RSA Keychain Final Report

1st David Choi
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
dcchoi@mit.edu

2nd Joseph Hobbs
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology
Cambridge, MA
jrhobbs@mit.edu

***Abstract*—We present our design for Keychain, a hardware accelerator for an RSA cryptosystem that allows for asymmetric encryption in real-time applications, particularly a messaging service between two users. We display the block diagram and implementation for a baseline RSA algorithm, then show the various optimizations that we used to handle larger numbers while maintaining efficiency for secure communication. Specifically, we focus on the modular exponentiation portion of the RSA algorithm, which we approach through bit manipulation schemes and other optimizations to allow Keychain to function.**

***Index Terms*—cryptography, field-programmable gate arrays, digital systems, hardware acceleration.**

I. Introduction

Asymmetric encryption is a unique method of secure communication between two parties without the need for a shared secret key, allowing users to encrypt and decrypt data with each other without prior introduction. However, asymmetric encryption is mainly used for sharing a secret key for symmetric encryption, which requires shared keys but is more efficient. We wish to test the efficiency of asymmetric encryption, particularly the RSA algorithm, with our Keychain project. We use hardware optimizations such as bit manipulations for our implementation, allowing for quick and secure communication in a server-client model. In this report, we display our progress in the implementation of Keychain, particularly the optimizations we have made for scaling and efficiency in our encryptor and decryptor. We detail how plaintext and ciphertext data is transferred between personal computers and an external FPGA, and we show how it compares to other implementations in security, efficiency, and size. Finally, we provide some limitations of our design and how we could further improve performance in future iterations of Keychain.

A. System Overview

Our RSA Keychain encrypts and decrypts 256-bit messages by means of either a public key provided by external hardware or a secret key stored in Keychain, which is 512 bits long. The secret key is stored in an internal BRAM block and is not accessible to external hardware for maximum security. Although we were able to get this working in simulation and surpassed our minimally viable product goal of 256-bit keys for 32-character messages, we could not achieve this as Vivado was unable to synthesize our design for unknown reasons. As a result, we were forced to cut down our product to meet timing on hardware with 16-bit messages and 32-bit keys.

B. Operating Modes

As described further below, Keychain operates in one of two modes: Encryption Mode and Decryption Mode. These modes are provided by a bit pattern over UART from hardware, which is sent alongside an action message for encryption or decryption.

1) Encryption Mode: In Encryption Mode, external hardware provides a plaintext message, a public modulus, and a public key over UART. A deserializer prepares the data for encryption, and an encryptor/decryptor module performs the necessary computation to produce the output ciphertext. Finally, a serializer then prepares the ciphertext for transmission back to the external hardware via UART.

2) Decryption Mode: In Decryption Mode, external hardware provides a ciphertext message and a public modulus over UART. A deserializer prepares the data for encryption and accesses an integrated BRAM block to retrieve the secret key. The encryptor/decryptor module then performs the necessary computation to produce the output plaintext. Finally, a serializer prepares the plaintext for transmission back to the external hardware via UART.

II. Background

In this section, we discuss the high-level algorithm and system architecture of our RSA Keychain.

A. RSA Overview

In this section, we will demonstrate the asymmetric key encryption/decryption algorithm provided by RSA. The core principle underlying the RSA algorithm is the *Fermat-Euler Theorem.* It states that, for two coprime integers m and N,

$$m^{\varphi(N)} \equiv 1 \mod N \tag{1}$$

Here, we will define $N = p_1 p_2$, where p_1 and p_2 are large primes, and we use $\varphi(N)$ to denote Euler's totient function. We can generalize this formula by introducing integer k and modifying the exponent slightly so that the result is congruent to m, like so.

$$m^{k\varphi(N)+1} \equiv m \mod N \tag{2}$$

We then select integer k such that $k\varphi(N)+1 = ps$, where p and s are integers. We define p to be the *public key* and s to be the *secret key*. Therefore, we can see that

$$m^{ps} = (m^p)^s \equiv m \mod N \tag{3}$$

In (3), m is the *plaintext* and N is the *public modulus*. This equation reveals a powerful insight: the modular exponentiation function, applied to the input plaintext m twice, is sufficient to perform full RSA encryption. Specifically, the function is

$$\mathrm{ME}(m, e, N) := m^e \mod N \tag{4}$$

By this definition, we can rewrite (3) as

$$\mathrm{ME}(\mathrm{ME}(m, p, N), s, N) = m \tag{5}$$

Therefore, we focus our efforts on constructing a hardware module that can perform the function $\mathrm{ME}(m, e, N)$ efficiently while meeting hardware timing, as this function is sufficient for both RSA encryption and decryption.

