DEFENDING CRYPTOGRAPHY AGAINST QUANTUM ATTACKERS

An Undergraduate Research Scholars Thesis

by

DAVIS SHANE BEILUE

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Fang Song

May 2020

Major: Computer Science

TABLE OF CONTENTS

Page

ABSTRACT 1

DEDICATION 2

ACKNOWLEDGMENTS 3

NOMENCLATURE 4

SECTION

I. INTRODUCTION 5

The Evolution of Cryptography 5
Constructs of Modern Cryptography 6
The Quantum Threat 7
NIST and the Proactive Community 8
The Chosen System 8

II. METHODS 11

Analysis Introduction 11
Runtime Analysis 12
Space Analysis 13
Correctness 13
Known Attackers 14
Device and Implementation 14

III. RESULTS 16

Runtime Results 16
Space Results 18
Correctness Results 19
Attacker Analysis 20

IV. CONCLUSION 24

REFERENCES 25

APPENDIX 26

ABSTRACT

Defending Cryptography Against Quantum Attackers

Davis Shane Beilue
Department of Computer Science & Engineering
Texas A&M University

Research Advisor: Dr. Fang Song
Department of Computer Science & Engineering
Texas A&M University

The world of computing looks to propel itself into a new age with the progression of quantum computers and their growing capabilities. With these new capabilities come new threats, especially in the world of cryptography. Modern encryption schemes are built upon mathematical concepts that have proven difficult for today's computers to solve instances of. However, studies over the past few decades have proven that a fully realized quantum computer will not find all of these things difficult. This means the privacy of every person and entity could be at stake if no preparations are made in anticipation of the fully realized quantum computer. Preparing for this threat means having cryptographic standards understood as much as possible with schemes ready for implementation.

This preparation is being regulated by organizations such as the National Institute of Standards and Technology (NIST) who has requested that research teams all over the world submit their ideas for encryption methods and standards so that the community as a whole can collaborate through analysis, critiquing and learning from them, in turn pushing progress forward at a faster pace. In this paper, I take up this mantle of analysis and will discuss the results from a selected scheme.

DEDICATION

I dedicate this document and the work required for its creation to those who have encouraged me to take every major step that led me here: my mother, father, sister, and the many friends I have met through so many diverse paths. This would not be possible without your prayer and support.

ACKNOWLEDGMENTS

Thank you first and foremost to Dr. Fang Song for giving me this incredible opportunity to stretch myself in a way I never before imagined I could. I greatly appreciate the time you have spent meeting with me and answering countless emails.

Thank you to both the Craig and Galen Brown Engineering Honors program and the Undergraduate Research Scholars thesis program for providing, not only the opportunity, but the necessary resources to make this process as smooth as possible.

NOMENCLATURE

sk	Secret key
pk	Public key
c	Ciphertext
m	Message to be encrypted
m'	Decrypted message
K	Key to be encapsulated
K'	Decapsulated key
NIST	National Institute of Standards and Technology
LAC	Lattice-based Cryptography (cryptosystem)
LWE	Learning with Errors

CHAPTER I

INTRODUCTION

Cryptography has been protecting the world's most sensitive information for thousands of years. Many cereal boxes have had kids reverse the classic "Caesar Cipher" to reveal a hidden message, coined for its usage by Julius Caesar himself when disguising sensitive documents.

The countless hours required to develop a cryptographic scheme would hardly be needed if there were no threats to protect information from; as this threat continues to develop, so must our defenses against it.

Note: discussion of topics in this introduction will be done in a highly reductive manner so that a beginner in this field might quickly grasp the required breadth of knowledge.

The Evolution of Cryptography

Early Cryptography

Modern cryptographers view the encryption methods of far yesteryear as more of an art than the blueprints for effective solutions (Katz and Lindell, 2015). The aforementioned Caesar Cipher involves simply choosing a number x and changing all of the letters in the document to the one x spaces forward in the alphabet. While a ciphertext produced under this method would perhaps be unreadable to an adversary at first glance, simple linguistic analysis has proven highly effective in cracking such codes (Katz and Lindell, 2015).

Many other schemes of this time used simple one-for-one letter substitutions that, though often chosen arbitrarily, remained vulnerable to the same attacks. Even more complex ciphers such as the Vignère, one that relied on using one word as the key to encrypting many, could be attacked effectively by simply learning the length of the keyword. An attacker of this time period

needed so little information to begin putting pieces together: we could not dream of using such schemes with any confidence today (Katz and Lindell, 2015).

Modern Cryptography

The prevailing issues that plagued most early schemes came largely from their lack of rigorous forethought and testing. The modern cryptographer now aims to wield complexity rather than show the beauties of simplicity and our cryptographic schemes find their roots in complex mathematical problems. Functions are used to produce keys that encrypt data so that Alice can safely send information to Bob, and only Bob will have the capability to decrypt that data (and vice-versa). This will be expanded on in the following section.

