Trevor Tomlin
Michael Theisen
TCSS 487 Cryptography
06/04/2023

Programming Project Part 2

Our first step in moving forward with Part 2 of this Programming Assignment was to ensure that our issues with the Keccak implementation were fixed. A significant hurdle that we needed to deal with was finding out why our hashed values consistently gave us the wrong output to that of the test vectors despite seeming to have successfully been processed. While our function did technically work, we realized that there was an issue regarding the correct conversion for the vector's values in the **`cSHAKE`** samples, specifically sample #3. This led us to looking deeper into the FIPS PUB 202 and finding Section B.1 Conversion Functions. The input data is detailed as **`"00 01 02 03"`** with *N* being listed as an empty string. It shows that the encoded output is supposed to be **`"01 00"`**. We know that when translated to binary, this value is **`"00000001 00000000"`**. This was our first red flag, since the NIST SP 800-185 lists the **`encode_string()`** function as taking in an empty string and yielding **`"10000000 00000000"`** after putting it through the **`left_encode()`** function. This then led us to deduce that perhaps a portion of the "bit fiddling" problems we might experience have been manifested here, such that we would need to ensure that there is an additional conversion function that reverses every block of 8 bits before translating them back into a hexadecimal output.

After about 20 hours of trying to find out why it seemed to only be a situational issue, we came to the realization that the only time those two blocks of 8-bit sections were reversed in such a way was after they were passed from the **`left_encode()`**. Therefore we deduced that we should apply a **`reverser()`** conversion to the output from left_encode. The **`reverser()`** function splits the output into 8-bit chunks, reverses them, then concatenates them back together.

We had to deal with these types of bit fiddling issues constantly, as they continued to pop up after different implementations were finished, and subsequently caused the code to break, usually due to parsing issues of some kind with String bounds. If we were to go back and start from scratch, we would definitely choose byte arrays as our primary method for storing data instead of **`Strings`** and **`BigIntegers`**.

After dealing with the final implementation issue from Part 1 of the assignment and running tests, we moved on to the Elliptic Curve implementation. In this part, we created a class that represents a point on the Ed448 Goldilocks curve. Test cases from the project document and test vectors from the course were used to verify that our solution produced the correct results. The class supports negation, addition, multiplication using the specified algorithm, and initializing a point from the compressed form. We added the five basic services described in the programming document; generating a key pair, encrypting data, decryption data, signing, and verifying a signature.

As before with Part 1, our implementation uses a basic command line interface that is written in such a way to ensure the user can do all of the required operations when **`Main`** is run. When the user runs the application, they are given 11 choices to select from. Some of these choices have additional input requirements such as a filename, message, or password. The included options are:

1. Compute a plain cryptographic hash of a given file,
2. Compute an authentication tag (MAC) of a given file under a given passphrase,
3. Encrypt a given data file symmetrically under a given passphrase,
4. Decrypt a given symmetric cryptogram under a given passphrase,
5. Random bit generator,
6. Generate an elliptic key pair from a given passphrase and write the public key to a file,
7. Encrypt a data file under a given elliptic public key file and write the ciphertext to a file,
8. Decrypt a given elliptic-encrypted file from a given password and write the decrypted data to a file,
9. Sign a given file from a given password and write the signature to a file,
10. Verify a given data file and its signature file under a given public key file
11. Run Tests.

It is important to mention that we are still having issues with the formatting of the strings and we believe that there is an issue with converting between different encodings of strings and the endianness of the binary strings. This is one of many bugs that seems to have caused a cascading effect throughout the entire project. In some situations, we found that altering the **`left_encode()`** function would actually have intended results such as our initial error at the end of our implementation of **`encode_string()`**, where we originally wrote the return statement as: **`"return left_encode(len(S) || S)"`** , when we should have instead written it as: **`"return left_encode(len(S)) || S"`**.

This small parenthesis issue caused significant confusion when debugging. However after fixing the issue, we realized that a lot of other implementations were done so in a way that utilized this error. This was primarily due to the fact that the test used the examples that were in the NIST SP 800-185 documentation where **`left_encode()`** and **`encode_string()`** showed the same result being tested on **`'0'`** and an empty string **“”** respectively.

It is always easy to look back and know how we could better implement the assignment if we were to do it again. As Fred Brooks said in *The Mythical Man Month*, “Build one to throw away.” We plan on rebuilding the entire project from scratch later, as it is not only an intellectually stimulating process, but implementing SHA-3 derived functions and Elliptic Curve Cryptography looks very nice on a portfolio.