Practical Project - Cryptographic Library & App - Part 1 and 2

Names: Elijah Immer, Khalid Rashid, Grant Koenig

Objective Implement (in Java) a library and an app for asymmetric encryption and digital signatures at the 128-bit security level.

Algorithms

1. SHA-3 hash and SHAKE extendable output functions.
2. ECDHIES encryption and Schnorr signatures.

We set up the project by making two classes, the SHA3SHAKE class which have the prebuilt foundation to build the algorithm as well as a main class. We started out with the absorb methods, squeeze methods, and the digest methods. Absorb methods XORs bytes into the sponge, tracks how many bytes filled in the current rate and calls Keccak to mix it, then it applies the SHA-3 padding and calls Keccak again. The squeeze method extract the length of the bytes from the sponge and copies bytes from the state into "out". The digest method squeezes "digest_length" bytes into the out buffer and returns it. The init method initialized and allocates the buffer and clears it, setting the "digest_length" from bits to bytes for the security size of the sponge.

After these, we decided to work on the steps of the Keccak function Θ(Theta),P+Π(Rho+Pi),X(Chi), and I(Iota). What we did first was try to translate it differently from reference C code but it failed badly so we ended up just using the code and directly translating it to Java and that was successful. We set the rounds to our constant in the instance field "KECCAK_ROUNDS" for the desired 24 rounds. We needed to implement a helper method used in Rho and Theta, as well as the rotate offset for Rho and the Pi lane permutations. In Main, it parses the security bits size (128,224,256,384,512), reads contents of the file into a byte array then goes through our sponge and prints the hashed result. By testing the code, we were comparing the outputs from our code to the outputs of the C code. Eventually we got the same output as the C code. We then implemented all the cases in Main so SHA3SHAKE can be used to encrypt and decrypt and can be used as a tool in terminal.

Part 2

Once we got all the SHA3SHAKE code finished, we needed to work on the Edwards Curve Implementation. We copied the codes foundation. We set up our instance fields, p,c,r, and point G. We used the given square root function as well as implement the point addition, scalar multiplication, the point methods, and the key generation. The key generation produces the private scalar s and a public point $V = s * G$ and the public key file stores V.x and V.y as hex on separate lines. After this, we went into main, made the keygen, encrypt, decrypt, signature and verification methods for the ECIES algorithm. We generate a random 384-bit scalar, compute W and Z, Used SHAKE-256 to get symmetric keys from W.y (64 bytes) and split it into ka and ke (first and next 32 bytes), encrypt ke using SHAKE to a keystream length m, and set c as m XOR keystream. We set authentication tag "t" by using our SHA3-256 algorithm, taking our authentication key "ka" and concatenate it on to the ciphertext "c", giving us a 32-byte digest "(Z,c,t)". In decryption, we compute W, derive "ka", "ke", from W.y with SHAKE-256, and compute "t'" using the same algorithm and set up. Then it tests whether t = t'. If it doesn't match it aborts, if it does match then we set m by XORing c and using SHAKE-128 to "ke".

In order to implement digital signatures using elliptic curve cryptography on the Edwards curve, we created two methods ec_sign and ec_verify. ec_sign receives a password and a file. it uses that password to Recompute the private key s. we used the java secure Random class to generate k from a random 384-bit byte arrays. we then compute U from $k \cdot G$ and absorb its y-coordinate and the message(file) before casting the resulting digest to a BigInteger mod(r) labeled h. Finally we computed z = $k - h \cdot s$. ec_verify receives a public key file, a signature file, and the signed file. after

reading these files and retrieving the point V from the public key file, and h from the signature file we computed U' = $z \cdot G + h \cdot V$. After this we absorbed the U'.y and the message(file) resulting in h'. Finally we return true if h' equals h, accepting the signature, and false, denying it, otherwise.

Instructions:

Compile: javac Main.java SHA3SHAKE.java Edwards.java
SHA3: java Main sha3 256 FILE
SHAKE RAND: java Main shake-random 128 SEED LEN
SHAKE XOR encrypt: java Main shake-encrypt 128 KEY FILE > out.bin
EC ENCRYPT: java Main ec-encrpyt KEY_FILE FILE
EC DECRYPT: java Main ec-decrypt KEY_FILE FILE.bin > decrypted.txt
SIGN: java Main ec-sign PASSWORD FILE > sig.txt
VERIFY: java Main ec-verify PUB_KEY_FILE sig.txt FILE

Source: https://github.com/mjosaarinen/tiny_sha3/tree/master