TLS-Inspired Custom Secure Network Communication System

Danlu Liu
Zixuan Nie
Jiahui Wang

April 2021

1.0 Overview

1.1 Motivation for Project

The security of data when it is being sent has become important in the information technology age. However, many of the algorithms used in cryptography are very computationally intensive and operate on large data structures [1], which place great loads on the systems' CPU and may not be able to meet modern requirements for speed. Hardware implementation of cryptographic algorithms have been shown to enhance the speed and reduce delay [2]. By using hardware accelerators to perform the cryptographic computations and thus freeing up the CPU, the overall efficiency of the system can be improved.

There has been a good deal of research into hardware implementation of cryptographic algorithms. In [1], researchers presented the design and implementation of a RSA crypto accelerator, using HDL-based design methodology, and highlighted the advantages of a full-featured, flexible, and parameterized design. The authors of [3] used Verilog to implement AES-128/192/256 encryption and decryption circuits in hardware which used resources more efficiently and met current demands in performance, size, and flexibility. In a survey study on hardware implementation of cryptographic algorithms using FPGAs [2], the researchers' examined parameters such as throughput, operating frequency, number of slice registers used, number of clock cycles and concluded that hardware implementation is quite a flexible and secure method that also give improvements to speed and efficiency.

1.2 Project Goal

The main goal of this project is to explore network security and implement cryptographic algorithms in hardware. This project will include an RSA encryption/decryption module and AES encryption/decryption module implemented on the FPGAs, along with supporting infrastructure that will allow the functionality to be tested.

While it is debatable that this project will present a better design than what is referenced in the papers above, it is expected that the hardware implementation of the cryptographic algorithms will lead to a faster system than a purely software implementation.

1.3 Block Diagram

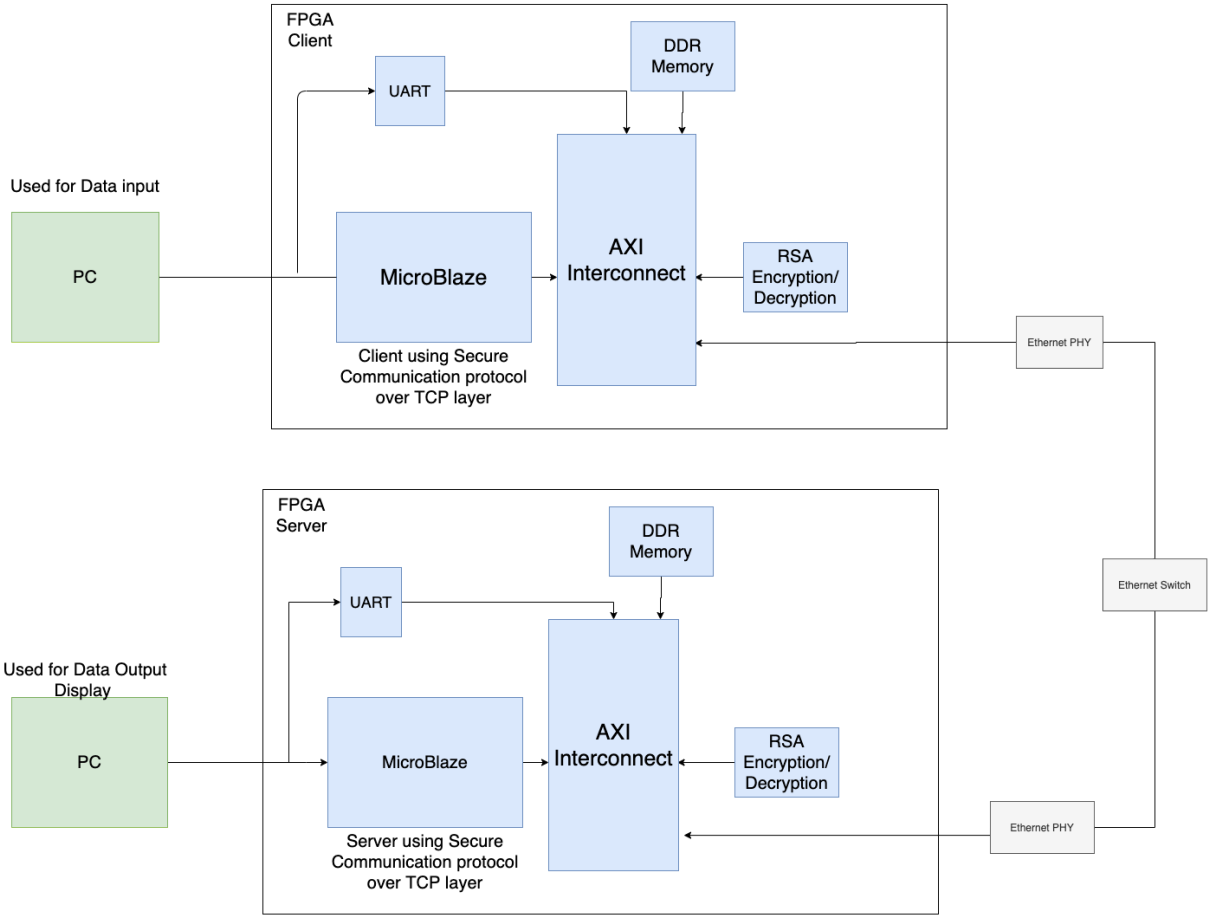


Figure 1. System block diagram of MicroBlaze-RSA system.

