RSA Encryption Hardware & Software Implementation on Pynq Board

Noah Evantash
April 26th, 2023

EECE4632 HW-SW Design FPGA, Prof. Leeser

Introduction:

The core of this project is the RSA algorithm, named after its inventors Ron Rivest, Adi Shamir, and Leonard Adleman in 1978 [1]. RSA is one of the earliest and most prominent encryptions algorithms that is still widely used within the industry for encrypted communications and data security. The popularity of RSA is credited to the algorithms' security and inability to be decrypted through iterative attempts. RSA is so secure because it uses a complex formula with a pair of large prime numbers that make it difficult to decrypt the value [2]. As a result, it would take a traditional computer billions of years to attempt to crack RSA since large primes require intensive and excessive computations. Only recently has RSA security been considered at risk due to the current advancements in Quantum computing [3, 4]. As a result, security standards are suggesting industries update their RSA implementations to larger bit size key values by 2030 to ensure ample security [5].

Regardless of the future of RSA, the encryption method is still one of the most widely used security algorithms. It has been applied to a variety of different applications since its conception and is critical in securing sensitive data being sent over unsecure networks. Commonly used tools, such as emails, websites, and database repositories, all have some sort of encryption system to ensure your digital information will not be intercepted. As a result, industries require improved methods of securing data to rapidly encrypt and decrypt transmitted data. One such method is to implement dedicated hardware that performs the encryption continuously. Cryptosystems are therefore implemented on FPGAs, with an array of accessible packages, libraries, and accelerators to develop these systems [6]. To expand further into this topic, an RSA algorithm will be developed and implemented on a Pynq Z2 board using a Zynq-7000 SoC FPGA.

RSA Overview:

The RSA algorithm works by using a pair of prime numbers to encrypt and decrypt input data. There are two sides to this algorithm: the encryption which uses the public key, and the decryption which uses the private key. For the encryption, a message value, "*M*", is inputted into the arithmetic function below:

$$C = M^{e} mod(n)$$

The returned cipher, "*C*", is an encrypted value equal to the modular exponent of the message and the public key (*e, n*). The public key is calculated through evaluating the input pair of prime numbers, *p* and *q*. The public key, *n* is equal to the product of the prime numbers *p* and *q*, while *e* is equal to a value coprime and less than the totient (*t*) value that is the product of (*p*-1) * (*q*-1). The primary inputs to this encryptor are therefore *p* and *q* for the prime numbers, *e* for the public key, and most importantly the data to be encrypted, *M*.

On the other side, the decryption is performed by evaluating the cipher, "*C*" and the private key (*d, n*) in the equation:

$$M = C^d mod(n)$$

The only new variable, "*d"*, represents the private key, while *n* remains the same as the public key. The value *n* needs to be the same in the encryption and decryption for this to work since *p* and *q* need to be the same. The value *d* is calculated from the public key, *e*, using the identity:

$$1 \equiv ed \; mod \; (t)$$

Where *e* is the public key and *t* is the totient. To evaluate for *d,* the modular inverse, e^{-1}mod(t), is calculated. The only needed inputs to find *d* are *e* and *t*, which are both calculated from the prime numbers when generating the public key.

In all, there are 4 iterations of functions occurring, with 2 functions operating to encrypt, and 2 functions operating to decrypt. The 2 encryption functions are the generating public key function and performing the RSA encryption function. The public key function has the prime numbers as inputs, while the RSA encryption function has the public key (*e, n)* and the message, *M*, as inputs. On the other side, the 2 decryption functions are the private key generator and the RSA decryption. The private key function requires the prime number inputs as well as the value of the public key *e*. The decryption then inputs the private key's output (*d, n*) and the encrypted cipher *C* to perform the decryption. The result of running these four functions should be the original message sent, with an encrypted value being returned in the middle. To visualize this process, there is a flow chart depicting the process shown in figure 1.

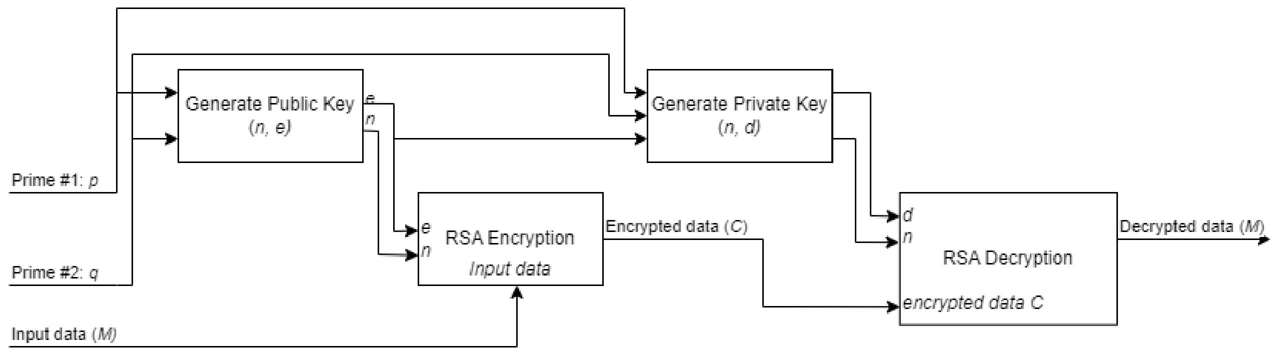


Figure 1: Flow Chart of RSA Algorithm

PS Implementation:

The first iteration of creating the RSA algorithm was built out using Python on the Processing system for the Pynq board. This iteration was simple and borrowed heavily from the source code available for introducing the RSA algorithm [2, 7]. It began with evaluating for the public key using the greatest common denominator function to evaluate the totient and prime numbers to get the key. A value is then encrypted in the function "RSA_Encryptor" by using the convenient "pow(a, e, m)" function on Python which evaluates "a^e mod (m)". The returned value then becomes the encrypted cipher.