B. Block Diagram

We now show visuals of our project, which displays the advancement of data through our encryptor, as well as our various optimizations.

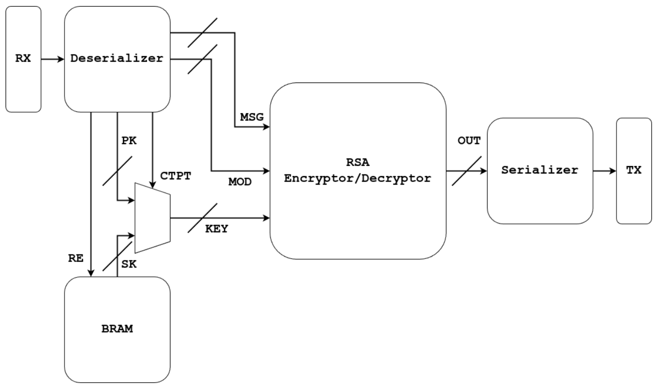

Fig. 1. A top-level block diagram of Keychain.

Figure 1 shows a top-level block diagram for the RSA keychain. Omitted from this block diagram for brevity are the following:

- Clocking. All modules run on a 100 MHz clock. Although this remained constant throughout the development of Keychain, we would like to explore changing the frequency of our clock in future iterations and balance the advantages and disadvantages of higher/lower clock speeds with respect to the amount of pipelining required for performing operations on large numbers.
- Reset. All modules use a common active-high RESET signal that is asserted before Keychain's first use.
- `READY` and `VALID` signals. The deserializer, the encryptor/decryptor, and the serializer are connected with `READY` and `VALID` signals in accordance with the AXI-Stream specification to ensure the modules are communicating effectively.

The bulk of our project lies in the RSA Encryptor/Decryptor block itself, as its design is what determines both the message size and system efficiency for Keychain. We display a more detailed version of this block in Figure 2. We use a modulo calculator that sends and receives data from both BRAM and a multiplier module for efficient computations. We detail these optimizations and their considerations in the following sections.

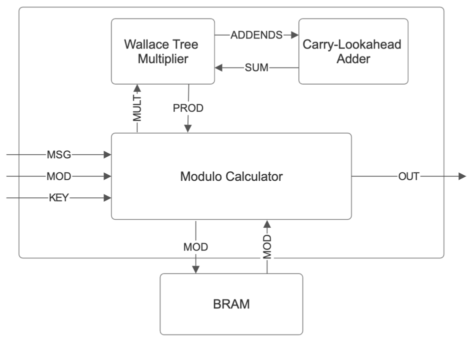

Fig. 2. A close-up of the RSA Encryptor/Decryptor block.

We have designed an implementation on hardware with 16-bit messages and 32-bit keys, which we have extensively tested through simulations on the UART bus. Although we were unable to scale this further, we were able to get to 256-bit messages and 512-bit keys in simulation.

III. Communication

Although an efficient RSA algorithm is the main focus of our Keychain project, our design would not be able to function without robust communication channels for both computer-computer and computer-FPGA communication. We describe both the techniques we used and their reliability below.

A. Overview

To allow our computers to communicate with each other, we use the Python `websockets` library to create a simple server-client model through a WebSocket server. One user sets up the server, while the other user connects through the first user's public IP address. Once both users have established a connection, they can both input messages that are encoded and decoded by Keychain. When the user inputs a message into their terminal, our Python script converts the message into a bytestring and pads it to a fixed length. The script then adds the public modulus and an 8-bit ”action” identifier to the message. If the action is for encryption, the script adds the public key to the message. Otherwise, the FPGA needs to decrypt the message using a secret key that is stored in its BRAM, so the computer just adds more padding to the message. We show an example of this in Figure 3.

