Prime Factorization Through Reversible Logic Gates

by

Patrick J. Bollinger

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in the

Electrical and Computer Engineering

Program

YOUNGSTOWN STATE UNIVERSITY

May, 2019

Prime Factorization Through Reversible Logic Gates

Patrick J. Bollinger

I hereby release this thesis to the public. I understand that this thesis will be made available from the OhioLINK ETD Center and the Maag Library Circulation Desk for public access. I also authorize the University or other individuals to make copies of this thesis as needed for scholarly research.

Signature:

Patrick J. Bollinger, Student Date

Approvals:

Dr. Frank X. Li, Thesis Advisor Date

Dr. Jalal Jalali, Committee Member Date

Dr. Eric W. MacDonald, Committee Member Date

Dr. Salvatore A. Sanders, Dean of Graduate Studies Date

ABSTRACT

The purpose of this thesis is to determine the feasibility of using hardware to perform prime factorization of a semiprime number. The application of this research can primarily impact the field of mathematics as well as cybersecurity. This research dives into deconstructing the view of digital logic gates being one-way functions and proposes to reverse the typical flow of information. By using the reversible logic gates developed, larger reversible digital circuits are constructed until a full array multiplier is ready for testing. An analysis is performed with a semiprime number of 4 binary digits up to 1024 binary digits long, to see the effectiveness of factorization using this method. This analysis is to replicate the textbook definition of RSA public key generation; generating two similar in length prime numbers and then taking the product of them. Although the reversible logic gates are able to deduce new information, it is not enough information to perform the prime factorization of a semiprime number. Based on these results, we can conclude that more information needs to be created in order for reversible logic gates to be a feasible method of prime factorization. Further research can be performed to develop more information, such as defining more relationships between bits, and research can be done to apply the reversible logic gates to other digital circuits.

ACKNOWLEDGEMENTS

I would like to thank Dr. Jalali, Dr. Li, and the rest of the faculty of the Electrical and Computer Engineering department for their teachings and support as I pursued this milestone in my education and life. I would also like to thank my friends and family for their support and apologize for any absence these past two years. Last, but certainly not least, I would like to thank my wife, Judy. She is a daily inspiration to me and I could not have done this without her support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CSV ..... Comma-separated values
I/O ..... Inputs/Outputs
RSA ..... Rivest–Shamir–Adleman
VHDL ..... VHSIC Hardware Description Language
VHSIC ..... Very High-Speed Integrated Circuit

CHAPTER I: INTRODUCTION

1.1 Motivation

The product of two large prime numbers is used in common encryption algorithms, such as the RSA algorithm [1], as the foundation for public key encryption in cybersecurity. As a result, the company formerly known as RSA Laboratories created a monetary challenge for researchers to try and perform the factorization of specific semiprime numbers [2, 3]. The monetary reward was withdrawn between May and June of 2007 [4]. Even with the withdrawal, there still exist challenges that have yet to be solved, based on the number of bits in the semiprime number.

1.2 Purpose

The purpose of this thesis is to investigate a novel approach for performing factorization of semiprime numbers. This approach investigates the process of hardware multiplication, using an array multiplier, and determining if the typical one-way operation can be reversed.

1.3 Objectives

The objectives of this thesis are as follows.

- Develop reversible logic gates using Python
- Develop unit tests for the reversible logic gates using Python
- Assemble the reversible logic gates until a reversible generic multiplier array is formed.
- Evaluate new information generated when supplying the multiplier array with a given semiprime for its output.

1.4 Organization

This thesis is organized into 5 chapters. Chapter II describes in greater depth the background for this research. Chapter III discusses the design of this research. Chapter IV reviews the results of this research. Chapter V provides a conclusion to this research.

CHAPTER II: LITERATURE REVIEW

2.1 Factorization of Semiprimes

2.1.1 Semiprime Numbers in Cybersecurity

A semiprime number is a number that is the product of two prime numbers [3]. Given the number 35, it is easy to realize that 5 and 7 are the only factors, other than 1 and itself. Based on this information, we can conclude 35 is a semiprime number. This example may seem trivial, but what if we needed to determine the prime factors of 113547311?[1] This would prove to be difficult by hand since we would have to iterate through all the prime numbers until we found one that divided evenly into 113547311.

The difficulty in factoring semiprime numbers is a basis of security in the RSA encryption algorithm. The semiprime number contributes to a public key used to encrypt a message and the prime factors contribute to the creation of the private key used to decrypt a message. The sender of a message will use the recipient's public key to encrypt a message and the recipient will use their private key to decrypt the message. Figure 1 displays a high-level overview of how information is encrypted and decrypted.

[1] The answer is 10601 and 10711, for those who are curious.

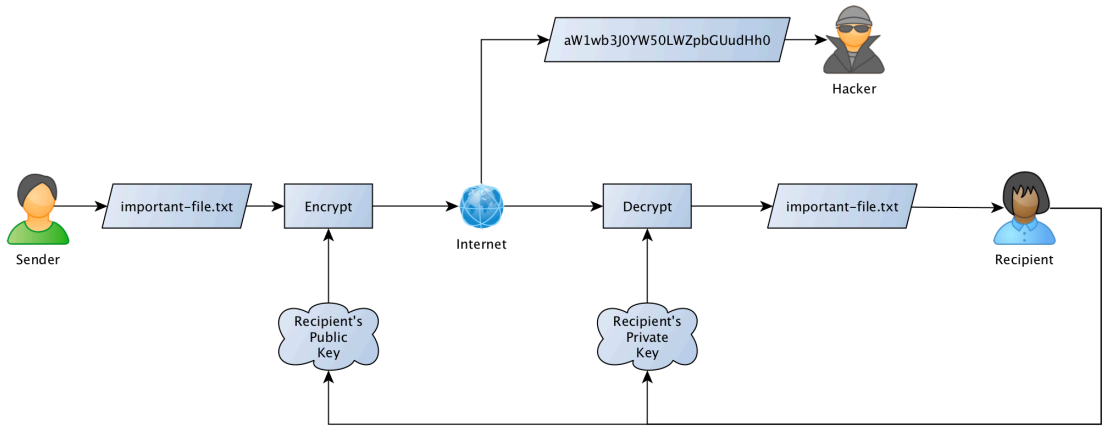

Figure 1: Overview of Public-Key Encryption

It is important to note that the difficulty of factoring a semiprime number increases exponentially as the length of the number in bits increases.

2.1.2 An Example of RSA Encryption

To demonstrate how this method of encryption fully works, the following is an example.

Let's start with a sender who wants to send the message "hi". If we treat "hi" as an 8-bit ASCII character string, it can be represented as the binary number "01101000 01101001" or the decimal number "26729". This sender now needs to have the public key of the recipient, so let us generate it next.