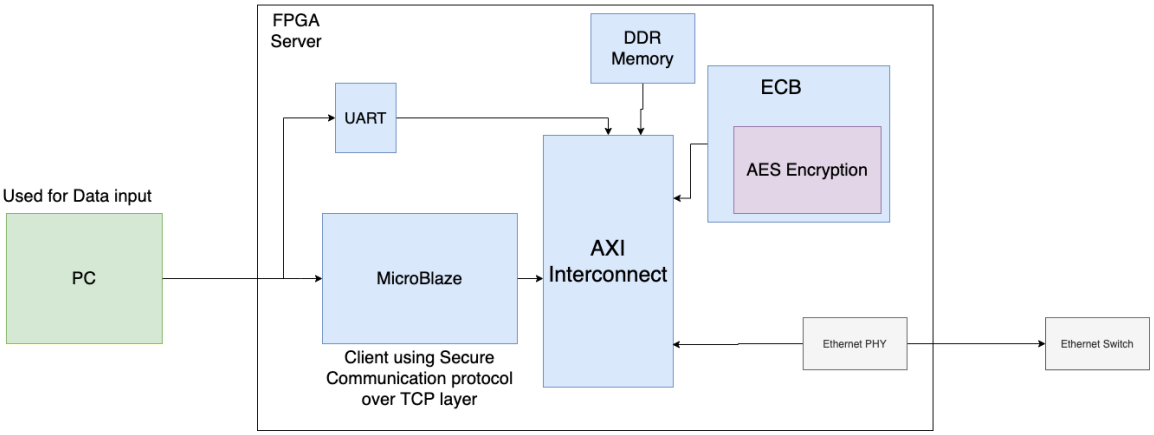


Figure 2. System block diagram of MicroBlaze-AES encryption system.

1.4 Brief Description of IP Blocks

Table 1: Brief Description of IP Blocks Used and Modified

IP Block	Functional Description	Source
RSA Encryption	Encrypts and decrypts the message received using the RSA encryption and decryption algorithms.	Custom
AES ECB Encryption	Encrypts the message received using the Advanced Encryption Standard in Electronic Code Book mode.	Custom
MicroBlaze	A on-board processor used for executing software of the system.	Xilinx
AXI Interconnect	Manage communication between MicroBlaze processor, cryptographic modules, and DDR memory.	Xilinx
UART	Enables communications between PC and FPGA, allowing the users to input command and messages from PC.	Xilinx
Ethernet	Enables FPGA and MicroBlaze systems to communicate with another PC.	Xilinx

2.0 Outcome

Multiple changes were made during the implementation of our design. Most significant changes include system block changes and protocol changes.

2.1 System Block Changes

The original block diagram and the final design block diagrams are shown in Figure 3 and Figure 4.