The next part of the code uses the inverted modular function to evaluate the private key given the public key and primes. The inverted modulo function is evaluated using the function “pow(pubkey, -1, phi)” where pubkey is the public key and phi is the totient. With the private key, the decryption of the cipher is performed by evaluating the same equation as the encryption, except with the private key and cipher. The returned value becomes the decrypted cipher and should be the original value. However, due to the limitations of the integer value in Python, this iteration was unable to encrypt large values due to the key and cipher data sizes. The cause of this issue was the integer value would be converted to a float when evaluating the large exponential and would therefore be inaccurate when evaluating the modulo. This can be seen in figure 6 in the verification and validation section, where the value 5000 is too large to be encrypted/decrypted.

The next iteration designed used large integer value for the prime values by generating random prime numbers with a certain bit’s size. To do this, an implementation of the Rabin Miller algorithm was applied to generate the large prime values [8]. However, because of implementing the prime generation randomly, the delay of this iteration was significant at roughly 8 seconds. On the other hand, this iteration was more capable of larger values being encrypted/decrypted. This allowed it to encrypt strings rather than small integer values.

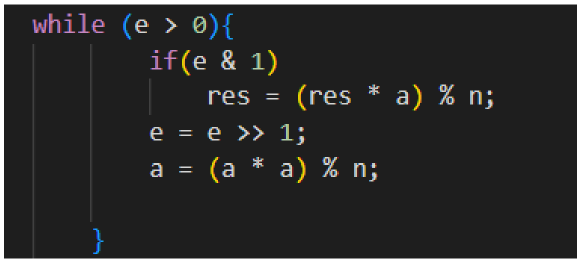
In the final software iteration, the functionality of encrypting a string was kept, but the random generation of the string was altered. Instead, the string was broken down into an array of the ASCII values of the characters, where each value is encrypted in the same manner. As a result, the input message size was limited, ensuring overflow wouldn’t occur and the resulting encrypted values could be decrypted.

The last implementation on the PS was considered the best since it was capable of handling string values without requiring an excessive delay time to compute. It was decided to build upon this iteration to design the PL implementation since it would pair well with an AXI stream interface.

PL Implementation:

For the PL implementation, the first step was to build the encrypting code on Vitis HLS and test the code with a testbench. The first implementation omitted generating the keys like the Python code, and instead opted to use static values for the prime numbers and keys. The values used for the keys were verified with the PS implementation, and furthermore allowed verification with the resulting encrypted values.

Unlike the Python implementation, the HLS code did not have a convenient method of evaluating the modular exponent. The right-to-left binary method of evaluating modular exponentials was the best choice to use since it has reduced memory footprint and operations [9]. Figure 2 shows the code implemented to compute the value of the encrypted result.

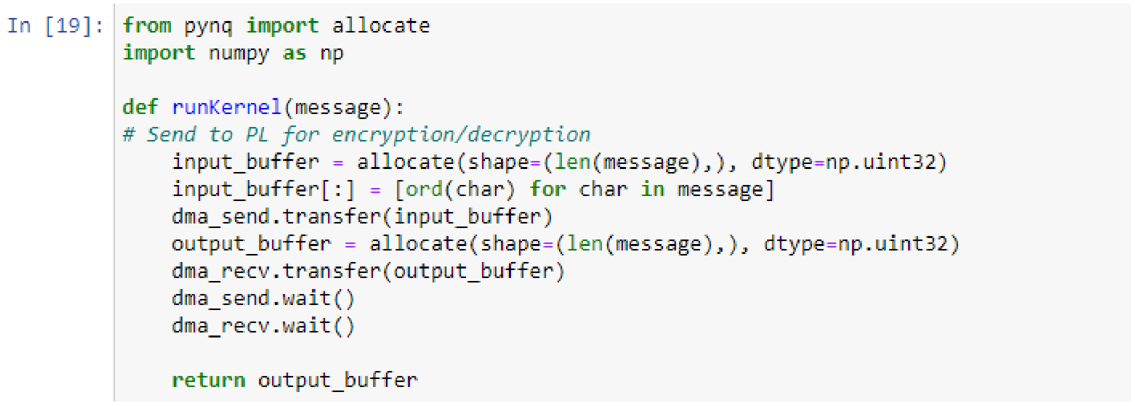


```
while (e > 0){
        if(e & 1)
            res = (res * a) % n;
        e = e >> 1;
        a = (a * a) % n;

    }
```

Figure 2: Right-to-left binary evaluation of modular

To implement AXI stream to encrypt an array of values, the while loop in figure 2 is embedded in another while loop that reads the data from the stream. The top while loop operates until the value the last value is sent. By evaluating the input AXI Stream array this way, each value in the array is returned as its own individually encrypted value, much like the 3rd PS implementation. To verify the results, the input array and selected keys for the 3rd PS implementation were assessed on the hardware through a test bench. As a result of selecting the same keys and values, the returned encrypted values from the testbench should match the values of the PS encryption. The comparison of the encrypted arrays is shown in the validation and verification section under figure 8 for the PS, and figure 9 for the PL.

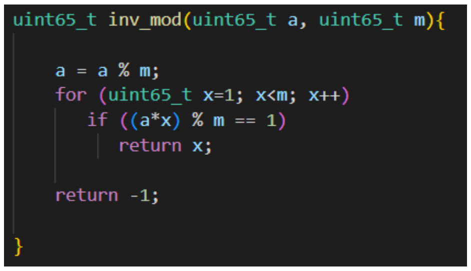
With the values confirmed on the testbench, the hardware implementation was validated and ready for the overlay. The block diagram for the overlay shown in figure 5 was generated and tested on a notebook. The overlay utilized the transformation of the input string on the PS, where the string is converted to an array of ASCII integers and set to the input buffer for the stream. To do this, the function runKernel(), shown in figure 3, has an input for the string message, where it allocates the input buffer and output buffer to the length of the message. The input buffer has each character in the message converted into its ASCII value and inputted as an array. The buffer is then transmitted to the PL through the DMA where the values are encrypted. The output buffer then receives the output as an array of encrypted values.



```
In [19]: from pynq import allocate
         import numpy as np

         def runKernel(message):
         # Send to PL for encryption/decryption
             input_buffer = allocate(shape=(len(message),), dtype=np.uint32)
             input_buffer[:] = [ord(char) for char in message]
             dma_send.transfer(input_buffer)
             output_buffer = allocate(shape=(len(message),), dtype=np.uint32)
             dma_recv.transfer(output_buffer)
             dma_send.wait()
             dma_recv.wait()

             return output_buffer
```