The public key is first generated by generating two prime numbers. We will use 163 and 157 as our prime numbers. We can then take the product of the prime numbers to generate a semiprime, $n = 163 * 167 = 27221$. Notice that the semiprime is larger than the decimal value of the message we are sending; this is important for the math to work out. Next, we subtract 1 from each prime number and multiply them together for the totient, $\phi(n) = (163 - 1) * (167 - 1) = 26892$. Now, we choose a coprime of the

totient, denoted as *e*. Numbers are coprime if they do not share common factors. Therefore, we can select 11 as a coprime of 26892 since they do not share common factors. Thus, $e = 11$. Now that we have a coprime to the totient selected, we have all the information for the public key. For this example, the public key is $(n = 27221, e = 11)$ and could be shared over the Internet.

The sender of the message "hi" can now use this public key to encrypt the message using the following formula, where *m* is the message being sent and *c* is the encrypted message: $c = m^e \ mod \ n = 26729^{11} \ mod \ 27221 = 4272$. Therefore, our encrypted message has a decimal value of 4272.

Finally, the recipient needs to be able to decrypt this message. This can be done by generating the private key. We have to choose a "*d"* that satisfies the following: $d * e \equiv 1 \ mod \ \varphi(n)$. Let us choose $d = 36671$ in this example. Now, we have all information to create a private key, $(n = 27221, d = 36671)$. From this private key, we can decrypt the message with the following formula: $m = c^d \ mod \ n = 4272^{36671} \ mod \ 27221 = 26729$. Notice that the outcome matches the original message's decimal value.

One item to note from this example is how the "*n"* is shared in both the public key and the private key. This is why research is being performed on how easily factorable this semiprime number, because from the semiprime number, you can figure out all other important information for this encryption methodology.

2.1.3 Existing Method for Factoring Large Semiprime Numbers

When it comes to factoring semiprime numbers, there are many approaches that can be taken. The most obvious, and naïve, would be to iterate through the primes one-

by-one and checking if the prime divides evenly into the semiprime number. This method would take an extremely long time. For instance, if a semiprime for RSA-768 was chosen, a person would have to go through $\sim 10^{116}$ numbers, at most, before finding the factors.

The most common method for factoring large semiprime numbers to date is using the General Number Field Sieve algorithm. [5] This algorithm is quite complex in nature and requires a heavy understanding of mathematics to complete. At the time of writing this thesis, the current largest RSA number factored is RSA-768, a 232 decimal digits (768 bits) semiprime number. [6] This factorization was completed using a Number Field Sieve algorithm.

2.2 Array Multiplier

In order to understand how we can take a semiprime number and factor it, we first need to understand how numbers are multiplied. At a young age, we are taught how to perform multiplication by hand. The act of multiplication involves iterating through each of the digits being multiplied, multiplying the pairs together, and then carrying over any product to the next column. For computers and other electronic devices, the process can be very similar. Whereas we use a base 10 numbering, electronics use a base 2 numbering system (binary) in order to perform arithmetic. With binary numbers, these arithmetic operations are performed by logic gates.

2.2.1 Logic Gates

Logic gates are an electronic implementation of binary functions. There are a number of different logic gates that are used to build electronics. For multiplication, we will focus on the following three logic gates: *AND*, *OR*, and *XOR*.

Logic gates can be represented as a truth table, where you list all possible inputs to the binary function and the corresponding output. Below are truth tables for the three logic gates that we will investigate in this thesis.

Input A	**Input B**	**Output Y (Input A *AND* Input B)**
0	0	0
0	1	0
1	0	0
1	1	1

Table 1: AND Gate Truth Table

Input A	**Input B**	**Output Y (Input A *OR* Input B)**
0	0	0
0	1	1
1	0	1
1	1	1

Table 2: OR Gate Truth Table

Input A	**Input B**	**Output Y (Input A *XOR* Input B)**
0	0	0
0	1	1
1	0	1

1	1	0

Table 3: XOR Gate Truth Table

When an engineer is designing a circuit using these logic gates, the logic gates have an associated graphical representation. There are various standards of graphical representation, but this thesis will utilize IEEE Std 91/91a-1991. Following are the graphical representations of the same three gates, with inputs being on the left side of the gates and a single output being on the right side of the gates.

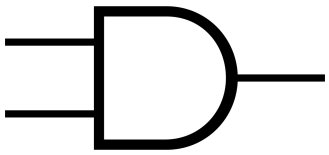

Figure 2: AND Gate Graphical Representation

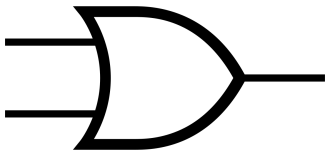

Figure 3: OR Gate Graphical Representation

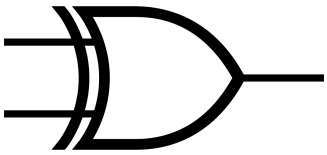

Figure 4: XOR Gate Graphical Representation

2.2.2 Full-Adder Digital Circuit

With logic gates, many different types of digital circuits can be constructed for different purposes. It is known that multiplying is the repeated operation of addition, therefore the circuit that will perform multiplication will be composed of sub-circuits that can perform addition.

The most fully featured digital circuit for adding binary numbers is called the full adder. It consists of 3 inputs (Input A, Input B, and Carry-In) and 2 outputs (Sum and Carry-Out). With the full adder, you can cascade them to add any size binary number.

Following is a truth table that describes the relationship between the inputs of the full adder and its outputs. In addition, there is a circuit diagram to show how the logic gates discussed previously create this full adder.

Inputs			Outputs	
A	**B**	**Carry-In**	**Sum**	**Carry-Out**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 4: Full Adder Truth Table

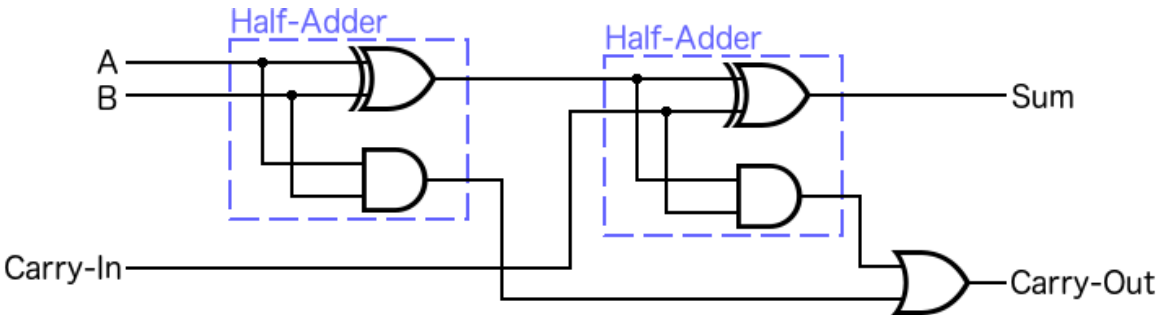

Figure 5: Full-Adder Digital Circuit

2.2.3 Array Multiplier Digital Circuit

Just like there are multiple methods to perform multiplication by hand, such as the standard algorithm or the partial product algorithm, there are multiple methods for

implementing a digital circuit to perform multiplication. The main method this thesis will focus on is the array multiplier. The array multiplier is a digital circuit that is composed of an array of full adders. This thesis will focus on one implementation, the ripple-carry multiplier array.

