* … */ }

  /**
   * Squeeze a chunk of hashed bytes from the sponge.
   * Call this method as many times as needed to extract the total desired number of bytes.
   *
   * @param out   hash value buffer
   * @param len   desired number of squeezed bytes
   * @return the val buffer containing the desired hash value
   */
  public byte[] squeeze(byte[] out, int len) { /* … */ }

  /**
   * Squeeze a chunk of hashed bytes from the sponge.
   * Call this method as many times as needed to extract the total desired number of bytes.
```

```
     *
     * @param len   desired number of squeezed bytes
     * @return newly allocated buffer containing the desired hash value
     */
    public byte[] squeeze(int len) { /* … */ }

    /**
     * Squeeze a whole SHA-3 digest of hashed bytes from the sponge.
     *
     * @param out   hash value buffer
     * @return the val buffer containing the desired hash value
     */
    public byte[] digest(byte[] out) { /* … */ }

    /**
     * Squeeze a whole SHA-3 digest of hashed bytes from the sponge.
     *
     * @return the desired hash value on a newly allocated byte array
     */
    public byte[] digest() { /* … */ }

    /**
     * Compute the streamlined SHA-3-<224,256,384,512> on input X.
     *
     * @param suffix    desired output length in bits (one of 224, 256, 384, 512)
     * @param X         data to be hashed
     * @param out       hash value buffer (if null, this method allocates it with the required size)
     * @return  the out buffer containing the desired hash value.
     */
    public static byte[] SHA3(int suffix, byte[] X, byte[] out) { /* … */ }

    /**
     * Compute the streamlined SHAKE-<128,256> on input X with output bitlength L.
     *
     * @param suffix    desired security level (either 128 or 256)
     * @param X         data to be hashed
     * @param L         desired output length in bits (must be a multiple of 8)
     * @param out       hash value buffer (if null, this method allocates it with the required size)
     * @return  the out buffer containing the desired hash value.
     */
    public static byte[] SHAKE(int suffix, byte[] X, int L, byte[] out) { /* … */ }

}
```