Figure 3: Overlay Code for Setting up and Communicating with the Interface

The next iteration on the PL expanded the RSA implementation by adding the key generator function. To do this, much like the PS key generator, a function was required for the greatest common denominator, the co-prime checker, and the inverse modular function. The GCD function and co-prime checker were simple and used the same logic as the PS. However, much like the exponential modular, the inverse modular was not readily available in hardware. To perform this calculation, the extended Euclidian algorithm was used, as shown in figure 4, and returned the private key value [10].



```
uint65_t inv_mod(uint65_t a, uint65_t m){

    a = a % m;
    for (uint65_t x=1; x<m; x++)
        if ((a*x) % m == 1)
            return x;

    return -1;

}
```

Figure 4: Extended Euclidian Algorithm for Modular Inverse Evaluation

While designing the key generator, a common issue was the data size due to limitations of the board's area. When increasing the size of the keys, all the supporting functions would also reflect the new data size, and consequently increase the amount of LUTs, FFs, and DSPs drastically. As a result, the data sizes were reduced to 32-bits until further optimizations could be evaluated. Due to the decreased data size, the RSA encryption would be considered less secure than the previous example that were tested with 64- & 128-bits keys.

With the key generation functions in place and able to synthesize, the next step was to validate its functionality with a test bench. The test bench first tested the generation of keys by inputting the prime values used prior on the PS and confirming that the correct public and private keys were returned. From there, the encryption function was tested to ensure the output cipher was the same as the PS results.

The interface was then implemented to allow the prime values to be inputted from the PS using AXI lite. Two integer values were added to the input of the encryption function and were assigned as AXI lite. On the PS side interacting with the overlay, the prime values are set by accessing the register map and setting values for the primes, as seen in figure 10. For this implementation to function, the prime numbers must be set before the AXI Stream is sent.

The last addition was very simple and allowed for switching between encryption and decryption on the same overlay through manipulating a Boolean value. Since the modular exponential evaluation is performed for both encryption and decryption, with the only difference being the exponent value, it is simple to implement a switch function. For encryption, the public key is the exponent evaluated, while for decryption, the private key is the exponent. Therefore, using the Boolean value, the exponent value is set to the public key if the Boolean is 0 (encrypt), or it is set to the private key if the Boolean is 1 (decrypt). The Boolean value also uses AXI lite and can be written from the PS through the register map.

The block diagram of this implementation is shown in figure 5. Note that the additional inputs for the prime numbers and Boolean did not change the block diagram since they operated on AXI lite.

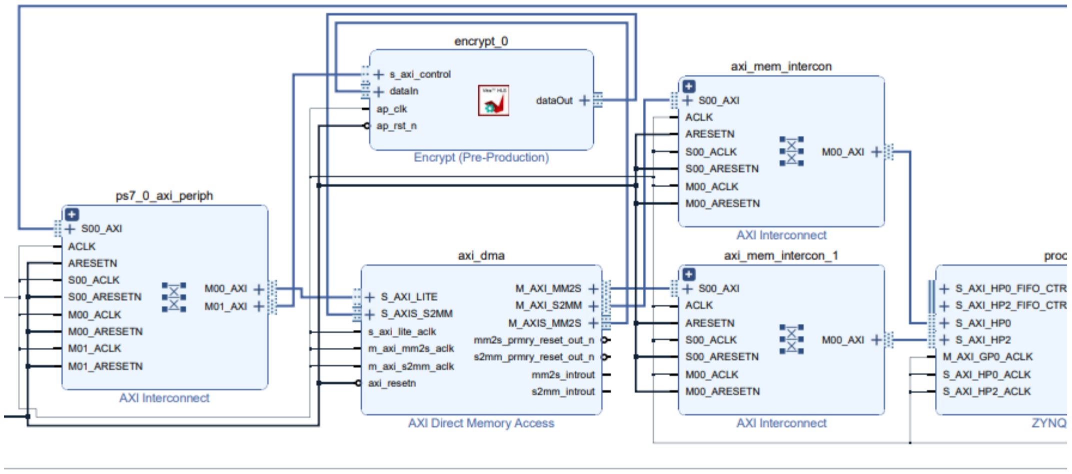


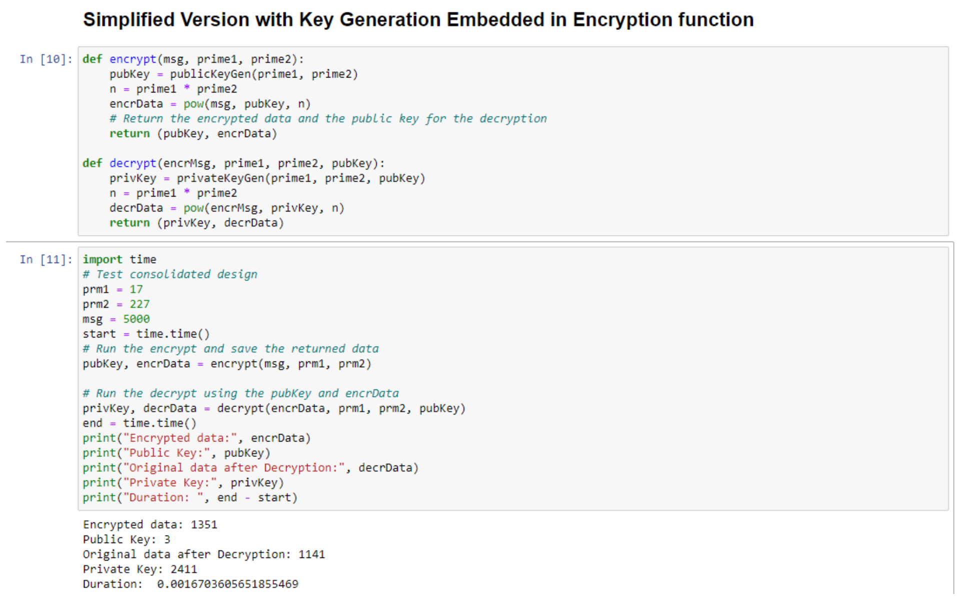
Figure 5: RSA Encryption Block Diagram

With the encryption and key generation, all the necessary iterations to perform the RSA as seen in figure 1 are accounted for. Additionally, almost all the computation is being performed on the PL apart from the string transformation. Moving forward, the next steps are to validate the functionality and explore potential optimizations to improve the speed.

