Poly1305 Authentication

Sambit Ghosh

December 2020

1 Abstract:

Poly1305 is a cryptographic one-time message authentication code. It can be used to verify the data integrity and authenticity of the message. Mbed-tls use this technique to authenticate the messages. In our project we implemented this authentication scheme using the language RUST. While implementing we have face some setbacks, some problems, some issues which we never faced before. In this report we will put some lights on those things.

2 Work Done:

While searching through the source code of the mbed-tls library one can identify that there are mainly two files which are used to implementing the poly1305 functionality named poly1305.h and poly1305.c. Our goal was to implement these two files. Apart from these there are many other dependencies of other files like platform.h, cipher.h etc. But to make our implementation concrete and runnable we included those dependencies in our implementation.

3 Code Links:

- Click here to view the Poly1305 implementation. Three files are there. One is main function $main.rs$, second one is $poly1305c.rs$ which includes the logic and function of the program. And the last one is $poly1305h.rs$, which is basically a header file, includes the constant and structures.

4 Problems and Solution

Now comes the most important and interesting part. We implemented those file from top to bottom in RUST. While doing this we came to know about many programming feature which we will discuss now.The immediate issue for us to find a proper file to work upon. We scan through the whole source code of mbed-tls and figured out a interesting topic named poly1305.

Now let's come to the technical obstacles we have faced through our work. From top to bottom we have maintained a fixed strategy. We convert the code to RUST in line by line manner first to understand and maintain the overall flow of the program. This helps us to identify the main intuition behind the code, the logic and flow of the code. This is a big plus for us to convert the code in shorter amount of time. In the first stage of our work it is a bit difficult to come up with exact syntax of RUST, for which we used Google to find the syntax. Google reduces our effort in a significant level. Now let's discuss the problems to implement the poly1305.h file.

1. This file's implementation is very much straight forward. This is basically a header file which included some macro definitions and some function declaration. Macros definitions are easily converted to RUST using $pub\ const < constant\_or\_structure\_name >$ syntax. And for the function declaration we directly implement the functions without declaring in one place, implement in other place because in RUST there is no provision of function declaration.

2. Another problem we faced is that, there are other header files which are being included in this file. But fortunately those header files are included here just to use the macro definitions. This time we included those definitions here just to avoid increasing number of small files.

Thus our poly1305.h is implemented in this way and named as poly1305h.rs. This file only contains the constant structure and few constants.

Now let's move forward to the main file implementation poly1305.c, which is doing the main hard-work.

1. The first problem which we encountered is how to access data of the header file. A simple google search solves the problem, using $use\ crate :: < header\_file\_name > :: < used\_structure\_or\_constant\_name >$

2. Now going forward we encountered several syntactical obstacles, this kinds of problem we solved easily by checking the RUST documentation. In this report we are not going to discuss those issues because its very trivial. Any programmer will face the same issue at any point of time. One search through browser can fix this within 3 minutes.

3. After run the program several kind compile and runtime issue has occured. To cope up with this situation we follow a strategy. In a way like start from main, first fix the first function and comment the other ones. After this include the second one and comment rest. This strategy helps us to debug and fix codes in a short span of time. As an example $main() \rightarrow run() \rightarrow mbedtls\_poly1305\_mac$. And from $mbedtls\_poly1305\_mac$, 3 function is invoked. Our strategy was to comment out the invocation of all the function which are being called from $mbedtls\_poly1305\_mac$, fix the error in that function and then one by one un-comment the three functions. Its kind of bottom up approach to solve the problem.

4. while checking the data flow of the program, we have came to know that RUST is not so much type flexible like C. At every step of our program we face data missmatch error. T0 solve this we started from the bottom and changed corresponding parameter data type accordingly.

5. In poly1305.c file there are two places where the $BYTES\_TO\_U32\_LE$ function is called. In this two instances two different sized array is being passed. C handles it using an array pointer. But one of the array is mut other is not. Therefore to avoid the chaos in RUST, we made two separate function one is for mut pointer data $BYTES\_TO\_U32\_LE2$ and other is for static constant data $BYTES\_TO\_U32\_LE$ because RUST doesn't allow data to be mut and constant sized at the same time.