Constructs of Modern Cryptography

Though there are many important concepts to understand about modern cryptography, it is necessary for the reader to be familiar with the following:

One-Way Functions

A one-way function is usually described as a function that is easy to compute but difficult to reverse. This means that a person can easily use inputs to generate an instance of the function by computation, but an unknowing adversary cannot easily use the results of that instance to find the original inputs (even when they know what the function is). Such functions are foundational for the generation of keys. These functions typically have binary strings as inputs and outputs.

Symmetric Cryptographic Schemes

Alice generates a key k (by use of some one-way function) and shares it (discretely) with her friend Bob. She then uses k to encrypt her message m, generating a ciphertext c. She sends c to Bob who uses the same k on c to regenerate m.

Asymmetric Cryptographic Schemes

Alice uses a key generation algorithm with some input to obtain certain (often numeric) values. These values are used to generate the keypair (*sk*, *pk*). She distributes *pk* publicly so that anyone can use it to encrypt a message meant for her. Alice keeps the secret key, *sk*, for no one but her to know. Bob encrypts message *m* with *pk* to get ciphertext *c*. He sends *c* to Alice who uses *sk* on *c* to regenerate *m*.

An attacker is usually assumed to have access to *pk* and *c* through some form of eavesdropping. The aim is to develop an encryption scheme such that an attacker cannot determine certain information about *sk* (thus giving them access to *m*) in polynomial time.

Most modern cryptographic schemes are of the asymmetric form as distributing keys is far easier, but often a symmetric key will be "encapsulated", or encrypted, using an asymmetric public key and sent to the owner of the respective private key. The two parties now both have access to an identical secret key that can be used to quickly encrypt and decrypt larger chunks of data.

The Quantum Threat

Current encryption schemes rely on the fact that the mathematical problems they are based on are difficult for the modern classical computer to solve in polynomial time. Research is currently being done in the realm of quantum computers, which would theoretically have the computing power to solve these problems far more efficiently (Shor, 1995). This means that most cryptographic schemes that are currently seen as secure would no longer be that way when attackers have quantum capabilities.

Action must now be taken to ensure that information remains secure. Encryption schemes need to be developed such that they are able to be run on a classical computer but are also

difficult for quantum attackers to solve. The task is a large one, but there are many who are striving to find the answer.

NIST and the Proactive Community

No quantum computer built at this point in time has the capabilities required to break current encryption in polynomial time, but since research continues to progress the field, we must ensure we are ready for the day a quantum attacker reaches full power.

NIST has sent a call out to the cryptographic community for researchers to submit their ideas for the newly necessary schemes. The purpose of this preemptive call is to ensure that we have the necessary security standards properly set and implemented across the internet before the day of full realization (NIST, 2017). While some submit schemes, many others analyze and critique them through multiple rounds of submissions, ensuring that any standards that are set have received thorough scrutiny: this is our purpose here.

The Chosen System

I have chosen to analyze a system titled LAC (Lattice-Based Cryptography). This system employs lattices as a basis for its operations. Lattice cryptosystems utilize concepts such as the Shortest Vector Problem (SVP) and Learning with Errors (LWE) as a basis to make it difficult for adversaries to reverse encryption unwantedly. I chose this system as my knowledge of linear algebra allowed me to grasp the concept of lattices much more firmly than some of the other bases. Below, I will explain some of the more important overarching features of the system, while staying somewhat out of the weeds.

Lattices

A lattice can be "defined as the set of all linear combinations

$$L(\mathrm{b}_1,\ldots,\mathrm{b}_n) = \{\sum_{i=1}^{n} x_i \mathrm{b}_i : x_i \in \mathbb{Z} \text{ for } 1 \le i \le n\}$$

of n linearly independent vectors $\mathrm{b}_1,\ldots,\mathrm{b}_n$ in $\mathbb{R}^n$ (Micciancio, 2009)." As an example, consider the two-dimensional plane of integer numbers ($\mathbb{Z}^2$). In this case, your "basis" vectors, b_1 and b_2 could be

$$\mathrm{b}_1 = \binom{1}{0}, \mathrm{b}_2 = \binom{0}{1}$$

so that all vectors in $\mathbb{Z}^2$ can be rewritten as a linear combination of the given basis vectors.

Those familiar with linear algebra and related fields will thus far note the apparent similarities between lattices and vector spaces. That which sets the lattice apart from such spaces lies in the more specific and specializes forms of the structures. One of the most significant forms in cryptography is a group known as q-ary lattices. "These are lattices L satisfying $q\mathbb{Z} \subseteq L \subseteq \mathbb{Z}$ for some (possibly prime) integer q (Micciancio, 2009)." Essentially, each element in a given vector of L is an integer multiple of the integer q.