Once our extended message has all the information it needs for processing, the computer sends it to the FPGA over UART.

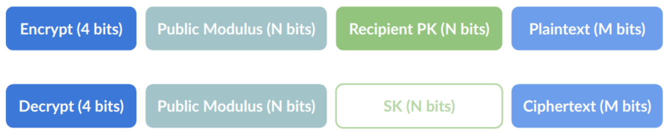

Fig. 3. Communication packet structure. The deserializer must pull the user's secret key from BRAM when it is in decrypt mode.

The FPGA deserializes this data for its RSA calculation, then serializes its data output and sends it back to the computer over UART. This design provides complete confidentiality between users, as all data sent between the WebSocket is encrypted through Keychain.

B. UART

Although we have already designed a UART transmitter and receiver earlier this semester, there was still a significant amount of development and testing that we had to implement for our product. Particularly, we needed to make sure that all data received from the computer was properly transmitted for resilient deserialization. As we note throughout our report, our Keychain design is only effective if it always produces the correct calculations for RSA. As such, successfully deserializing and serializing UART data is a major focus that requires significant testing to ensure its resilience against bugs and edge cases.

IV. Multiplication

For Keychain to properly function, we required a module for efficient multiplication that can both deal with large numbers and meet clock timing. Since the built-in multiplication function in SystemVerilog is not suited for this performance, we document our various approaches to this problem as well as our final product for Keychain's design.

A. Wallace Tree Multiplier

Our initial design used a Wallace Tree Multiplier (WTM) for our multiplication, which is a hardware-based multiplier that adds partial products to significantly decrease the time it takes to multiply two numbers compared to a standard design. From our research, we concluded that our WTM would produce results in $O(N^{1.5})$ time, compared to a simple multiplier delay of $O(N^2)$ [1]. We show an example of multiplying two 4-bit numbers with our WTM in Figure 4. When given two inputs, the WTM organizes the partial products by digit, then continuously uses both half-adders and full-adders to reduce the partial products into sum and carry bits. Due to the small computation time of these two-to-three bit adders, the WTM can do many of these operations at once, which is where the optimization takes place. Once there are only one or two bits at each digit, the WTM simply stores these bits into two bitvectors and sums them to get the overall product of the original 4-bit numbers.

We began by creating a simple 4-bit WTM in SystemVerilog for a conceptual understanding of its intricacies, which we attempted to scale up to a 128-bit version. However, each

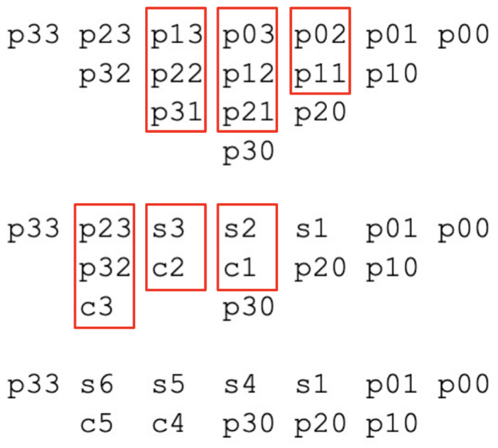

Fig. 4. A 4-bit Wallace Tree Multiplier. The partial products of the numbers are reduced in stages of full adders and half adders until there are only one or two bits in each digit.

WTM has a specific configuration based on its number of bits, which consists of thousands of full adders and half adders that we had to calculate individually. Due to this specificity, we also realized that the design for an N-bit WTM would not provide much usefulness for the design of a 2*N-bit WTM. Although we developed a Python script to help configure all of these adders, there were still many issues we needed to deal with, such as adding pipelining to meet timing and keeping track of all the adders. We include the initial attempt for our 128-bit WTM multiplier in our codebase, which has more than 2,000 lines of code for the first stage of partial product reduction.