Data Validation and Verification:

To operate and validate each PS and PL function and code, all relevant files are stored in the repository: https://github.com/noahe7700/RSA_Project/tree/main/Project%20Report.

For the PS code, all the iterations of Jupyter notebooks designed are stored in the folder "PS Code". For the first iteration that used an integer value for encrypting with small primes, open and run "PS Notebook #1". The result of running this notebook is shown below in figure 6:



Simplified Version with Key Generation Embedded in Encryption function

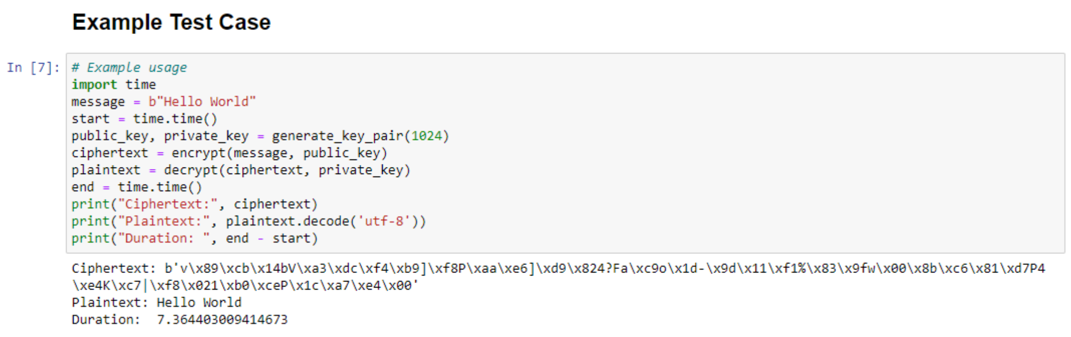
```
In [10]: def encrypt(msg, prime1, prime2):
             pubKey = publicKeyGen(prime1, prime2)
             n = prime1 * prime2
             encrData = pow(msg, pubKey, n)
             # Return the encrypted data and the public key for the decryption
             return (pubKey, encrData)

         def decrypt(encrMsg, prime1, prime2, pubKey):
             privKey = privateKeyGen(prime1, prime2, pubKey)
             n = prime1 * prime2
             decrData = pow(encrMsg, privKey, n)
             return (privKey, decrData)
```

```
In [11]: import time
         # Test consolidated design
         prm1 = 17
         prm2 = 227
         msg = 5000
         start = time.time()
         # Run the encrypt and save the returned data
         pubKey, encrData = encrypt(msg, prm1, prm2)

         # Run the decrypt using the pubKey and encrData
         privKey, decrData = decrypt(encrData, prm1, prm2, pubKey)
         end = time.time()
         print("Encrypted data:", encrData)
         print("Public Key:", pubKey)
         print("Original data after Decryption:", decrData)
         print("Private Key:", privKey)
         print("Duration: ", end - start)

         Encrypted data: 1351
         Public Key: 3
         Original data after Decryption: 1141
         Private Key: 2411
         Duration:  0.0016703605651855469
```

Figure 6: Simple RSA Software Implementation

The input value in this case is 5000 and is encrypted with the prime numbers 17 and 227. The encrypted value becomes 1351 with the public key, and the decrypted value is returned as 1141 with the generated private key. The result shows that the value gets converted to float when being evaluated, and therefore returns an inaccurate decryption. For this case, the encrypted value only decrypts when the provided input message and keys are relatively small values. For example, the encryption is successful when performing the encryption on a value of 500.

For the results of the second attempt, run "PS Notebook #2". This attempt encrypted the value of "Hello World" as a 1024-bit value. The resulting cipher shown in figure 7 is an unrecognizable and highly secure value; however, the timing is significant at 7.364, and can fluctuate greatly.