The ripple-carry multiplier array is similar to how we perform the standard multiplying algorithm on paper; you take the first digit, multiply it through (while carrying-over as necessary) and then move to the next digit after shifting by a power of ten. Following is a diagram that shows the construction of a 4-by-4 multiplier array. In the diagram, the inputs are denoted by *x#* and *y#*, where *x* and *y* are the inputs and # represents the index of the bit, starting at 0. For example, if we looked at the product of 11 and 13, 11 is represented by *1011* in binary and 13 is represented as *1101* in binary. If we treat 11 as our *x* term, then *x3* would equal *1*, *x2* would equal *0*, *x1* would equal *1*, and *x0* would equal *1*. Continuing, 13 would be our *y* term, so *y3* would equal *1*, *y2* would equal *1*, *y1* would equal *0*, and *y0* would equal 1. After the multiplier cells have processed their inputs, the final product would be given at the bottom. Since the product of 11 and 13 is 143 and 143 can be represented in binary as *10001111*; *p7*, *p6*, *p5*, *p4*, *p3*, *p2*, *p1*, and *p0* would equal *1*, *0*, *0*, *0*, *1*, *1*, *1*, and *1*, respectively.

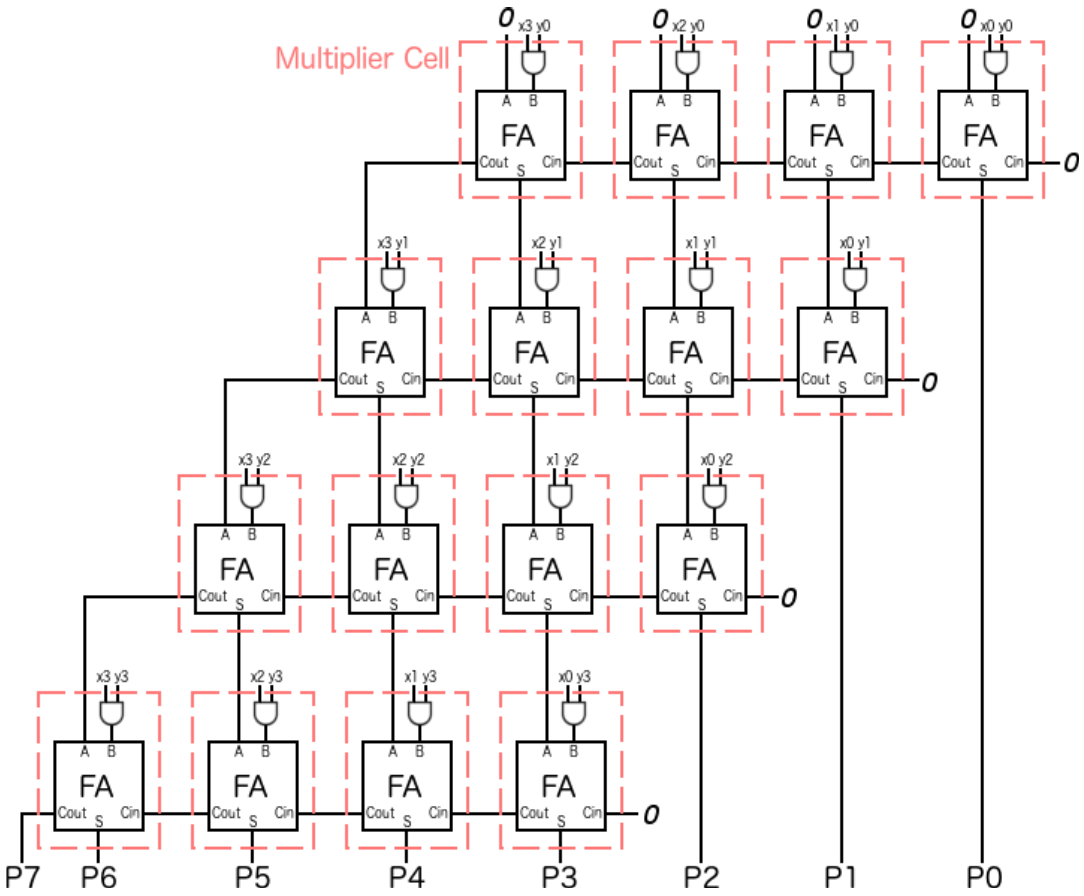

Figure 6: Digital Circuit of Array Multiplier

CHAPTER III: DESIGN OF REVERSIBLE ARRAY MULTIPLIER

3.1 Reversible Logic Gates

For clarification, this section will be defining reversible, for logic gates and the digital circuits created with them, as the ability to deduce what the inputs and outputs should be based on the given state of the inputs and outputs. There exists terminology in the community for reversible logic gates, or computing, to create models that would conserve energy, such as the Fredkin gate. [7] There is previous research on creating an array multiplier using the quantum reversible logic gates, but this research focuses on the conservation of energy rather than the ability to deduce information. [8]

In this section, we will step through the construction of an array multiplier, much like Chapter II, and define what the reversible logic gates/circuits are and the impact they have on the factorization of semiprime numbers.

3.1.2 Reversible AND Gate

As mentioned earlier, the core of this thesis is the ability to deduce information based on what is given. For instance, if the output of a AND gate is *1* and the inputs are unknown (herein referred to as *X*), what can we deduce about the AND gate? We can deduce that both inputs for the AND gate are *1*, resulting in a fully defined state of the AND gate. Following is a table that breaks down the various states the AND gate can be in and the corresponding state that we can deduce. As we design the reversible AND gate, we have to track erroneous states that would be impossible for a traditional AND gate.

State and Information at a given time				Deduced State and Information			
State #	**Input A**	**Input B**	**Output A AND B**	**State #**	**Input A**	**Input B**	**Output A AND B**
1	X	X	X	1	X	X	X
2	X	X	0	2	X	X	0
3	X	X	1	27	1	1	1
4	X	0	X	5	X	0	0
5	X	0	0	5	X	0	0
6	X	0	1	Error	X	X	X
7	X	1	X	7	X	1	X
8	X	1	0	17	0	1	0
9	X	1	1	27	1	1	1
10	0	X	X	11	0	X	0
11	0	X	0	11	0	X	0
12	0	X	1	Error	X	X	X
13	0	0	X	14	0	0	0
14	0	0	0	14	0	0	0
15	0	0	1	Error	X	X	X
16	0	1	X	17	0	1	0
17	0	1	0	17	0	1	0
18	0	1	1	Error	X	X	X
19	1	X	X	19	1	X	X

State and Information at a given time				Deduced State and Information			
State #	**Input A**	**Input B**	**Output A AND B**	**State #**	**Input A**	**Input B**	**Output A AND B**
20	1	X	0	23	1	0	0
21	1	X	1	27	1	1	1
22	1	0	X	23	1	0	0
23	1	0	0	23	1	0	0
24	1	0	1	Error	X	X	X
25	1	1	X	27	1	1	1
26	1	1	0	Error	X	X	X
27	1	1	1	27	1	1	1

Table 5: Reversible AND Gate State Table

Following is a graphical representation of the reversible AND gate state table.