B. Karatsuba Multiplier

Although the WTM's efficient design seemed promising, we soon realized that most of our project's focus would be on manually testing and debugging the thousands of adders for its partial product optimizations, which would be infeasible in our project timeline. Because of this, we changed our multiplication design into a Karatsuba multiplier, which is a divide-and-conquer algorithm that recursively reduces the multiplication of two N-digit numbers into three multiplications of $N/2$-digit numbers. These reductions allow Karatsuba to work in $O(N^{\log_2 3})$ time, while requiring much less code from its recursive design. We show an example of Karatsuba in action in Figure 5, which breaks down the numbers 1234 and 567 to get the product 699678.

We created a parameterized design of the Karatsuba algorithm, and we were able to get our design working for numbers up to thousands of bits in our logic simulators. However, we were unable to get this new implementation working in Keychain, as Karatsuba's reliance on the built-in multiplication module became an issue for timing when key sizes became too large. We suspect that this could be fixed through better pipelining of our intermediate values, but we did not have time to implement this and resorted to a different multiplication method.

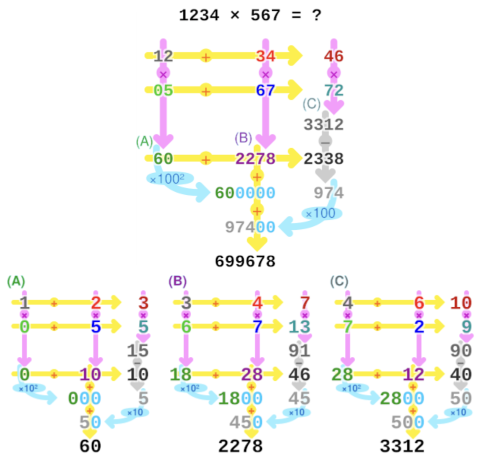

Fig. 5. A Karatsuba Multiplier. Both operands are reduced into smaller pieces.

C. Final Design

Although both the WTM and Karatsuba seemed like promising optimized multipliers that would significantly reduce the number of cycles for a calculation, we ran into setup time violations and struggled to pipeline their internal logic sufficiently to meet timing requirements. Because of this, we have designed a bit-shift multiplier that adds up partial products at each cycle until it reaches the full product. We provide a visual of this multiplier in action in Figure 6. Although this is less efficient than WTM and Karatsuba, it still implements some optimizations without violating setup time constraints, such as doing many of these shifts and additions in parallel. Still, future iterations of this design would use a well-established and efficient hardware multiplication module, as we discuss in the Evaluation section.

```
1011 * 0110
            shifts:
      0000 ← << 0
     10110 ← << 1
    101100 ← << 2
  +0000000 ← << 3
   1000010
```

Fig. 6. A Bit-Shift Multiplier. This module creates partial products by shifting the first input based on the bits of the second input, then adds them all up for a product.

V. Modular Exponentiation

Once we have our output from our exponentiation operation, we want to compute $a^b \mod c$ in an efficient way in hardware. We will express b as its binary equivalent $b_0 + 2b_1 + 4b_2 + \cdots + 2^n b_n$. We will then rewrite our desired calculation as

$$a^{(b_0+2b_1+4b_2+\cdots+2^n b_n)} \mod c \tag{6}$$

Because $b_i \in \{0,1\}$, we have simplified the problem to computing $a^{2^n} \mod c$ for integers $n \geq 0$. Once we have the value of each, we may selectively multiply them together based on the values of b_i to determine our final answer. We may perform the binary modular exponentiation as follows. First, compute $a^0 \mod c \equiv 1$. Then, using the following rule, generate successive powers of a modulo c.

$$a^{2n} \mod c = (a^n \mod c)^2 \mod c \tag{7}$$

The multiplication and squaring operations are performed by our bit-shift multiplier, and we have a pipelined modulo calculator that uses multiple subtractions to work with large numbers.