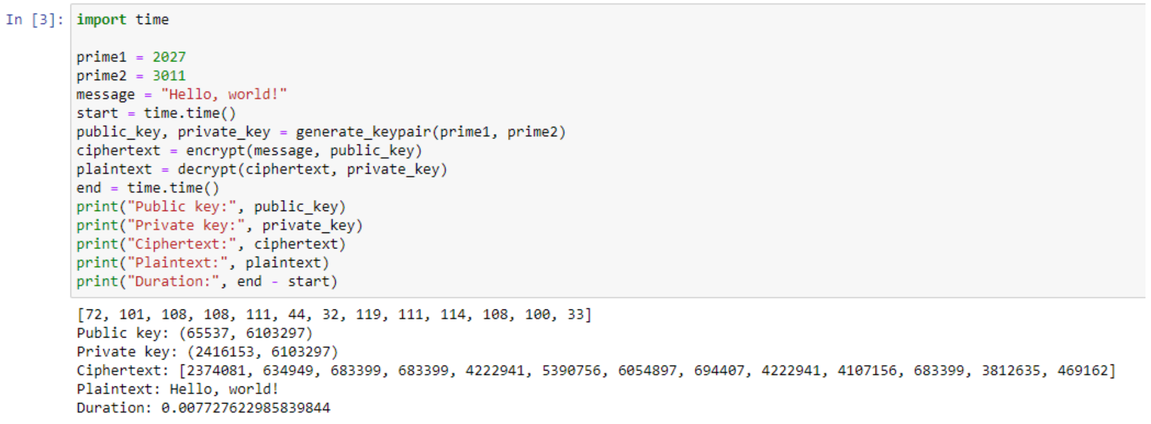


Example Test Case

```
In [7]: # Example usage
        import time
        message = b"Hello World"
        start = time.time()
        public_key, private_key = generate_key_pair(1024)
        ciphertext = encrypt(message, public_key)
        plaintext = decrypt(ciphertext, private_key)
        end = time.time()
        print("Ciphertext:", ciphertext)
        print("Plaintext:", plaintext.decode('utf-8'))
        print("Duration: ", end - start)

        Ciphertext: b'v\x89\xcb\x14bV\xa3\xdc\xf4\xb9]\xf8P\xaa\xe6]\xd9\x824?Fa\xc9o\x1d-\x9d\x11\xf1%\x83\x9fw\x00\x8b\xc6\x81\xd7P4
        \xe4K\xc7|\xf8\x021\xb0\xceP\x1c\xa7\xe4\x00'
        Plaintext: Hello World
        Duration:  7.364403009414673
```

Figure 7: Software RSA Implementation with Random Prime Generation

The third notebook, “PS Notebook #3”, is what the PL is based on and will be used to verify the results of the PL implementation. The message “Hello, World!” is the input used for the encryption and converted into its ASCII values as an array. The results of the encryption are shown below in figure 8.



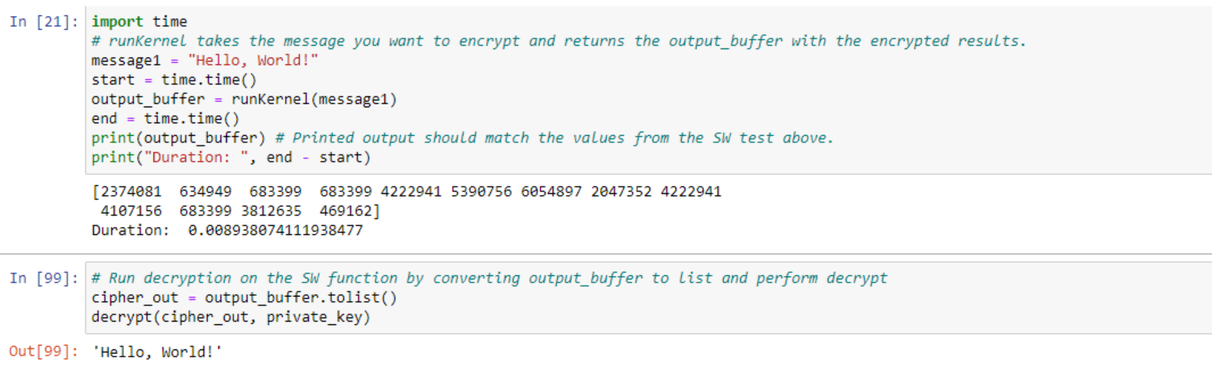
```
In [3]: import time

        prime1 = 2027
        prime2 = 3011
        message = "Hello, world!"
        start = time.time()
        public_key, private_key = generate_keypair(prime1, prime2)
        ciphertext = encrypt(message, public_key)
        plaintext = decrypt(ciphertext, private_key)
        end = time.time()
        print("Public key:", public_key)
        print("Private key:", private_key)
        print("Ciphertext:", ciphertext)
        print("Plaintext:", plaintext)
        print("Duration:", end - start)

        [72, 101, 108, 108, 111, 44, 32, 119, 111, 114, 108, 100, 33]
        Public key: (65537, 6103297)
        Private key: (2416153, 6103297)
        Ciphertext: [2374081, 634949, 683399, 683399, 4222941, 5390756, 6054897, 694407, 4222941, 4107156, 683399, 3812635, 469162]
        Plaintext: Hello, world!
        Duration: 0.007727622985839844
```

Figure 8: Software RSA Implementation using ASCII Array

The message is first converted into its ASCII values, which is printed as the top array in the results. The ASCII array is then encrypted using the public key (65537, 6103297) and produces the encrypted array which is printed as the Ciphertext in the results. The Ciphertext is then decrypted with the private key (2416153, 6103297) to produce the original ASCII array. Lastly, the decrypted ASCII array is converted back into a string and joined to return the original message. The resulting encrypted value of this implementation was recorded and will be used for testing and validating the PL encryption.

To run the hardware code, navigate to the “PL Code” folder and open the “Stream Notebook” file. This notebook contains all the necessary code to run the overlay. Additionally, the overlay files need to be uploaded to the board to test. The overlay files are in the “Simple Overlay” and “Final Overlay” folders, with their source code being RSA1.cpp and RSA2.cpp respectively in the “Source Code” folder.

For validating the first interface, upload the overlay files from the “Simple Overlay” folder to the board and change the text in the notebook to the correct location of the overlay files. Run the first few lines of code and the section Overlay Code. The input string, “Hello, World!” is converted into an ASCII array and then transmitted to the overlay. The encrypted values are then read back from the overlay and are printed below in the results in figure 9.

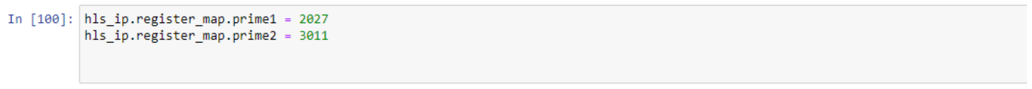


```
In [21]: import time
         # runKernel takes the message you want to encrypt and returns the output_buffer with the encrypted results.
         message1 = "Hello, World!"
         start = time.time()
         output_buffer = runKernel(message1)
         end = time.time()
         print(output_buffer) # Printed output should match the values from the SW test above.
         print("Duration: ", end - start)

         [2374081  634949  683399  683399 4222941 5390756 6054897 2047352 4222941
          4107156  683399 3812635  469162]
         Duration:  0.008938074111938477

In [99]: # Run decryption on the SW function by converting output_buffer to list and perform decrypt
         cipher_out = output_buffer.tolist()
         decrypt(cipher_out, private_key)

Out[99]: 'Hello, World!'
```

Figure 9: Overlay Implementation with static keys

The returned array in figure 9 is validated by the Ciphertext array in figure 8. Furthermore, the resulting array can be decrypted by the PS operation and return the original string, thus confirming the overlay's functionality.