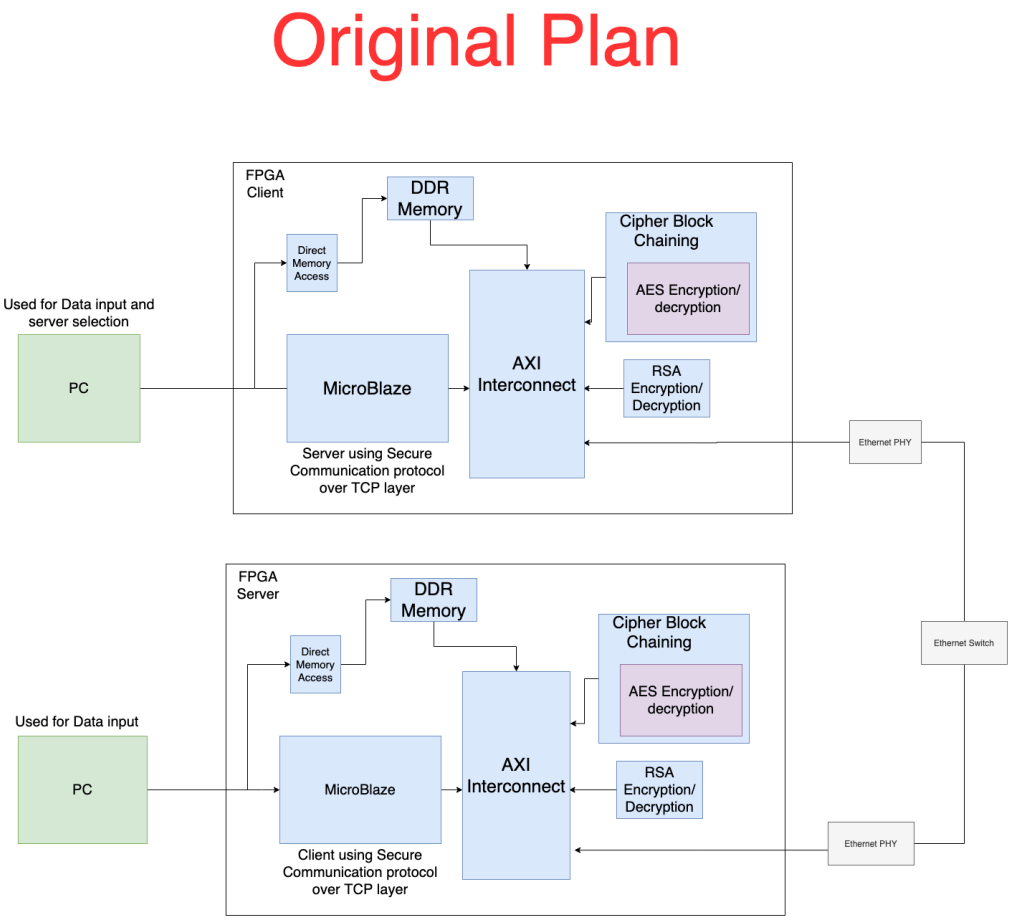
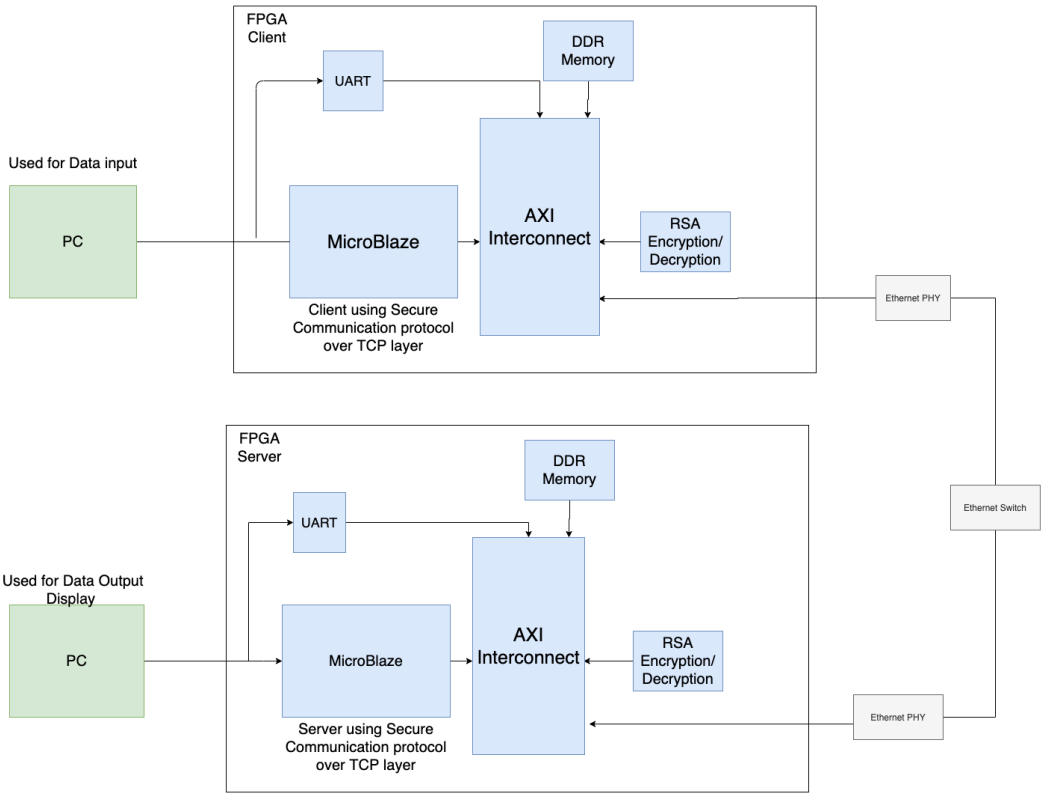


Figure 3. Original planned system diagram.