VI. Memory

Because Keychain is designed for secure communication between devices through RSA, it encourages security by storing the user's secret key in BRAM. This key is not available outside of the FPGA and is only pulled from BRAM when the user is decrypting a message, which sends a "decrypt" message through UART. When the deserializer receives this message, it pulls the secret key from BRAM and continues with the modular exponentiation calculation. Otherwise, the deserializer is in "decrypt" mode and simply extracts the public key over the UART data. We provide a visual of this in Figure 7.

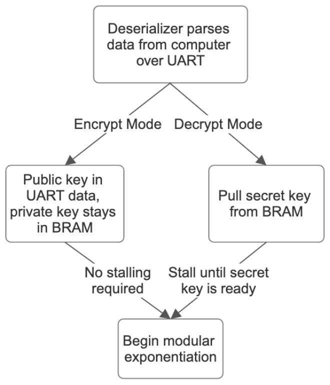
Deserializer parses data from computer over UART
Encrypt Mode
Decrypt Mode
Public key in UART data, private key stays in BRAM
Pull secret key from BRAM
No stalling required
Stall until secret key is ready
Begin modular exponentiation

Fig. 7. The deserializer must pull the user's secret key from BRAM when it is in decrypt mode.

Because our hardware only supports up to 32-bit public and secret keys, we were able to fit our secret key in just one entry of our BRAM, preventing the need for address indexing for a full key. This allowed our deserializer to only stall for a few cycles while waiting for an output from BRAM. However, we assume that this may not work as our key size expands to thousands of bits in length, forcing the key to be split into chunks across the BRAM for a secure transfer that meets timing. This would be something to implement in future iterations of our design.

There are definitely some more optimizations that we could implement in our design through additional BRAM usage, especially for our large number subtraction in our modulus module. However, this would likely not decrease the timing requirements for the modulus calculations, and would be more focused on reducing the resource usage of our product. Because timing was a bigger focus than resource usage throughout the development of Keychain, we did not see this as a large priority for our design. Still, we discuss this in further detail in our Evaluation section.

VII. Performance

In this section, we evaluate our Keychain product based on performance metrics and hardware resource utilization.

A. Size

Our physical hardware implementation could only accept 32-bit keys and 16-bit messages, as Vivado proved incapable of synthesizing designs with larger key and message widths. However, we demonstrated in simulation that our design could be scaled to accept 512-bit keys and 256-bit messages.

B. Resources

Table I provides a breakdown of the resources utilized on our FPGA. We assume a 32-bit key and a 16-bit message due to the limitations mentioned above.

TABLE I
Resource Utilization of Keychain

Resource	Used	Percentage
Slice LUT	2651	8.13%
Flip-Flop	6029	9.25%
F7 MUX	645	3.96%
BRAM	0	0.00%

Clearly, as we are using less than 10% of our FPGA's resources, we are capable of scaling up our project. Currently, our only limitation is Vivado's synthesis engine. We did not implement BRAM to store intermediate values; however, doing so would reduce flip-flop utilization as we increase key size.

C. Timing

Table II summarizes the timing characteristics of our design. In order to meet timing requirements, we had to reduce our clock speed to 10 MHz from the initial 100 MHz. However, based on the WNS value shown, we are capable of increasing our clock speed to approximately 33 MHz without risking timing violations. Further logic optimizations would permit higher clock speeds of 100 MHz or greater.

TABLE II
Timing Analysis of Keychain

Variable	Value
Clock Frequency	10 MHz
Worst Negative Slack	70.100 ns
Worst Hold Slack	0.054 ns
Maximum Clock Skew	-0.296 ns

Our timing analysis also shows a WHS of 54 picoseconds. This value is very low, but we are not concerned about this causing timing violations as we scale because hold slack becomes more positive as contamination delays increase. For larger key sizes, we expect contamination delays to increase, so we expect the WHS to only increase. However, because setup slack decreases with larger propagation delays, we expect the need to conduct further optimizations in order to meet timing as our project scales to 1024-bit or larger keys.

D. Efficiency Comparison

Table III summarizes the results of a comparison study analyzing the relative performance of our Keychain implemented on a Spartan 7 FPGA and a Python 3 script. The Keychain results were computed using the `cocotb` simulation tool assuming a 10 MHz clock, and the Python 3 results were computed by computing the average duration of 100,000 encryption operations run in CPython on an AMD Ryzen 5 64-bit processor core.