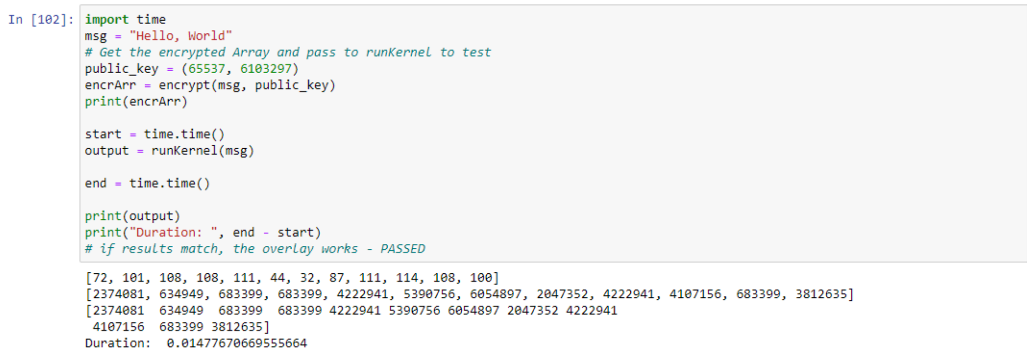
For validating the final implementation where the prime values are input, upload the Final Overlay files to the board and update the location in the notebook. This example has an additional step where the prime values are set in the register map, as shown in figure 10.



```
In [100]: hls_ip.register_map.prime1 = 2027
          hls_ip.register_map.prime2 = 3011
```

Figure 10: Writing to AXI Lite Values for final RSA Implementation

With the prime values set, the PL will now generate the keys and perform the encryption when the input buffer is sent. Additionally, this overlay has two run functions. One run function is used for encryption with the Boolean set to 0, and the input buffer set to the string's ASCII value. The other run function is used for decryption with the Boolean set to 1, and the input buffer set to the previous output buffer result. The decryption function could also reconstruct the string – however this was taken out to isolate the time only from the PL. This overlay was tested with the input "Hello, World" and the prime values from figure 10. The results of running this overlay are shown below in figure 11.



```
In [102]: import time
          msg = "Hello, World"
          # Get the encrypted Array and pass to runKernel to test
          public_key = (65537, 6103297)
          encrArr = encrypt(msg, public_key)
          print(encrArr)

          start = time.time()
          output = runKernel(msg)

          end = time.time()

          print(output)
          print("Duration: ", end - start)
          # if results match, the overlay works - PASSED

          [72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100]
          [2374081, 634949, 683399, 683399, 4222941, 5390756, 6054897, 2047352, 4222941, 4107156, 683399, 3812635]
          [2374081  634949  683399  683399 4222941 5390756 6054897 2047352 4222941
           4107156  683399 3812635]
          Duration:  0.01477670669555664
```

Figure 11: Overlay Implementation with Key Generation Results

The results shown in figure 11 are three arrays and the duration. The first array is the original ASCII array from the message. The second array is the encrypted array that was performed with the PS, while the third array is the encrypted array performed by the PL. Note that both the encrypted arrays match, and therefore confirm the function of this overlay. It is also important to note the duration, with this iteration having a slightly longer delay than the previous example in figure 9. This is a result of the key generation, which performs additional computations and increases the latency. Regardless, the results of testing the overlay all indicate that this implementation succeeded at RSA encryption, with the only remaining work to optimize the design.

Optimization:

When optimizing, it was discovered that the synthesis had trouble estimating the latency of the RSA implementation with the key generators. This was a result of the while loop being dependent on the size of the private or public key, which was now dynamic due to the key generator. Consequently, the synthesis could not calculate the number of cycles or the latency. It additionally caused issues with scheduling and unrolling since the size of the loop was unknown when synthesizing. To accommodate these challenges, the first iteration of the RSA algorithm, called RSA Simple, was used since its loops were dependent on static values. With static values, different optimizations were tested to see if latency speed up occurs before applying any optimizations to the final implementation.

The different designs tested primarily focused on pipelining optimizations of the while loop used to encrypt/decrypt the values. The results of the different optimizations tested are shown in table 1.

Design	**Latency**	**Area**	DSP	FF	LUTs
RSA_Simple - no pipeline	32120	3583	4	3183	2833
RSA_Simple - while(1) pipelined*	NA	160747	31	157647	117491
RSA_Simple - pipelined at 64 loops*	NA	158853	31	155753	119808
RSA_Simple - Unroll 2	32120	3583	4	3183	2833
RSA_Simple - Unroll 8	32120	3583	4	3183	2833
RSA_Simple - while loop pipeline	26000	12126	3	11826	9389
RSA_Simple - while pipeline @ 10ns	13440	12394	3	12094	9427
RSA_Simple - 32-bit no pipeline	24420	2551	3	2251	1977
RSA_Simple - 32-bit pipeline and unroll 2	13440	23976	4	23576	18651
RSA_Simple - 32-bit pipeline @ 9ns II=90	11100	12606	3	12306	9476
RSA_Simple - 32-bit no pipeline with 8ns	20420	2743	3	2443	2044
RSA_Simple - Pipeline loop 16, II = 61 @ 10ns	10390	12382	3	12082	9424

*Table 1: Optimization Design Results - *exceeded board area limit.*

The results show that there is moderate speed-up when pipelining, changing the clock speed, or modifying the data size. However, the pipelining needs to be done around the encryption while loop since the stream loop cannot be pipelined. This is because it is continuously looped until the last value is read – and therefore the total schedule would be unknown. Additionally, the result of some optimizations had large areas that exceeded the size limit of the pynq board.

Another frivolous result was the partial unroll values, which had no effect on the latency or area without pipelining. Partial unroll optimizations on the pipelined loop also provided little to no benefit, with only an increase in area. The other resulting optimizations were graphed on a pareto graph to compare their results.