Final Design - RSA



Final Design - AES

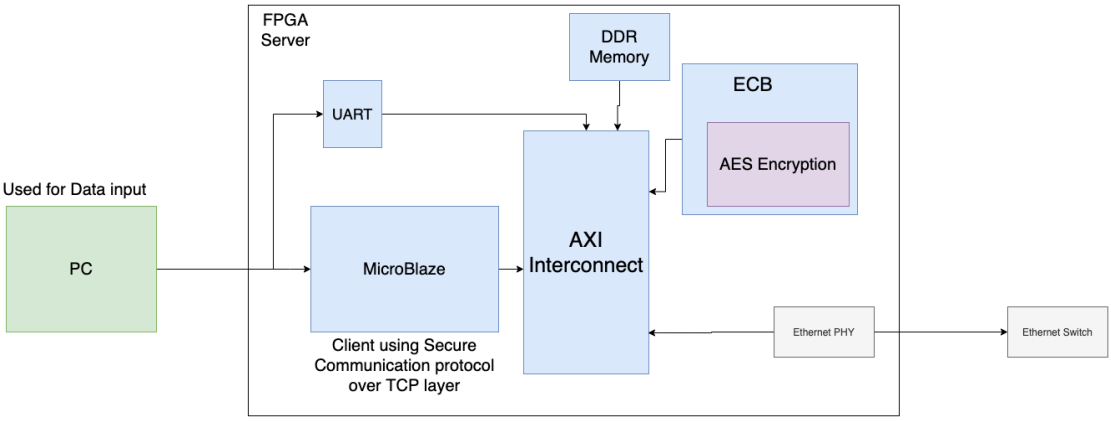


Figure 4. Final design Block diagram.

Some changes are made in the final design. In the original plan, the RSA and AES modules are integrated together into a single system. This is changed in the final design due to the heavy resource utilization of the RSA module. The RSA module uses roughly 95% of available DSP's, which leaves no room for the AES module. Because of this, we were forced to separate the RSA and AES cryptography modules into two discrete systems.

Secondly, in the original plan, we planned to use Cipher Block Chaining (CFB) mode for the AES module, which requires extra hardware or software implementation on top of the AES module. In the final design, we used Electronic Code Book (ECB) mode for the AES module, since this mode simply instantiates multiple AES modules, which is significantly less work than CFB mode. The downside of ECB mode is that it is less secure than CFB mode. This change was made due to time constraints. We did not have adequate time to implement and integrate our originally intended AES module with CFB mode.

2.2 Protocol Changes

Below is the original planned and final implemented communication protocol.