6. The next big problem we have encountered is the 32 bit multiplication. Every time we are doing this the memory is getting error and the result is panic. There is a inbuilt function RUST $wrapping\_mul$, which is used to solve this issue. Refer to $mul64$ function in poly1305c.rs. This $wrapping\_mul$ function solves the problem of out of bound multiplication by detecting it and solving it.

7. In $mbedtls\_poly1305\_update$ function, there is a call to $poly1305\_process$ function. One of the parameter is a array pointer which pointing to the middle of the array. We can't just send the array slice as parameter because it need to mutable. That's why we created another $dummy$ array , copying the data from previous to new and sent it as a parameter.

5 Expected Output

We made some test case to find the correctness of our program. We took two test cases to run the program. One of them is output the wrong and other one is correct. All the data and keys are mentioned in $poly1305c.rs$ file. In one of them the authentication failed and other will pas the test. To create this mac using mentioned keys we used freely available online tools found via google search.

Expected Output:[3a, 97, fa, cd, fb, d, b2, fd, 4a, bf, f6, af, 41, 49, f5, 1b]
Calculated mac is [73, eb, 68, 25, fd, d, b2, fd, 4a, bf, f6, af, 41, 49, f5, 1b] **Output:** failed

Expected Output:[85, e6, a9, a5, 42, 2b, 80, 9, 9d, ca, 5c, bc, 20, 70, 75, c0]
Calculated mac is [85, e6, a9, a5, 42, 2b, 80, 9, 9d, ca, 5c, bc, 20, 70, 75, c0] **Output:** passed

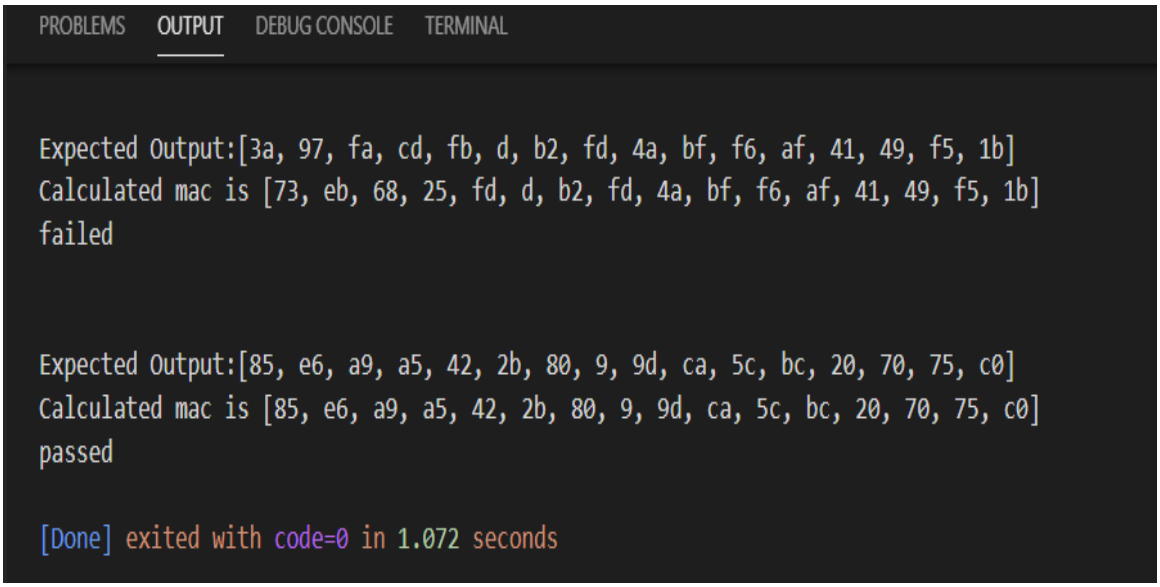

Figure 1: This is an output by running the code in visual studio editor.

References

[1] https://tls.mbed.org/source-code

[2] https://en.wikipedia.org/wiki/Poly1305

[3] https://www.google.com/

[4] https://tls.mbed.org/api/poly1305_8h.html