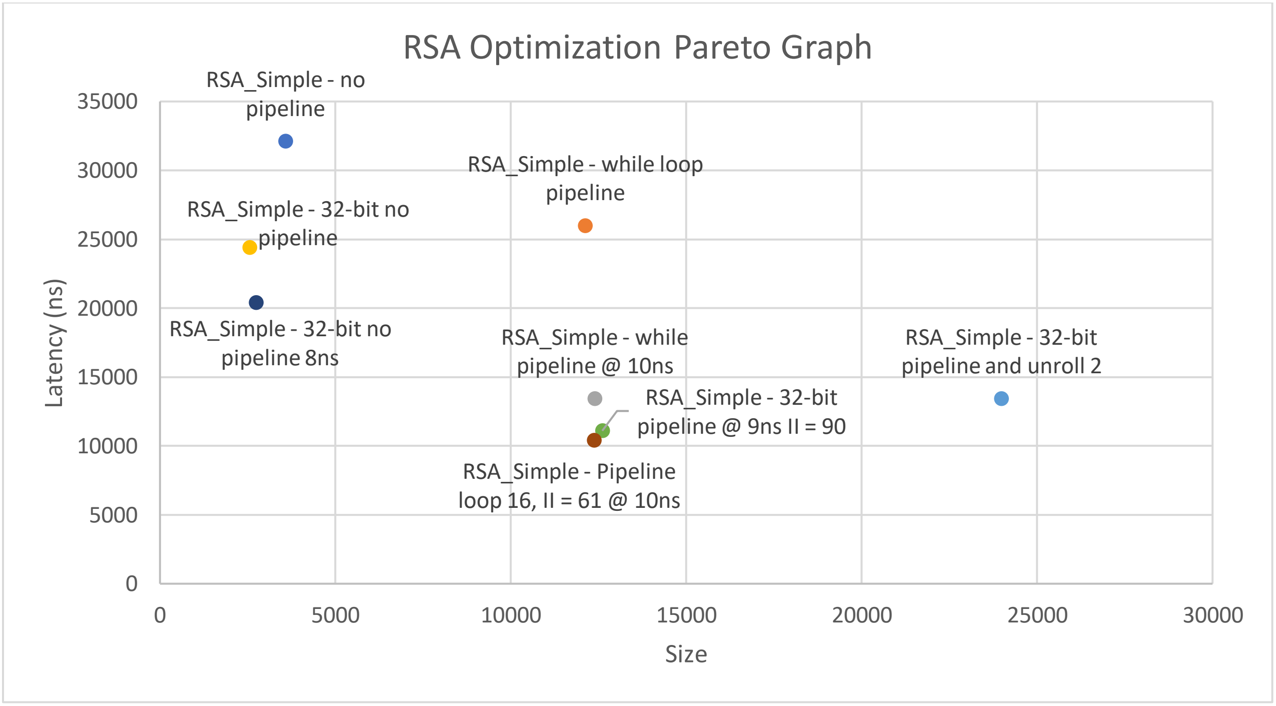


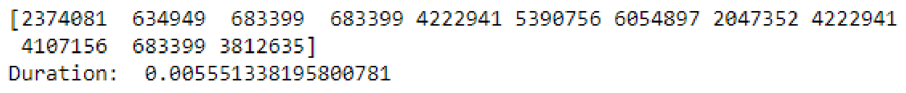
Figure 12: Pareto Graph of Optimizations for RSA Simple

The pareto graph showed that pipelining appeared to be the best trade-off for the optimizations since it decreased the latency more significantly. Another noted result was the benefit of using smaller bit values. While the 32-bit version had a smaller area as expected, it additionally resulted in a significant latency speed-up. The unfortunate consequence of using a smaller bit size would be the notable trade-offs with the level of security this RSA implementation introduces. What allows RSA to remain so secure is that cracking it requires factorizing prime values for complex arithmetic. If the prime values selected are particularly large, in the range of 1024 to 4096, then it becomes significantly harder to factorize the result. Therefore, using smaller data types for the keys limits the level of security this RSA implementation provides. Additionally, compared to other encryption algorithms, RSA requires more bits to maintain the same level of security according to the Nation Institute of Standards and Technology [4]. It is also noted that developments in quantum computing is generating insecurity with RSA encryptions, and that soon the standard key bit size will have to increase [3]. Therefore, RSA is becoming more challenging to implement at high security levels than other encryption systems [11].

With all these optimizations considered and tested on the simple RSA synthesis, the next step was adapting the directives to the final RSA implementation with the key generator. As stated, this iteration used dynamic variables for the loops, and consequently was unable to estimate latency and struggled to schedule certain optimizations. The right-to-left binary evaluation of the modular exponential was pipelined in the directives, but it was unable to be fully scheduled due to the loop uncertainty. The result of just the modular exponential optimization was only a slight speed-up. It seemed that the key generation functions were causing more of the delay.

From this observation, different pipelining directives for the key generation functions were tested. The best optimization for the key generation was pipelining the for loop in the public key generator, which contained the GCD function. Another small optimization for the final version of the RSA implementation was hoisting the code that switched between encryption and decryption. Prior, the Boolean would be checked every time in the stream while loop and set the exponent value to the public or private key based on the input. Now, the code checking the Boolean value is hoisted outside the loop and sets a static variable to the selected key.

With all the optimizations, the overlay was generated again and tested. The best result of these optimizations is shown in figure 13.



```
[2374081  634949  683399  683399 4222941 5390756 6054897 2047352 4222941
 4107156  683399 3812635]
Duration:  0.005551338195800781
```

Figure 13: RSA Implementation with Key Generation Optimized Results

The result of all these optimizations significantly improved the latency from the original test – however the final duration was still not much faster than the PS. Additionally, this implementation was limited to 32-bit key encryption and was therefore very insecure. The optimization process of the final implementation was very tedious since the overlay needed to be generated each time it was tested. This was because the HLS synthesis couldn't provide details on speed-up, and only testing the overlay yielded results for duration. Therefore, each optimization design change took ~10 minutes to test, resulting in less optimization attempts.