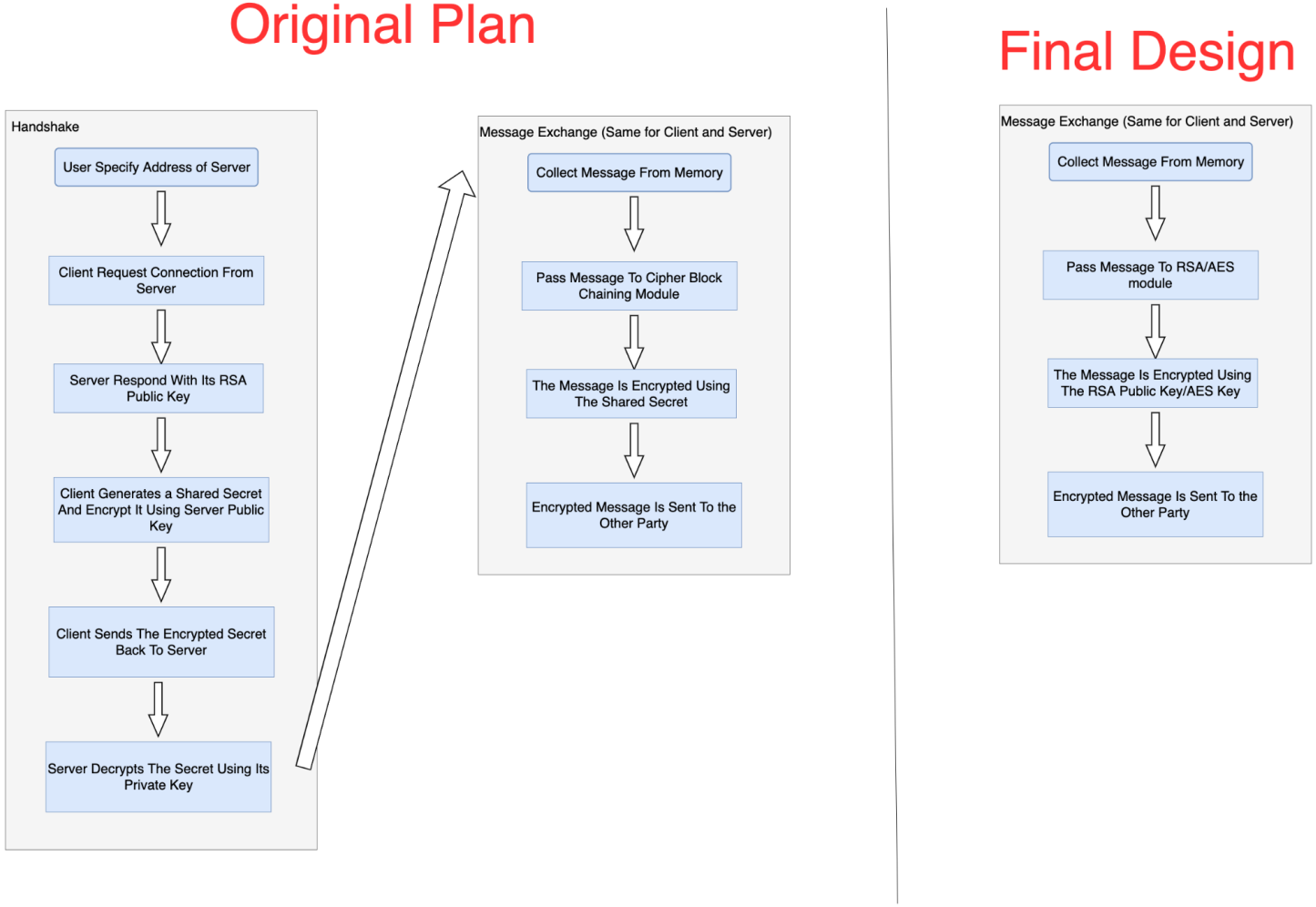


Figure 5. Original protocol vs. final design protocol

Originally, we have designed a secure communication protocol that resembles the TLS protocol, where the two sides of the communication goes through a handshake phase before secure communication. In the handshake phase, the two sides exchange an AES key (shared secret) using RSA encryption/decryption, and the RSA public key of each party is known by the other side. Once the AES key is shared, The two parties communicate using CFB mode AES encryption/decryption with the shared secret key. This extra handshake step is required because AES in CFB mode operates much faster than RSA. Moreover, AES in CFB mode can encrypt/decrypt a message of arbitrary length, where RSA can only encrypt/decrypt a message less than a certain length.

In the final design, the handshake phase is removed, because having the handshake would mean that RSA module and AES module must be both present in the system since RSA is used for handshake and AES is used for message exchange. Because we cannot fit both RSA and AES in the same system in our final design, we have decided to modify the protocol where only the message exchange phase is present. However, we have changed the system so that both the RSA and the AES system can be used for message exchange, with the RSA system having a limit on the message length per exchange.

2.3 Reflection

If there is a chance that we start over, there are several changes that we want to make in our implementation of the project.

1. Before we start implementing anything, we would write the whole system and protocol in software to test the viability of the system. This would have given us a better understanding of where the difficulty may lie in the system and helped us adjust the system accordingly.
2. We would start working on the AES module and CFB mode earlier. During our actual implementation, we started working RSA first. However, RSA proved to be much harder to implement than AES. By the time RSA was implemented, we were behind schedule and did not have enough time for the remainder of the work. During our subsequent implementation of AES, we found out that AES is easier to implement in hardware than RSA. If we had started working on implementing the protocol and AES first, it would have been a smoother design process and probably lead to a more refined final design.
3. We would have moved RSA implementation to purely software. In our original proposed protocol, RSA encryption/decryption was only used during the AES key exchange phase (handshake phase), and subsequent message exchanges are all done by AES. This means that efficient AES implementation is the key to the throughput of the system, and that the efficiency of RSA would not affect the system much. Since a software implementation of RSA is easier than a hardware one, moving RSA to purely software would save a lot of design time.

3.0 Project Schedule

2021/02/02 —— Milestone 1	
Plan: Setup Server and Client Communication using Plain text over TCP connection.	**Progress:** Setup server/client communication over Ethernetlite using lwip project
Challenges: None	

2021/02/09 —— Milestone 2	
Plan: Show testbench with RSA encryption and decryption module functioning.	**Progress:** Finished OS2IP Verilog module. Researched efficient hardware implementation method for modular exponentiation. Completed most of the wrapper for RSA encryption and decryption module.
Challenges: Difficulty in implementing modular exponentiation in hardware and did a lot of research on this. Danlu's laptop experienced technical difficulties which prevented her from using the laptop.	

2021/02/23 —— Milestone 3	
Plan: Show functioning AES encryption and decryption algorithm on testbench. Show functioning AES CBC mode implementation on testbench.	**Progress:** Completed, simulated, and tested I2OP and OS2IP module. Finished, simulated, and used VIO to test RSA modular multiplication module. Completed and simulated RSA modular exponentiation module. Finished SubByte module and ShiftRow module for AES encryption block.
Challenges: Synthesis of the modular exponentiation module failed several times due to design being too large. Team decided to downsize the algorithm. Modular exponentiation module needed to be rewritten due to timing errors after testing in hardware.	

2021/03/09 —— Milestone 4	
Plan: Implement protocol logic in software on MicroBlaze. Show that the data can be transferred from PC to FPGA through Direct Memory Access.	**Progress:** Finished implementation and testing of RSA modular exponentiation module in hardware. Completed and simulated KeyExpansion module for AES encryption. Finished RSA encryption and decryption modules in hardware.
Challenges: Could not get the modular exponentiation module to operate properly at 100MHz. Many attempts were made to optimize the module with little effect. Module operates at 10MHz. RSA module consumed a lot of resources; could not reduce further.	

2021/03/016 —— Milestone 5	
Plan: Implement RSA and AES modules on hardware and connect them to MicroBlaze through AXI interface.	**Progress:** Added RSA module to MicroBlaze embedded system. Completed AES MixColumn module. AES KeyExpansion error debugged and implemented in hardware.
Challenges: Unable to pipeline and optimize the AES algorithm very much.	