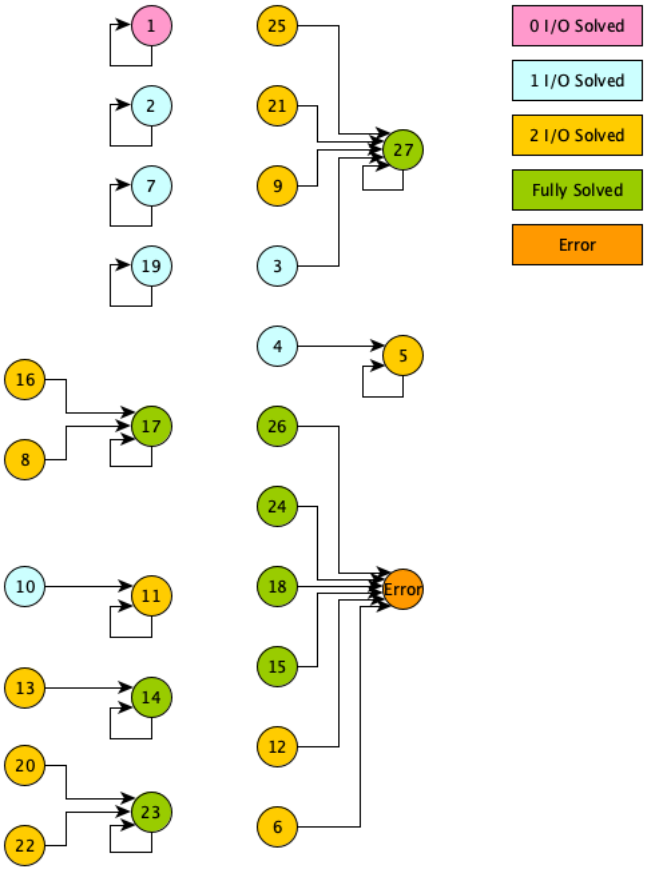

Figure 7: Diagram of Reversible AND State Transitions

With this information, we can begin programmatically implementing the logic of the reversible AND gate. This thesis utilizes Python as the primary programming language for implementation. The code is written in such a way so that this can be ported to a hardware implementation language, such as VHDL, with relative ease. Following is a snippet of Python code for how this reversible AND gate was implemented.

```
def and_gate(a, b, z):
    if (a, b, z) == ('X', 'X', 'X'):
        return {
            "gates": ('X', 'X', 'X'),
            "error": '0',
            "change": '0',
        }
    elif (a, b, z) == ('X', 'X', '0'):
        return {
            "gates": ('X', 'X', '0'),
            "error": '0',
            "change": '0',
        }
    # .
    # . The other 24 possible states would be here.
    # .
    elif (a, b, z) == ('1', '1', '1'):
        return {
            "gates": ('1', '1', '1'),
            "error": '0',
            "change": '0',
        }
    else:
        return {
            "gates": ('X', 'X', 'X'),
            "error": '1',
            "change": '1',
        }
```

Figure 8: Code Snippet of Reversible AND Gate

As it may be noticeable from the snippet, the reversible AND gate is treated as a function with three input arguments, *a* and *b* are the inputs to the AND gate and *z* is the output of the AND gate. The code will match the three arguments to a given state in the table and return the new state of the AND gate. Also returned are two flags. One flag is for announcing any errors and the other flag is for announcing if the bits have changed. These additional pieces of information will be useful when constructing larger digital circuits.

Now that we have an implementation of the reversible AND gate; the OR gate and XOR gate will follow suit.

3.1.2 Reversible OR Gate

The same process that was used for the reversible AND gate will be used for the reversible OR gate. Following is a state table for the reversible OR gate, where *X* is an unknown value.

State and Information at a given time				**Deduced State and Information**			
State #	**Input A**	**Input B**	**Output A OR B**	**State #**	**Input A**	**Input B**	**Output A OR B**
1	X	X	X	1	X	X	X
2	X	X	0	14	0	0	0
3	X	X	1	3	X	X	1
4	X	0	X	4	X	0	X
5	X	0	0	14	0	0	0
6	X	0	1	24	1	0	1
7	X	1	X	9	X	1	1
8	X	1	0	Error	X	X	X
9	X	1	1	9	X	1	1
10	0	X	X	10	0	X	X
11	0	X	0	14	0	0	0
12	0	X	1	18	0	1	1
13	0	0	X	14	0	0	0
14	0	0	0	14	0	0	0
15	0	0	1	Error	X	X	X

State and Information at a given time				Deduced State and Information			
State #	**Input A**	**Input B**	**Output A OR B**	**State #**	**Input A**	**Input B**	**Output A OR B**
16	0	1	X	18	0	1	1
17	0	1	0	Error	X	X	X
18	0	1	1	18	0	1	1
19	1	X	X	21	1	X	1
20	1	X	0	Error	X	X	X
21	1	X	1	21	1	X	1
22	1	0	X	24	1	0	1
23	1	0	0	Error	X	X	X
24	1	0	1	24	1	0	1
25	1	1	X	27	1	1	1
26	1	1	0	Error	X	X	X
27	1	1	1	27	1	1	1

Table 6: Reversible OR Gate State Table

Following is a graphical representation of the reversible OR gate state table.

