Katherine M. Orozco, Ian Phelps, Arthur Phung, **Dr. Mohamed El-Hadedy**

*Electrical and Computer Engineering Department at California State Polytechnic University, Pomona*

Contact: [kmorozco@cpp.edu](mailto:kmorozco@cpp.edu), [iphelps@cpp.edu](mailto:iphelps@cpp.edu), [arthurphung@cpp.edu](mailto:arthurphung@cpp.edu), [**mealy@cpp.edu**](mailto:mealy@cpp.edu)

TinyJAMBU: Implementation on FPGA with Verilog

*Abstract –* With the increased need for data security, the need for stronger algorithms to encrypt data has also risen. One such model is TinyJAMBU, which is derived from JAMBU. The aim of this project is to translate the TinyJAMBU-128 algorithm into Verilog, to be used on a NEXYS A7 FPGA board. A single message is able to be encrypted using only a 128-bit key and 96-bit nonce. The primary design goals are to ensure that the algorithm is as optimized as possible, in cases where a key can be stored. Due to time constraints, only the initialization and encryption phases have been programmed.

I. INTRODUCTION

TinyJAMBU is derived from JAMBU, which is a lightweight encryption mode that is based on keyed permutation modes [1]. The permutation modes of TinyJAMBU are based on a 128-bit nonlinear feedback shift register [1], which is passed through a series of logic gates. Although the original version of TinyJAMBU presented in the provided document is meant for VHDL, the design can be adapted to be written in Verilog. The process can be broken up into three distinct sections: the initialization, encryption, and decryption. For sake of simplicity, the project focuses on the first two phases, that being initialization and encryption. For the initialization, a 128-bit key and 96-bit nonce are required. The encryption phase uses the key and nonce to encrypt the plaintext. It can be noted that TinyJAMBU can be modified to fit a larger 192-bit and 256-bit key. The primary security goal for TinyJAMBU is to protect a single message [1]. This can be run several times to cover a series of messages, however for the project, only a single message is to be tested.

In this report, section I covers background information on a TinyJAMBU, section II covers the algorithm of the TinyJAMBU, section III covers each individual module in the design of the code, section IV covers the results achieved in the project, and section V covers the conclusion.

II. TinyJAMBU ALGORITHM

The TinyJAMBU encryption mode uses a 128bit keyed permutation and a message block that is 32bits. The permutation consists of a certain number of rounds and on every round the 128-bit nonlinear feedback shift register called state is updated. Both the length of the associated data and the plaintext in TinyJAMBU-128 are less than 2 to the power of 50 bits long and the authentication tag is 8 bytes. In the initialization phase both the key and the nonce are set up. The key is set up to randomize the state using the keyed permutation. This is done by setting all bits of the state register to 0 and then updating the state using a permutation of 1024. The nonce setup has three stages: first the frame bits of the nonce are XORed with the state register, then the register is updated with a permutation of 640, and lastly the 32 bits of the nonce are XORed with the state register. After the initialization we process the associated data using the same process as we did to set up the nonce. However, this time we use the associated data in place of the 32-bit nonce. Lastly, we encrypt the plaintext. This is done again with a similar process; in each step the frame bits are XORed with state, then the state is updated with a permutation of 1024 and then the 32-bit plaintext is XORed with the state register giving us the 32 ciphertext.

III. DESIGN

The C code begins by defining certain parameters of the overall design, those begin the sizes of different frame bits and rotations of the algorithm. Next a state update function is initialized that calls for a state input, which is four 32bit registers, a 128-bit key, and a variable that will determine the number of times a state is updated. To update the state, the 70 and 85th bits are NANDed together and then ORed with the 0-bit, the 47th bit and the 91st bit as well as a specified bit of the key determined by the number of the rotations to generate feedback that updates the state.

The initialization function takes a 128-bit key, 96-bit initial value, and the state as inputs. It first begins by setting all bits of the state registers to 0 and then begins to update them based on the previously described state update function. After the state register has been updated by a single call of the state update function, initial value Frame Bits are introduced to the state register by getting XORed to the first 32 bits of the state register. After this, the state register is updated once again using the state update function, this time using fewer rotations however, and finally a specific bit of the 96-bit initial value is XORed with the third 32-bit chunk (or state[3]) of the state register; this process happens three times.

The process\_ad function allows the code to process the associated data. The implementation of adlen allows the code to verify the if the blocks of associated data are full (multiples of 4) or partial. In the cases where the blocks of associated data are full, the FrameBits are XORed with the state. The state then becomes updated by the keyed permutation which then 32 bits of the associated data is XORed with the state. In the cases of partial blocks of associated data, the last block is XORed with the state. Additionally, the remaining number of bytes of the associated data is XORed with the state.

Finally, the process of encrypting the plain text can begin. First, the key and the state register go through the initialization phase. Next, the associated data is processed using the key, the state register, and the length of the associated data itself. After this, the plain text is processed by XORing the first 32 bits of the state register (state[1]) with the Frame Bits of the plain text. The state register is then updated using a large number of rotations. After that a bit, specified by the number of times the process has been executed, of a character m is XORed with state[3]. Lastly, a character c is assigned the value of the result of the XORing of state[2] and m; this process is also repeated three times. If the length of the character m is not a multiple of four the remaining data must be processed through a similar operation. Finally, we enter the finalization stage of the algorithm. In this stage we assume a tag length of 8 bytes; from here, state[1] is XORed with the finalization frame bits and the state update function is called once more. After this state[2] gets assigned the value of the 0 bit of the character mac. This process is repeated once more with the only difference being that in the second execution state[2] is now assigned the value of the 1st bit of mac. The encryption is complete when the length of the character c is expanded to the length of character m plus 8 bits and the memory function is called.

IV. RESULTS

Unfortunately, we were not able to get a working Verilog code by the deadline. We attempted to write a pseudo code utilizing the C code as a reference; however, translating the languages deemed to be too difficult due to the differences in syntaxes and rules.

The results that we wanted to achieve was to implement the given C code in Vivado using Verilog. Then to implement UART which would allow us to communicate with the FPGA board using a computer. To test the results, we would input the given the key, nonce, plain text, and associated data and compare the outputs. These can be found in the LWC\_AEAD\_KAT\_128\_96.txt file.

For example:

Count = 37

Key = 000102030405060708090A0B0C0D0E0F

Nonce = 000102030405060708090A0B

PT = 00

AD = 000102

CT = FE397BE6A452C3A584

In count 37, the key, nonce, plain test, associated data, and cipher text are all given in lines 2-6. The cipher text being the output.

V. CONCLUSION

In conclusion, we were unable to translate the C code in Verilog nevertheless implement UART. Despite our struggles, we gained a better understanding on encryption and the abilities of TinyJAMBU.

VI. REFERENCES

[1] “TinyJAMBU: A family of lightweight authenticated ...” [Online]. Available: <https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/TinyJAMBU-spec.pdf>. [Accessed: 21-Nov-2021].