2021/03/23 —— Milestone 6	
Plan: Testing of the whole system / cushion for any delayed task.	**Progress:** Verified that MicroBlaze-RSA system could act as server and encrypt/decrypt messages. Finished AES top module and tested in both simulation and in hardware.
Challenges: None	

2021/03/31 —— Final Demo	
Plan: Prepare for demo	**Progress:** Created MicroBlaze-RSA client. Debugged and integrated AES module into MicroBlaze-AES system.
Challenges: Client hosted on FPGA cannot connect to FPGA server in testing.	

Milestone 1 was set as a week to have all members confirm that they have the proper lwip server and client codes and can communicate over the network connection. This week was also set aside so that we could do more research on the implementation of cryptographic algorithms.

Milestone 2 was where the first big delay happened in the project. Modular exponentiation was harder to implement in hardware than we expected, so additional time had to be taken to research various implementation methods and evaluate them. This delay meant that the RSA encryption and decryption module could not be fully constructed and simulated in time. Danlu's laptop also experienced technical issues that severely hampered her ability to use it for almost a week, which greatly impacted progress. This delay impacted all future milestones.

By Milestone 3, the team has finished simulating the RSA module and implementing it in hardware, which fulfilled the original plans for Milestone 2. But the team encountered additional challenges as listed above, which required more debugging and modifications. This endeavour took up a fair bit of our bandwidth, so only 2 submodules for the AES algorithm were completed at this stage.

The plans for Milestone 4 were outdated at this point in time. During the proposal and proposal presentation stage, we received feedback that DMA to FPGA communication is not really an option with the device setup, so it was removed from the project scope. This week, the RSA module was finished, but further changes needed to be made to the project block diagram since we discovered that our implementation of the RSA module consumed 95% of DSP resources on the boards and thus left very little for the AES module to be placed on the same board.

The plan for Milestone 5 was partially met as the RSA module was successfully connected to the MicroBlaze system and all AES modules were implemented in hardware. Milestone 6 was used as cushion time in case of delays, which became very useful. By Milestone 6, the MicroBlaze-RSA system was confirmed to be working and able to encrypt and decrypt messages with network communication. The AES algorithm was also implemented in hardware at this point. The week before the demo was used to verify the MicroBlaze-RSA module further and to

debug client connection issues. Unfortunately, that issue could not be resolved in time for the demo.

Overall, the delay from Milestone 2 impacted the rest of the project and despite best efforts the team was unable to fully make up for the delay. Shifting project structure based on the results of hardware implementation, time constraints, and the problems experienced also rendered the original project schedule obsolete by the mid-project demo, so the disparity between the project schedule planned is not insignificant by the end. We also concentrated resources onto a single module, namely RSA, for longer than ideal, which led to other parts of the project being delayed; more parallelism should have been used. However, the team did manage to make up for the delay on the RSA module and successfully integrated it into a MicroBlaze system.

4.0 Description of Blocks

4.1 Algorithm

4.1.1 RSA

RSA is an asymmetric encryption algorithm which embeds several components in the private and public key. In the private key, one can find the private exponent and modulus. In the public key, one can find the public exponent and modulus. The encryption and decryption process involves doing a mathematical operation:

$$(m^e)\%n$$

Where *m* is the message to be encrypted/decrypted, *e* is the public exponent/private exponent, and *n* is the modulus. Given we implemented a 256 bit RSA, this means that both the private exponent and modulus are 256 bits. This makes the modular exponentiation not trivial. We have decided to implement this operation using the "Square and Multiply" algorithm, which makes use of the "Modular Multiplication" algorithm; they are both described below.

4.1.1.1 Square and Multiply

The pseudo code to achieve modular exponentiation is shown below.

```
Initialize result = m
For each bit in e from left to right:
        Result = Result^2 % n
        If current bit of e is 1:
                Result = (result*m) % n
```

Figure 6. Pseudocode for modular exponentiation

This algorithm guarantees that the computation would be reasonably fast when e and n are both large.

This algorithm needs to perform two modular multiplications, namely Result^2%n and (Result*m)%n. How we achieved this is described below.

4.1.1.2 Modular Multiplication

This calculation is split to 2 parts. The first part is multiplication, which could be easily implemented using '*' operator in Verilog. We tried different ways to optimize this part. However, we could not meaningfully improve Fmax. The second part is modulation. Verilog does not support divisors other than powers of 2. Thus, we wrote our own divider using sequential logic. In the end, we successfully implemented the functionality of this module and managed to fit this module onto the board.

4.1.2 AES

The AES algorithm has 4 major operations: Byte Substitution, Shift Rows, Mix Column and Add Round Key. The algorithm will run those four steps for different numbers of rounds. For each round, we use the Key Expansion module to calculate the key to add in this round. All 5 components will be described below and additional details can be found by consulting the Advanced Encryption Standard (FIPS PUB 197). For our purpose, we choose to use a 128-bit data block with 128-bit key. In this case, we need to run those steps for 10 rounds. But before we run our data with those modules. We need to rearrange the 128-bit data to a 4-by-4 array, where each element is a byte. The pseudo code is shown in Figure 7 below.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])

   for round = 1 step 1 to Nr-1
      SubBytes(state)
      ShiftRows(state)
      MixColumns(state)
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

Figure 7. Pseudocode for AES encryption. Nb is the number of bytes, which is 4. Nr is the number of rounds, which is 10.