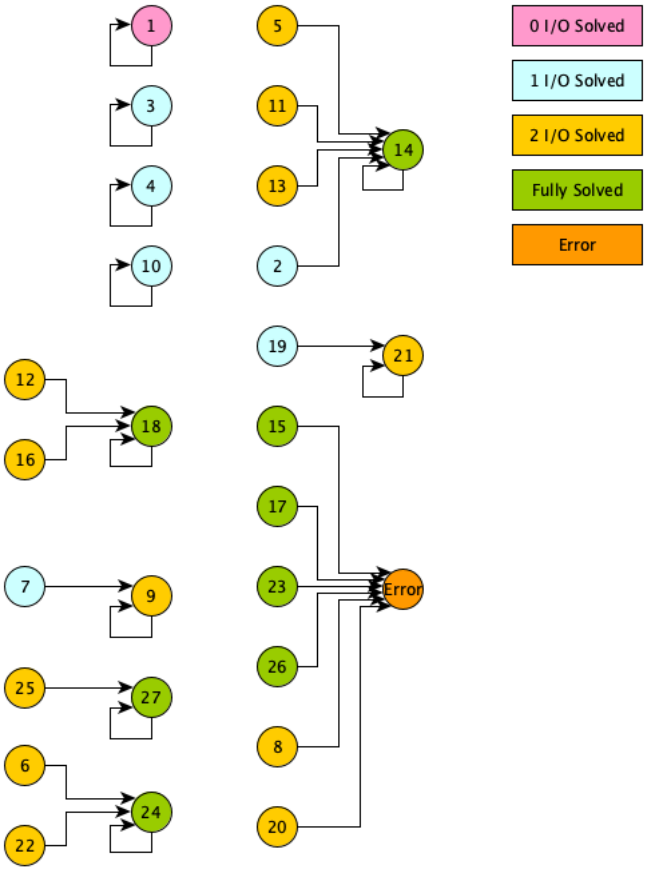

Figure 9: Diagram of Reversible OR State Transitions

With the various states defined, we can programmatically implement the reversible OR gate in Python. Following is a snippet of code for how the reversible OR gate is implemented.

```
def or_gate(a, b, z):
    if (a, b, z) == ('X', 'X', 'X'):
        return {
            "gates": ('X', 'X', 'X'),
            "error": '0',
            "change": '0',
        }
    elif (a, b, z) == ('X', 'X', '0'):
        return {
            "gates": ('0', '0', '0'),
            "error": '0',
            "change": '1',
        }
    # .
    # . The other 24 possible states would be here.
    # .
    elif (a, b, z) == ('1', '1', '1'):
        return {
            "gates": ('1', '1', '1'),
            "error": '0',
            "change": '0',
        }
    else:
        return {
            "gates": ('X', 'X', 'X'),
            "error": '1',
            "change": '1',
        }
```

Figure 10: Code Snippet for Reversible OR Gate

3.1.3 Reversible XOR Gate

Continuing the same process for the XOR gate, we can formulate the various states of inputs/output and how they relate to each other.

State and Information at a given time				**Deduced State and Information**			
State #	**Input A**	**Input B**	**Output A XOR B**	**State #**	**Input A**	**Input B**	**Output A XOR B**
1	X	X	X	1	X	X	X
2	X	X	0	2	X	X	0
3	X	X	1	3	X	X	1

State and Information at a given time				Deduced State and Information			
State #	**Input A**	**Input B**	**Output A XOR B**	**State #**	**Input A**	**Input B**	**Output A XOR B**
4	X	0	X	4	X	0	X
5	X	0	0	14	0	0	0
6	X	0	1	24	1	0	1
7	X	1	X	7	X	1	X
8	X	1	0	26	1	1	0
9	X	1	1	18	0	1	1
10	0	X	X	10	0	X	X
11	0	X	0	14	0	0	0
12	0	X	1	18	0	1	1
13	0	0	X	14	0	0	0
14	0	0	0	14	0	0	0
15	0	0	1	Error	X	X	X
16	0	1	X	18	0	1	1
17	0	1	0	Error	X	X	X
18	0	1	1	18	0	1	1
19	1	X	X	19	1	X	X
20	1	X	0	26	1	1	0
21	1	X	1	24	1	0	1
22	1	0	X	24	1	0	1

State and Information at a given time				Deduced State and Information			
State #	**Input A**	**Input B**	**Output A XOR B**	**State #**	**Input A**	**Input B**	**Output A XOR B**
23	1	0	0	Error	X	X	X
24	1	0	1	24	1	0	1
25	1	1	X	26	1	1	0
26	1	1	0	26	1	1	0
27	1	1	1	Error	X	X	X

Table 7: Reversible XOR Gate State Table

Following is the graphical representation of the reversible XOR gate state table.

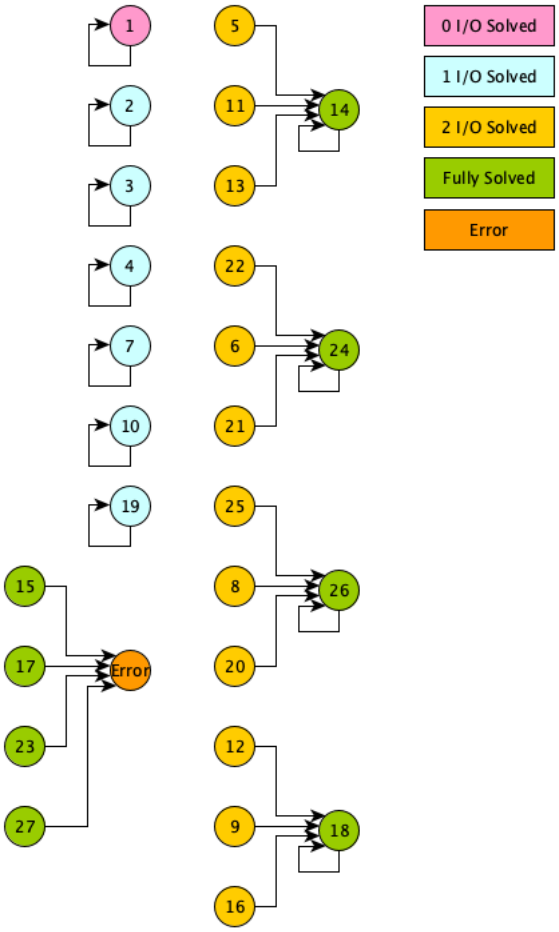

Figure 11: Diagram of Reversible XOR State Transitions

Much like the previous two gates, these state transitions can be represented programmatically. Following is a code snippet on how the XOR gate is implemented.

```
def xor_gate(a, b, z):
    if (a, b, z) == ('X', 'X', 'X'):
        return {
            "gates": ('X', 'X', 'X'),
            "error": '0',
            "change": '0',
        }
    elif (a, b, z) == ('X', 'X', '0'):
        return {
            "gates": ('X', 'X', '0'),
            "error": '0',
            "change": '0',
        }
    # .
    # . The other 24 possible states would be here.
    # .
    elif (a, b, z) == ('1', '1', '1'):
        return {
            "gates": ('X', 'X', 'X'),
            "error": '1',
            "change": '1',
        }
    else:
        return {
            "gates": ('X', 'X', 'X'),
            "error": '1',
            "change": '1',
        }
```

Figure 12: Code Snippet for Reversible XOR Gate

3.2 Reversible Digital Circuits

With the AND gate, OR gate, and XOR gate fully defined to deduce inputs/output based on any given state of the gate, we can proceed to develop more complex digital circuits that utilize these reversible logic gates. Like developing any complex system, it is best to try and tackle smaller problems and build up until the system goals are achieved. We will take this approach in developing the reversible array multiplier.

