Andrew Morgan, 2017

CSE 178

Rust as a language

I decided to write my AES implementation in Rust, as it’s a language I've heard much about in the past few years but have never actually created a project in. I find one can read and watch tutorials about a programming language all they like, but they won't actually understand or get a feel for that language until they've done a project with it. For this reason, I've decided to do my implementation of AES in Rust.

It actually took me nearly a week to get used to Rust. Its syntax is familiar but different from other programming languages in many ways, though it has all the common structures: objects, loops, conditional statements, etc. The main part of Rust that I kept getting hung up on was its idea of ownership. Rust has its own style of memory management that does away with garbage collection. Instead of iterating through memory every so often and looking for unused memory locations, you can have, at any given time, only a single object pointing to a location in memory.

You can create unlimited references (like pointers) to an object, but at any time there cannot be more than one object that directly refers to a point in memory. If you try to set another variable to this point in memory, that is called a 'move', and the original variable that pointed to it will no longer have a memory location associated with it. This prevents a whole class of common run-time errors where the program tries to use or free a value that no longer exists in memory. Rust will prevent you from compiling a program that is susceptible to these issues.

Implementation and Design of AES

All implementations of AES-128-ECB follow the same generic outline. There are four main methods, AddRoundKey, SubBytes, ShiftRows and MixColumns. There are 11 rounds, an initial one with only the AddRoundKey step, nine with all main methods in a row and a final round that lacks the MixColumns step. Outside of this, one also needs to generate the key blocks used in the AddRoundKey step. This process, which uses the originally provided encryption key, is called Key Expansion and derives multiple 128-bit blocks, which are then XOR'd against the state block during AddRoundKey.

I looked at some example implementations online that mostly use arrays to store the contents of the state and round keys. While this was efficient, it was harder for me to conceptualize as most of the images of the state block I had seen was represented as a 4x4 matrix. I thus found a crate (external library in Rust) that had an implementation of matrices, including several operations such as matrix multiplication. Although not as elegant in some places compared to a single array, working with matrices which I could grab from using (row, col) calls was desirable versus just a one-dimensional array. I also used matrices to store the preset elements of AES, such as the RCON and the Substitution Box.

When I initially looked at the AES algorithm structure, I was confused about where people were getting the values for the RCON and S-Box. As it turned out, those values are the same across all implementations of AES, and are designed to prevent certain cryptographic attacks against the algorithm. While the values of both the RCON and the S-Box can be generated on the fly, preferable for low-memory devices, I decided for easier implementation purposes to simply include the necessary values directly in the source code, stored as Rust matrices. In the case of the RCON, in AES-128-ECB only the first 11 values (sans the first) are used, and thus that is all that was included.

These matrices were very easy to work with. While allowing for simple row and column getting and setting, the matrices also featured simple matrix multiplication. While I initially attempted to use matrix multiplication during the MixColumns step (which involved multiplication of the state with a fixed matrix), things didn't end up quite working correctly. Instead of matrix multiplication, I ended up mixing each column with each element's derived values bounded within Rijndael's Galois field, consistently producing the correct result.

For most of the steps, I tried to break them up into multiple methods wherever necessary. For instance, within MixColumns, there is a mix_single_column() method that performs the mix, which MixColumns simply calls for each column of the matrix it is given. ShiftRows is similar, where it will simply call a method, matrix_row_rotate(), that will rotate a single row an arbitrary amount of times. This ended up being a very useful design choice as I later used it to perform ShiftColumn during Key Expansion, but formatting a column as a row, then transposing it back into a column. The matrix crate's .transpose() method for matrices came in very handy for this purpose.

Conclusion

While there were some initial road bumps in getting used to a new programming language, I thoroughly enjoyed the process of implementing the algorithm, and now have a very thorough understanding of its inner workings. While my implementation was admittedly much more verbose than most, I find it easier to understand than other available implementations that I have come across. With the printout of my tool, I was able to provide other students with the subsequent encryption steps, and due to Rust being a compiled language I could do so without revealing the source code. Overall, it was a worthwhile project, and I learned much in terms of concept and language along the way.

Tests

There are a multitude of tests for many different functions, but only do functions test_enc_dec_1 and test_enc_dec_2 test encryption across the entire implementation.

Reference

Understanding AES Algorithm for Implementation