4.1.2.1 Byte Substitution

In Byte Substitution, each byte from the array was individually processed and its multiplicative inverse in the finite field GF(2^8) was found. The byte is then replaced by the corresponding value in a table. When implementing this module, we miss the first step of the calculation and use the matrix multiplication described below:

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

We did not find a good way to implement the calculation to find the multiplicative inverse in the finite field. Thus, we manually implemented a function that maps each input byte to an output byte as shown below in Figure 8.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 8. S-Box: values used to map input bytes to output bytes used in Byte Substitution.

4.1.2.2 Shift Rows

In the Shift Row module, all we need to do is to shift each row by element described in the specification document. Figure 9 shows the exact operation and how each byte is shifted. For the 128-bit key and 128-bit data, we need to shift 0 byte for row 1, 1 byte for row 2, 2 bytes by row 3 and 3 bytes by row 4.

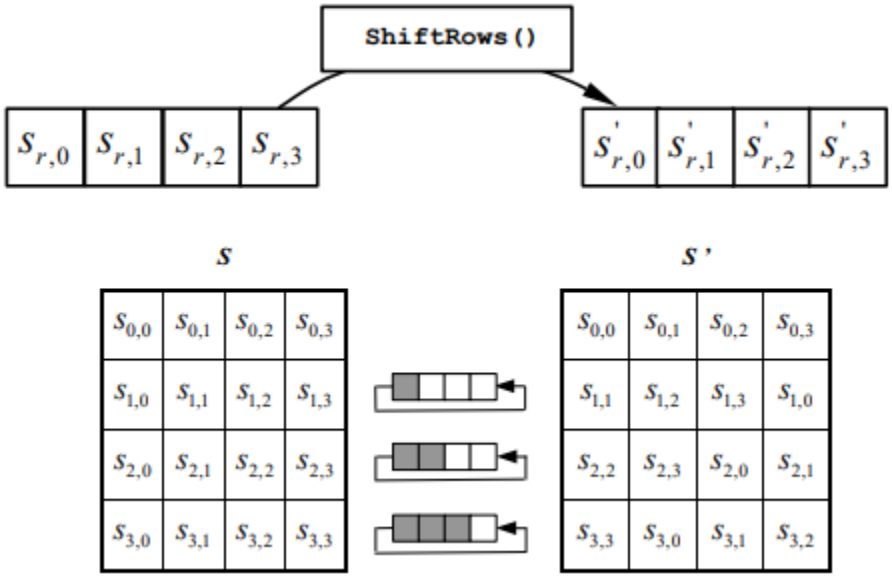


Figure 9. Illustration of shift row operation.

4.1.2.3 Mix Column

In the previous step, we perform a rotation row wise. In this step, we will perform a matrix multiplication column wise. The algorithm is shown below.

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).$$

One thing to note: the dot symbol here corresponds to a special type of multiplication. To multiply by 2, we need to check the MSB of input. If it's 1, then after we multiply by 2, we need to xor 11011 with the result. To multiply by 3, we need to first multiply by 2 then xor the result with it's own.

4.1.2.4 Add Round Key

In the Add Round Key module, we take the 128-bit data from the previous step and a 128-bit key. The value of the key is different for each round and it will be produced by the Key Expansion module. The output is generated by XOR-ing the data and key.