3.2.1 Reversible Half-Adder

The most fundamental digital circuit that we can build using the reversible logic gates is the half-adder. The half-adder is nearly identical to the full-adder, except that it does not have a Carry-In bit. This reduces the complexity of the full-adder down to two

inputs and two outputs, which results in a total of 81 possible states. The number of possible states is calculated by the following formula:

$$3^{((number\ of\ inputs)+(number\ of\ outputs)}$$

Figure 13: Formula for the Total Possible States

Since the number of states will grow exponentially as more inputs and outputs are identified, we will want to start with smaller circuits. Part of the exercise for this thesis is to determine how much information in digital circuits can be resolved with only using the heavily defined reversible logic gates. Therefore, we will programmatically develop instances of digital circuits that reference the reversible logic gates or other digital circuits. By using Python, we can use object-oriented programming practices for structuring the system. Then, we can use unit testing to verify that the digital circuits are behaving properly.

Since this is the first digital circuit to design, this thesis will go into more detail on the implementation of the reversible half-adder. Future sections will have fewer details since the method for implementation will be similar.

First, we will have to import the reversible logic gates that were defined in the previous section. Then, we will create a class to represent the reversible full-adder. Upon initialization, we will store the various bits that compose the reversible full-adder in an array (list) called *bits*. After initialization, we will declare a *solve* function that all reversible digital circuits will have. When called upon, the *solve* function will iterate through each logic gate and digital circuit that make up the new digital circuit and attempt to deduce information. Once all information has been deduced and there are no

further changes, the *solve* function will return the final deduced information. Following is the majority of the Python code that was used to create the reversible half-adder.

```
from .gates import and_gate, xor_gate
class HalfAdder:
    def __init__(self, bits=list("XXXX")):
        self.bits = bits
        # Default "XXXX"
        #          |||└ output Carry
        #          ||└─ output Sum
        #          |└── input B
        #          └─── input A

    def solve(self, change='0'):
        error = '0'

        if self.bits == ['X', 'X', '0', '0']:
            # Special Case 1
            return {
                "bits": ('0', '0', '0', '0'),
                "error": '0', "change": '1',
            }
        elif self.bits == ['X', 'X', '1', 'X']:
            # Special Case 2
            return {
                "bits": ('X', 'X', '1', '0'),
                "error": '0', "change": '1',
            }

        result = and_gate(
            self.bits[0], self.bits[1], self.bits[3]
        )
        if result["error"] == '0':
            if result["change"] == '1':
                self.bits[0] = result["gates"][0]
                self.bits[1] = result["gates"][1]
                self.bits[3] = result["gates"][2]
                change = '1'
        else:
            error = '1'
        # .
        # . Repeat for xor_gate
        # .
        return {
            "bits": tuple(self.bits),
            "error": error,
            "change": change,
        }
```

Figure 14: Reversible Half-Adder Code

It may be noticeable when reading the code snippet, but there are two special cases that are covered in the *solve* function. The first case is when both inputs are unknown (*X*) and the outputs are *0*. This is a special case because, when the output of the AND gate is 0 we know that one of the inputs are *0*, we just don't know which one. We also know that when the output of the XOR is *0*, then both of the inputs are equal. Therefore, we can deduce that both inputs are *0*. The second case is when both inputs are unknown and only the sum bit is known and the sum bit is *1*. We can deduce that the carry-out bit is *0* because no combination of inputs would result in both sum bit and carry-out bits to be *1* at the same time.

With the *solve* function created, we can use unit testing to perform checks that the function is performing as expected. Following is a code snippet of the unit tests.

```
from .HalfAdder import HalfAdder

def test_ha_case_XXXX():
    assert HalfAdder(['X', 'X', 'X', 'X']).solve() == (
        {
            "bits": ('X', 'X', 'X', 'X'),
            "error": '0',
            "change": "0",
        }
    )

def test_ha_case_XXX0():
    assert HalfAdder(['X', 'X', 'X', '0']).solve() == (
        {
            "bits": ('X', 'X', 'X', '0'),
            "error": '0',
            "change": "0",
        }
    )

# .
# . The other 78 unit tests would go here
# .

def test_ha_case_1111():
    assert HalfAdder(['1', '1', '1', '1']).solve() == (
        {
            "bits": ('X', 'X', 'X', 'X'),
            "error": '1',
            "change": "1",
        }
    )
```

Figure 15: Code Snippet of Unit Tests for Reversible Half-Adder

3.2.2 Reversible Full-Adder

With the reversible half-adder defined, we will build on top of it in order to create the reversible full-adder. The same organizational structure is used for programming the reversible full-adder. Since most of the Python code is very similar to the half-adder but tailored to the full-adder, we will focus on the special cases that arose when developing the full-adder.

```
# .
# . Rest of code to handle initialization
# .
def solve(self, change='0'):
    if (
        self.bits[0] == 'X' and self.bits[1] == '1'
        and self.bits[2] == 'X' and self.bits[6] == '0'
        and self.bits[7] == 'X'
    ):
        # Special case 1
        self.bits[7] = '1'
        return {...}
    elif (
        self.bits[0] == 'X' and self.bits[1] == '1'
        and self.bits[2] == '1' and self.bits[6] == 'X'
        and self.bits[7] == 'X'
    ):
        # Special case 2
        self.bits[7] = '1'
        return {...}
    elif (
        self.bits[0] == '1' and self.bits[1] == 'X'
        and self.bits[2] == 'X' and self.bits[6] == '0'
        and self.bits[7] == 'X'
    ):
        # Special case 3
        self.bits[7] = '1'
        return {...}
    elif (
        self.bits[0] == '1' and self.bits[1] == 'X'
        and self.bits[2] == '1' and self.bits[6] == 'X'
        and self.bits[7] == 'X'
    ):
        # Special case 4
        self.bits[7] = '1'
        return {...}
# .
# . Rest of code to handle solving
# .
```

Figure 16: Code Snippet of Special Cases for Full-Adder

Interestingly enough, all four special cases for the full-adder has the full-adder's carry-out bit resolving to *1*. In the first case and third case, one of the inputs is *1*, the sum output is *0*, and the rest are unknown (*X*). With this information, we can deduce that the

carry-out bit must be *1* since at least one of the inputs is *1*. In the second case and fourth case, two of the inputs are *1* and the rest are unknown (*X*). With this information, we can deduce that we will have a carry-out bit of *1*, since the sum of the inputs will at least be of decimal value 2 or greater.

To verify everything is working correctly, unit tests were created, similar to that of the half-adder, to account for all 243 possible combinations of inputs and outputs and the ability to deduce information.

3.2.3 Reversible Array Multiplier Cell

With the full-adder designed and implemented, the reversible array multiplier cell followed suit. Based on how everything has been implemented thus far, the reversible array multiplier cell experienced no special cases when performing the *solve* function. Therefore, the implementation of the reversible array multiplier cell was very straight-forward and had no exceptions to the standard procedure performed in prior sections.