TABLE III
Comparison of RSA implementation efficiencies

Method	Key Width	Cycles	Clock	Time
FPGA	32	4296	10 MHz	429.60 μs
Python	32	12576	2.95 GHz	4.27 μs
FPGA	256	248222	10 MHz	24822.00 μs
Python	256	291543	3.17 GHz	92.00 μs

It is easy to see that our application-specific hardware implementation can consistently complete in fewer operations than the Python implementation, as the AMD Ryzen 5 processor core is a general-purpose computing device. However, Python consistently performs faster as the Ryzen 5 operates at a clock speed of approximately 3 GHz, 300 times faster than our 10 MHz clock. A hypothetical ASIC using our architecture could therefore outperform general-purpose computer hardware at comparable clock speeds. Additionally, as discussed below, more advanced hardware arithmetic logic would decrease the number of clock cycles required even further.

VIII. Evaluation

Regarding the success of our product, we specifically look into our design test strategies that led to complete confidence in Keychain's ability to parse UART data and calculate modular exponentiation for large numbers. We also look into some of the additional optimizations that we could add into future iterations of Keychain, assuming that we had more time to test and implement these changes.

A. Testing

One of the highlights of our project is the extensive testing we have done on our modules, particularly the UART TX/RX, serializer/deserializer, and modular exponentiation blocks. Because our project deals with secure communication, it is integral that we always generate the correct ciphertext and plaintext messages through our cryptosystem. Therefore, we had to conduct rigorous verification on our system to eliminate any possible edge cases, and both unit testing on each module as well as integration tests on the entire hardware implementation verified this. We display an example of one of our integration tests in Figure 8.

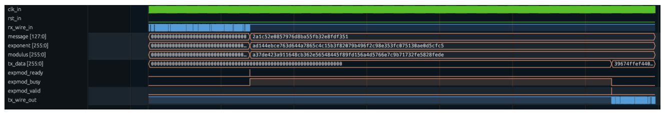

Fig. 8. Simulation output for 256-bit Keychain (MVP). The encryption completes correctly in 248K clock cycles.

B. Future Works

Although Keychain is functional and can successfully encrypt and decrypt messages, there are still many additional features and optimizations that we could implement into our design in future iterations. First of all, we were unable to get Keychain working for large key sizes with 100 MHz clock speeds, and we had to lower this frequency to 10 MHz to perform large operations and meet timing. This could be solved by further pipelining some of these operations, such as the large subtractions in our modulus block. We could also decrease the resource usage in our FPGA by utilizing BRAM to store intermediate values during our calculations.

Additionally, it would be good to replace our bit-shift multiplier with one of our previously tested hardware multipliers, such as WTM or Karatsuba. Either of these multipliers could significantly reduce the amount of time spent in the modular exponentiation phase if implemented correctly.

Overall, we were able to achieve our minimally viable product in simulation, but we did not have time to expand our cryptosystem further to handle larger keys in hardware. This was largely due to certain modules such as our UART deserializer requiring more focus than we had originally expected, as well as our various attempts for a working hardware multiplier that could tailor to our needs.

IX. Conclusion

Our design, as specified in this report, is an implementation of an optimized external RSA algorithm to transfer secure data between two users. Our hardware accelerator, Keychain, both encrypts and decrypts 16-bit messages using 32-bit public and secret keys. Although this was the farthest we were able to successfully test in hardware, we were able to simulate our minimum viable product of 128-bit messages with 256-bit keys and produce correct results. Much of our work was on extensive testing for our modular exponentiation function, so we are confident that we will always get expected results from our hardware calculations, even when sending and receiving data over UART. We are very proud with our design, as it has given us a much deeper understanding of both digital system development and secure cryptography, and we are excited to show the work that we have done this semester.

X. Source Code

[redacted]

References

[1] MIT, "6.111 Lecture 13: Arithmetic: Multiplication" 2011, [Online]. Available: https://courses.csail.mit.edu/6.111/f2007/handouts/L13.pdf