Conclusion:

Developing RSA on FPGA is very useful technology for dedicated digital security hardware and has many interesting ways to approach implementing it. The approach focused on in this implementation was unique in that it utilized AXI-Stream interface for communicating between the hardware and software. AXI-Stream seemed very promising as it would limit the size of data being transferred to the PL based on the way the software was implemented. Unfortunately, AXI-Stream posed some challenges, mainly with delay and scheduling in synthesis. The while loop for the AXI-Stream was unable to be pipelined and made it more difficult to optimize. Additionally, when running the overlay in succession, the second run would result in more significant latency and would vary drastically. This is likely the result of asynchronous digital logic or burst resetting. Luckily, for this implementation where only one message is sent at a time, this issue does not pose any limitations to its performance. However, in a practical application where the hardware needs to constantly encrypt/decrypt data, it could pose complications.

In the end with all the optimizations, the hardware was able to perform the same computations as the software in a slightly faster duration. The RSA algorithm utilizes a lot of iterative logic when expanding its arithmetic functions, and therefore made for a suitable topic to implement in hardware. The major limitations as to why this couldn't be implemented as 1024 RSA or excessively faster than the PS is due to the FPGA board limitations and the design choices.

When initially implementing the hardware source, the key size was tested and synthesized with 512 bits. It was quickly discovered that the area required for the synthesis far exceeded the available area in the FPGA. Therefore, the decision was made to limit the bits sizes to smaller values and sacrifice some of the security of the system. The implementation was able to synthesize 128-, 64-, and 32-bit RSA encryptions, but could not be optimized above 128. With the intention to focus on speed, the 32 bit was selected since it was fastest and allowed testing of additional optimizations.

As a result of these trade-offs, this implementation might not be best suited for a practical use case since it does not maintain the necessary level of security. With the expected developments in quantum computing, RSA security is likely going to be insecure below 2048-bit key encryption [12] [3]. The industry standards even suggest adapting to 4096-bit for more sensitive data. Therefore, the small key size of this implementation is a major liability and cannot be resolved without a significant increase in the latency or overallocation of the area.

While RSA is still a very popular and secure encryption method, there are several other encryption systems that might have been more suitable since they can provide the same level of security with less bits [5]. Research has also indicated that other encryption algorithms, like ECC, are able to perform faster at larger key and security bit values [11]. A potential way to further improve this implementation would also be exploring enhanced RSA, which implements the keys from four prime values rather than two to produce faster results [13].

Overall, examining the RSA algorithm was fascinating since it is a simple algorithm that can be expanded to be very complex. This made it a great topic to focus on implementing in hardware as it had the framework for learning how to develop IP with AXI interfacing. Unfortunately, due to RSA's unsustainability and the fact better encryption algorithms exist, this implementation was not the most practical focus. If a different encryption system was selected instead, then less bits could be used for the same level of security. Regardless, the process of developing this RSA interface reinforced an understanding of hardware design, and interfacing between hardware and software.

References

[1] R. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," February 1978. [Online]. Available: http://people.csail.mit.edu/rivest/Rsapaper.pdf.

[2] Geeks for Geeks, "RSA Algorithm in Crpytography," 23 January 2023. [Online]. Available: https://www.geeksforgeeks.org/rsa-algorithm-cryptography/.

[3] D. Goodin, "RSA's demise from quantum attacks is very much exaggerated, expert says," Ars Technica, 26 January 2023. [Online]. Available: https://arstechnica.com/information-technology/2023/01/fear-not-rsa-encryption-wont-fall-to-quantum-computing-anytime-soon/. [Accessed 26 April 2023].

[4] E. Barker and A. Roginsky, "Transitioning the use of cryptographic algorithms and key lengths," National Institute of Standard and Technology, 2019.

[5] E. Barker, "Recommendation for key management: Part 1," National Institute of Standards and Technology, 2020.

[6] T. O'neal, "Using cryptography in Zynq UltraScale MPSoC," Xilinx Inc., 15 January 2020. [Online]. Available: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842541/Using+Cryptography+in+Zynq+UltraScale+MPSoC#UsingCryptographyinZynqUltraScaleMPSoC-RunningRSAExample. [Accessed 26 April 2023].

[7] Javatpoint, "RSA Encryption Algorithm," [Online]. Available: https://www.javatpoint.com/rsa-encryption-algorithm. [Accessed 26 April 2023].

[8] P. Kahrer, "Generating (Very) Large Primes," LANGUI.SH, 7 March 2009. [Online]. Available: https://langui.sh/2009/03/07/generating-very-large-primes/. [Accessed 26 April 2023].

[9] Geeks for Geeks, "Modular exponentiation (power in modular arithmetic)," Geeks for Geeks, 24 June 2022. [Online]. Available: https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-arithmetic. [Accessed 26 April 2023].

[10] Geeks for Geeks, "Modular Multiplicative Inverse," GeeksforGeeks, 17 February 2023. [Online]. Available: https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/. [Accessed 26 April 2023].

[11] D. Mahto, D. A. Khan and D. K. Yadav, "Security Analysis of Elliptic Curve Cryptography," in *Proceedings of the World Congress on Engineering*, London, UK, 2016.

[12] M. Cobb, "What is the RSA algorithm? Definition Security," TechTarget, 4 November 2021. [Online]. Available: https://www.techtarget.com/searchsecurity/definition/RSA#:~:text=How%20is%20RSA%20secure%3F,directly%20tied%20to%20key%20size.. [Accessed 26 April 2023].

[13] G. Amalarethinam and H. M. Leena, "Enhanced RSA Algorithm with varying Key," in *2016 World Congress on Computing and Communication Technologies*, San Francisco, 2017.

[14] A. Thobbi, S. Dhage, P. Jadhav and A. Chandrachood, "Implementation of RSA Encryption Algorithm on FPGA," *American Journal of Engineering Research (AJER),* vol. 4, no. 6, pp. 144-151, 2015.