Similar to before, unit tests were created to verify that all 729 possible combinations of inputs and outputs were able to deduce information properly. This reversible array multiplier cell will be used in the construction of the array multipliers mentioned in Chapter II and the results of the implementation will be reviewed in Chapter IV.

All Python code for this chapter and other chapters of the thesis will be available, via a link, in the Appendix.

CHAPTER IV: RESULTS AND DISCUSSIONS

In this chapter, we will analyze the results of the reversible logic gates and reversible digital circuits. First, we will analyze the ability of these systems to generate new information. Then, we will analyze the ability of an assembled system to factor a semiprime number. Throughout the analysis, there will be a discussion of the results.

4.1 Information Generation Study

The purpose of this section is to dive into each of the reversible logic gates and each type of reversible digital circuits. The outcome will be an understanding of what new information will be generated by the various systems.

4.1.1 Information Impact of Logic Gates

When analyzing the logic gates, we can analyze the AND gate and OR gate together. The reason being is the information generated by them is nearly identical. The similarities can be seen when comparing Figure 7 and Figure 9. Both gates have the same types of transitions, ignoring the specific state identifiers.

With that, the AND gate and OR gate experience interesting results for generating information. For any given state of them, there is a 22% chance that the state could be an error. This would only happen if at least two I/O's are defined at the same time. If all I/O is unknown, the AND gate and OR gate will remain in that state. If one of the I/O is defined, there is a 50% chance that at least one more I/O will be deduced. If two of the I/O are defined, and we ignore the states that are errors due to misconfiguration, there is an 80% chance that the gate will be fully defined. Lastly, there is a 62% chance that the gate is fully defined or can be fully defined, given a gate with random inputs that does not

produce an error. With this information, we can generate a table to have a better view of the results.

Chance of…	**Percentage (%)**
Error, given random input	22
Deducing at least 1 I/O, given 1 I/O	50
Deducing all I/O, given 2 I/O, ignore errors	80
Deducing all I/O, given random, ignore errors	62

Table 8: Information Generation from Reversible AND/OR Gates

Now that we have gone through the exercise of analyzing the AND gate and OR gate, we can perform the same analysis on the XOR gate. The analysis will result in the following table.

Chance of…	**Percentage (%)**
Error, given random input	15
Deducing at least 1 I/O, given 1 I/O	0
Deducing all I/O, given 2 I/O, ignore errors	100
Deducing all I/O, given random, ignore errors	70

Table 9: Information Generation from Reversible XOR Gate

It is interesting to compare the results of these reversible logic gates. We can see that the AND/OR gates are better at producing some form of information regardless of the number of I/O defined. However, the XOR gate cannot deduce any information if only one I/O is defined. If more than one I/O is defined, the XOR will generate all I/O.

4.1.2 Information Impact of Digital Circuits

In this section, we will dive into the information generated by the various reversible circuits discussed in Section 3.2. In addition, we will look at the effectiveness when implementing them and how effective they are at deducing information.

Following is a table that breaks down the reversible digital circuits. The chance of error considers random values being assigned to all I/O of the digital circuit and determining the likelihood of the digital circuit being in an erroneous state. The chance of generating new information is calculated by dividing the following quantity, the number of possible states minus the number of error states minus the fully solved states, into the number of states that result in a change that is not an error. I/O states faulting without intervention was mentioned previously in Section 3.2, where we look at how many special cases were needed for the digital circuit to behave as expected.

Reversible Digital Circuit Type	**I/O Count**	**Chance of Error**	**Chance of Generating New Information**	**I/O States Faulting without Intervention**
Half-Adder	4	38.3%	82.6%	97.5%
Full-Adder	5	31.3%	73.0%	98.4%
Array Multiplier Cell	6	31.3%	70.1%	100%

Table 10: Information Generation of Reversible Digital Circuits

It is interesting to note how the chance of error does not change when going from the Full-Adder to Array Multiplier Cell. Also, as the I/O count increases the ability to generate new information decreases. This will be important for the ability to factorize the semiprime numbers because as the semiprime number grows, the complexity and I/O

count of the array multiplier will grow, likely resulting in less information being able to generate.

4.2 Feasibility of Factoring Semiprime Numbers

In this section, we will study how feasible it is to perform factoring of semiprimes numbers using the method described in this thesis. We will cover how the ability to generate information changes with the size of the semiprime number.

In order to perform the deduction of information, the reversible multiplier array has to be constructed. Following is a snippet of the Python code used to generate the multiplier array and solve each multiplier cell.

```
def solve(self, change='0'):
    error = '0'
    for row in range(len(self.input_b)):
        for column in range(len(self.input_a)):
            if row == 0 and column == 0:
                # Top Right Corner
                result = MultiplierCell(
                    [
                        self.input_a[column], self.input_b[row], '0', '0',
                        self.output[row + column], self.carry[row][column],
                    ]
                ).solve()
                if (
                    result["error"] == '0'
                    and self.output[row + column]
                    == result["bits"][4]
                ):
                    if result["change"] == '1':
                        self.input_a[column] = result["bits"][0]
                        self.input_b[row] = result["bits"][1]
                        self.carry[row][column] = result["bits"][5]
                        change = '1'
                else:
                    error = '1'
                self.sum[row][column] = None
            elif row > 0 and column == 0:
                # First Column
                # . . .
```

Figure 17: Code Snippet of Multiplier Array

Now that we have the ability to generate the multiplier array, we want to make sure it behaves as expected. Therefore, much like the previous digital circuits, we can create unit tests to ensure correct behavior. Following is an example of a test that was created to test the factorization of 143. In order to create these tests, the array multiplier was deduced by hand and the test case was created based on the deduction.

```
def test_143():
    # 13 - '1101'
    # 11 - '1011'
    test = SemiPrimeFactorization(143)
    expected_result = {
        "input_a": "XXX1",
        "input_b": "XXX1",
        "error": '0'
    }
    expected_internals = {
        "input_a": ['1', 'X', 'X', 'X'],
        "input_b": ['1', 'X', 'X', 'X'],
        "sum": [
            [None, 'X', 'X', 'X'],
            [None, 'X', 'X', 'X'],
            [None, 'X', 'X', 'X'],
            [None, None, None, None]
        ],
        "carry": [
            ['0', '0', '0', '0'],
            ['0', 'X', 'X', 'X'],
            ['0', 'X', 'X', 'X'],
            ['0', 'X', 'X', None]
        ],
        "output": "11110001"
    }
    actual_result = test.solve()
    actual_internals = test.internals()
    assert expected_result == actual_result
    assert expected_internals == actual_internals
```

Figure 18: Test for Reversible Array Multiplier

With a fully functioning reversible array multiplier, we can test a vast amount of possibilities for the semiprime number and perform data analysis on how much

information is generated. To generate a wide range of test cases, Python code was used to generate prime numbers for various binary lengths, from 2 bits to 512 bits. The prime numbers were then multiplied and fed into the output of the reversible array multiplier. After the reversible array multiplier deduced all possible information, the results of how much information was deduced were saved into a CSV file. Following is a graph that depicts the information deduced as a function of the size of the semiprime number. The information was generated by sweeping semiprimes of binary length starting at 4 and up to 1024 bits long. When generating the semiprime, two prime numbers of equal binary length were created to ensure symmetry in the array multiplier.