4.1.2.5 Key Expansion

The AES Key Expansion module takes a 128-bit key and expands it into 11 different 128-bit round keys using the pseudocode shown in Figure 10. These round keys are then used in the Add Round Key operation. The first 4 words of Key Expansion is the original key. All following words are the result of an XOR operation of the preceding word and the word that appeared 4 positions earlier. If the index of the word is a multiple of 4, the input word first undergoes cyclic permutation from [a0,a1,a2,a3] to [a1,a2,a3,a0] in the RotWord module; then Byte Substitution (SubWord) operation is performed on this modified word. Afterwards, the output of the SubWord operation is XOR'd with the round constant, a value obtained by the concatenation $[r_i,\{00\},\{00\},\{00\}]$, where r_i is mapped according to Table 2.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end
```

Figure 10. Pseudocode for Key Expansion module

Table 2: Round Constants Table

Round	1	2	3	4	5	6	7	8	9	10
$\mathbf{R_i}$	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80	0x1b	0x36

4.2 Hardware Blocks

4.2.1 Modular Exponentiation

This is a new IP developed by the team. This IP uses the algorithm described in section 4.1.1 to achieve the main mathematical operation of RSA. This block is controlled by the Microblaze processor, and is connected to the Microblaze System via GPIOs.

4.2.2 AES Encryption

This is a new IP block developed by the team. This IP uses the algorithm described in section 4.1.2. Because the resource usage is low for a single round, we implemented all 10 rounds in combinational logic. It is connected to the Microblaze System via AXI_GPIOs. We use a busy wait loop for the processor to wait for the output is ready.

4.2.3 Microblaze

Version 11.0.
This is the embedded processor soft core provided by Vivado. It runs the C code implementing network communication protocol and the program needed to operate the Modular Exponentiation and the AES module.

4.2.4 EthernetLite

Version 3.0.
This is a Vivado provided module that provides ethernet interfaces for the Microblaze processor. It provides 100 Mb/s interface in our project so that the design can achieve network communication.

4.2.5 Memory Interface Generator (MIG)

Version 4.2.
This is a Vivado provided module that generates the interface to access the on board DDR memory. This allows Microblaze to run programs and store data in DDR memory. This module is needed as the software for the system is too large to fit on the on-die BRAM.

4.2.6 GPIO

Version 2.0.
This is a Vivado provided module that provides a general purpose I/O that connects the AXI interface to our own cryptographic modules (Modular Exponentiation module and AES module). The Microblaze processor accesses the cryptographic module through GPIO’s.

4.2.7 AXI UARTLite

Version 2.0.
This is a Vivado provided module that provides communication between the FPGA and PC through AXI interface.

4.2.8 Clocking Wizard

Version 6.0.
This is a Vivado provided module. It is capable of providing different clock frequencies from the 100MHz system clock. This module is used to provide different frequencies to different parts of the system, for example, the RSA module required a 10 MHz clock, which is different from the rest of the system.

5.0 Design Tree

In our github repository, we only uploaded the necessary source files instead of the entire vivado project.

SRC:

- ★ **RSA**:
  - **I2OSP**: contains the I2OSP module Verilog file and its testbench.
  - **OS2IP**: contains the OS2IP module Verilog file and its testbench.
  - **decrypt**: contains the RSA decryption Verilog file and its testbench.
  - **encrypt**: contains the RSA encryption module Verilog file and its testbench.
  - **example_keys**: keys we used to test our RSA encryption and decryption result.
  - **mod_exp**: contains the modular exponentiation module Verilog file and its testbench.
  - **Module_and_description.txt**: the RSA algorithm in python and text description
  - **openssl.txt**: the command to run openssl for result verification purposes.
- ★ **AES:**
  - **keyExpansion**: contains key expansion module Verilog files and testbenches
  - **mixColumn**: contains mix column module Verilog files and testbenches
  - **shiftRow**: contains shift row module Verilog files and testbenches
  - **subbyte**: contains byte substitution Verilog files and testbenches
  - **AES.v**: AES top module
- ★ **sw:**
  - **RSA_server_echo.c**: the echo server for RSA algorithm with function of encryption and decription.
  - **RSA_client_main.c**: the client that asks for user input and encrypts the input before sending it to the server. The received response from the server is then decrypted and displayed.
  - **AES_server_echo.c**: the echo server that receives and encrypts incoming messages, then prints it out.
- ★ **Rsa_server_client:**
  - This folder contains the entire RSA system as a Vivado project, which includes all the software projects for both server and client.
- ★ **Aes_server_client:**
  - This folder contains the entire AES system as a Vivado project, which includes all the software projects for both server and client.

DOCS: This folder contains the design documents of our design.

6.0 Tips and Tricks

Here are some tips and tricks that we have for future students:

- Calculation takes time, try to find some algorithm can be done in simple logic.
- When implementing a hardware block, always test out the algorithm in software first. This saves a lot of failed trial and error when the algorithm is wrong.
- Try to begin work and exploration of all individual parts of the project as early as possible.
- Do not attempt to create too many separate custom IP blocks and try to use available IP blocks as much as possible to reduce workload.

7.0 References

[1] Rahman, M., Rokon, I. and Rahman, M., 2009. Efficient hardware implementation of RSA cryptography. *2009 3rd International Conference on Anti-counterfeiting, Security, and Identification in Communication*,.

[4] K. Kumar, K. Ramkumar, A. Kaur and S. Choudhary, "A Survey on Hardware Implementation of Cryptographic Algorithms Using Field Programmable Gate Array", 2020 IEEE 9th International Conference on Communication Systems and Network Technologies (CSNT), 2020. Available: 10.1109/csnt48778.2020.9115742 [Accessed 31 January 2021].

[3] Y. Zhu, H. Zhang and Y. Bao, "Study of the AES Realization Method on the Reconfigurable Hardware", 2013 International Conference on Computer Sciences and Applications, 2013. Available: 10.1109/csa.2013.23 [Accessed 31 January 2021].

[4] Dworkin, M. , Barker, E. , Nechvatal, J. , Foti, J. , Bassham, L. , Roback, E. and Dray, J. (2001), Advanced Encryption Standard (AES), Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, [online], https://doi.org/10.6028/NIST.FIPS.197 (Accessed April 14, 2021)