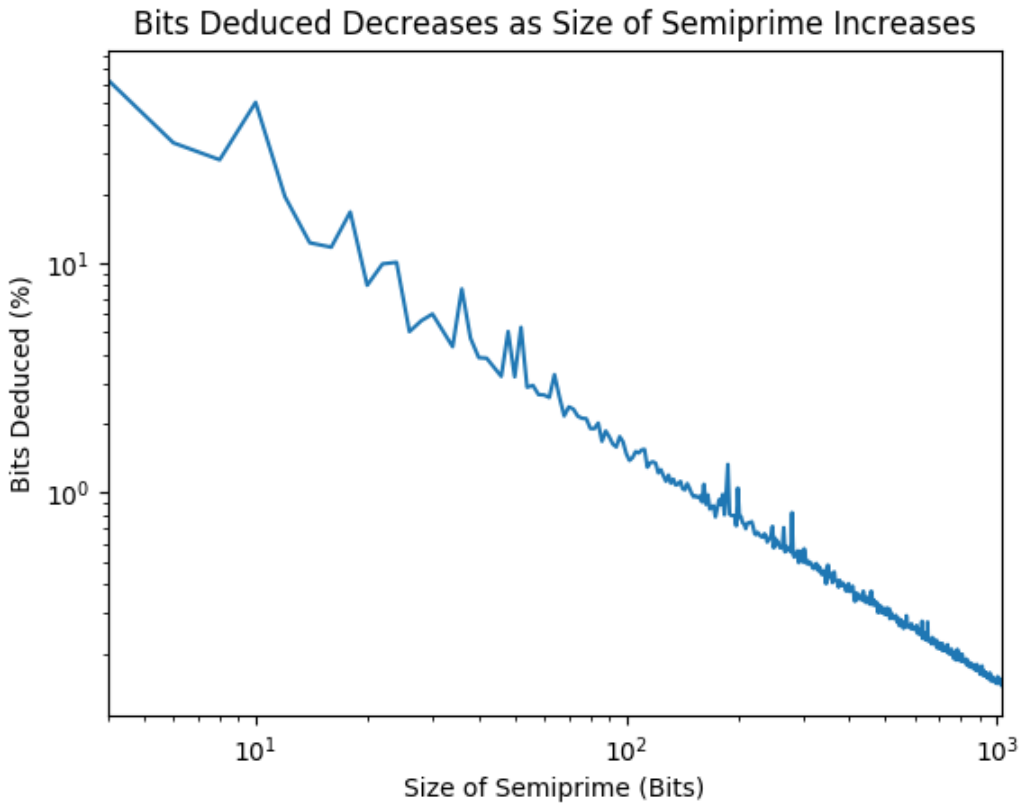

Figure 19: Bits Deduced versus Size of Semiprime

As we can see in the graph trend, the amount of information deduced decreases as the size of the semiprime number increases. Since only one sample was taken for any *n-bit* semiprime number, there is some jaggedness to the graph. However, the trend is very clear. If we were to approximate this trend, it would take the form of the following function, $Bits\ Deduced\ (\%) = 213.62 * (Size\ of\ Semiprime)^{-1.053}$. According to the National Institute of Standards and Technology, the recommended. RSA key length is 2048. [9] Given this information, using the standard RSA-2048, the reversible array multiplier would only generate ~0.07% of the bits in the digital circuit. Based on this, more research can be completed to determine how truly viable reversible logic gates can be.

CHAPTER V: CONCLUSION

Although information can be deduced using reversible logic gates, the approach taken does not appear to be a viable method for factoring semiprime numbers. We tested semiprime numbers of the binary length 4 to 1024. As the binary length of the semiprime increased, the amount of information deduced decreased. Since the amount of information generated diminishes as the size of the semiprime number grows, a different approach must be developed for reversible logic gates to be feasible. One such approach, that may grant a wider depth of information, is developing relationships between bits. As an example, if the output of an XOR gate is *0*, we know that the inputs are either both *1*'s or both *0*'s. Knowing this fact about XOR gates, we can incorporate a new level of information by applying a relationship to the inputs. If enough of these relationships can be developed, this approach of using reversible logic gates may still be feasible. In addition to more approaches being developed, future research can include analyzing different types of systems and digital circuits to determine various outcomes. Another point of future research is to implement the reversible array multiplier into a more complex factoring algorithm. All of this research can be applied to cybersecurity and the various encryption methodologies. Since all encryption could be implemented in hardware, hardware should be analyzed to the fullest to ensure there are no obscure methods to uncovering the secrets used in encryption.

BIBLIOGRAPHY

[1] R. L. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Commun. ACM,* vol. 21, no. 2, pp. 120-126, February 1978.

[2] E. W. Weisstein, "RSA Number," [Online]. Available: http://mathworld.wolfram.com/RSANumber.html. [Accessed 10 November 2018].

[3] "Sequence A001358," [Online]. Available: https://oeis.org/A001358. [Accessed 10 November 2018].

[4] Internet Archive WaybackMachine, "RSA Laboratories - The RSA Factoring Challenge," 2 May 2007. [Online]. Available: https://web.archive.org/web/20070502031806/ http://www.rsa.com:80/rsalabs/node.asp?id=2092. [Accessed 10 November 2018].

[5] M. Case, "A Beginner's Guide to the General Number Field Sieve," 2003. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.219.2389&rep=rep1&type=pdf. [Accessed 31 March 2019].

[6] T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. A. Osvik, H. t. Riele, A. Timofeev and P. Zimmermann, "Factorization of a 768-bit RSA modulus," Cryptology ePrint Archive, Report 2010/006, 2010.

[7] R. Garipelly, P. Kiran and A. Kumar, "A Review on Reversible Logic Gates and their Implementation," *International Journal of Emerging Technology and Advanced Engineering,* vol. 3, no. 3, pp. 417-423, March 2013.

[8] H. R. Bhagyalakshmi and M. K. Venkatesha, "An Improved Design of a Multiplier Using Reversible Logic Gates," *International Journal of Engineering Science and Technology,* vol. 2, no. 8, pp. 3838-3845, 2010.

[9] E. B. Barker and Q. H. Dang, "Recommendation for Key Management Part 3: Application-Specific Key Management Guidance," National Institute of Standards and Technology, 2015.

APPENDIX: SOURCE CODE ACCESS

Upon completion of this thesis, all source code will be made available at the following URL: https://github.com/pjbollinger/master-thesis