The authors of LAC have chosen to use "a provable secure design from Ring-LWE" (Lu, et al. 2019). LWE stands for Learning with Errors, which as previously mentioned is one of the hard problems associated with lattice-based cryptosystems. The essential steps of key generation in such systems are as follows: first define a q-ary lattice L. Then choose an $n \times l$ matrix S and an $h \times n$ matrix A from L uniformly at random. Next choose an $h \times l$ matrix E according to a centered binomial distribution with some parameter $\sigma \in \mathbb{R}$. You are now left with a keypair (sk; pk) = (S; A, AS+E) (Lu, et al. 2019; Micciancio, 2009).

The encryption process for these schemes involves taking a message m and vector a that has been chosen uniformly at random, then outputting ciphertext (u, c) = ($\mathrm{A}^T\mathrm{a}$, $(\mathrm{AS+E})^T\mathrm{a}+f(\mathrm{m})$)

(Lu, et al. 2019; Micciancio, 2009). To decrypt, compute $f^{-1}(c - S^T u)$, where S is the private key generated in the first step (Lu, et al. 2019; Micciancio, 2009). This style of cryptosystem is believed to be very efficient and fits the mold of being easy to compute difficult to reverse, most importantly difficult on the quantum level.

CHAPTER II

METHODS

Analysis Introduction

In order to ensure the world can continue to safely exchange data at a high rate, we must be certain that any new way of making that data safe can keep the pace while also keeping the integrity of the original message intact. The authors of LAC have provided their own benchmarks that they have garnered from set parameters on a highly capable machine, but that does not necessarily mean those results will translate to our everyday devices.

Categories	Key generation		Encryption		Decryption		Decryption(Const-BCH)	
	CPUCycles	Times	CPUCycles	Times	CPUCycles	Times	CPUCycles	Times
LAC128	61242	19.98	80173	25.91	25004	7.83	64238	20.77
LAC192	120528	38.87	130286	42.34	63266	26.41	134289	39.95
LAC256	136313	54.23	191543	63.14	72326	30.56	112654	48.99

Categories	Key generation		Encryption		Decryption		Decryption(Const-BCH)	
	CPUCycles	Times	CPUCycles	Times	CPUCycles	Times	CPUCycles	Times
LAC128	61242	19.98	80173	25.91	25004	7.83	64238	20.77
LAC192	120528	38.87	130286	42.34	63266	26.41	134289	39.95
LAC256	136313	54.23	191543	63.14	72326	30.56	112654	48.99

Figure 1. Example benchmarks from LAC documentation (Lu et al., 2019).

The typical order of steps in pure asymmetric cryptography can be found above. All of these steps are crucial in the process and, as can be glimpsed in the above tables, are tested separately. My testing will consist of the following steps:

1. Generate keypairs with total lengths of 128, 192, and 256 bits
2. Encrypt messages (*m*) of varying size with each public key
3. Decrypt ciphertexts with secret key and retrieve messages (*m'*)

The order of steps when using asymmetric principles to encapsulate symmetric keys is largely similar:

1. Generate keypairs with total lengths of 128, 192, and 256 bits
2. Encapsulate some key *K* with each public key
3. Decapsulate ciphertexts and retrieve key (*K'*)

The key lengths of 128, 192, and 256 bits are not chosen arbitrarily, but rather at the suggestion of the authors (Lu et al., 2019). In the following sections, I will detail what specific analysis will take place at each of the above steps.

Runtime Analysis

My personal runtime tests will be recorded in tables identical to those seen in Figure 1. This means that I will be recording the times taken at each step of the encryption/encapsulation processes. My results will be compared to those given by the authors as well as the standards previously set by modern cryptosystems to assess any discrepancies. The times recorded are the

average of 10,000 separate operations and is measured in microseconds. All tests will be run three times, and the displayed results will be the average of the three.

Space Analysis

Before the key generation step, users must run a “Makefile” to prepare the code for execution. This creates certain files that allow for the execution of the functions. In order to test different key sizes, the code will have to be “remade” with the different sizes specified in the parameter files. The question is, how much do the different key sizes affect the overall size of the system, if at all? These results will again be compared to current cryptographic benchmarks, and will follow the following format:

Table 1. Empty size table.

	Size Before	On Disk	Size After	On Disk
LAC128				
LAC192				
LAC256				

Note: size will be measured in MB.

Correctness

It is imperative that, when sending information, the message encrypted by the sender is indeed the message decrypted by the recipient. In each encryption test, the encrypted *m* will be compared with the decrypted *m'* to detect discrepancies. In the ideal scenario, there will be none.

Known Attackers

The authors have identified a multitude of attack methods that have proven capable of "solving" the various hard problems that form the basis for lattice-based schemes, thus breaking encryption (for a specific instance, not all encryption). This fact is not necessarily worrisome on its own, as such attacks exist for all standardized cryptosystems. However, it is imperative to ensure that these attacks cannot succeed in polynomial time, otherwise, the scheme is near useless.

As has been discussed, LAC is a candidate for post-quantum cryptography, so the intended adversaries are expected to have quantum capabilities. I do not have anything of the sort at my disposal, so my analysis of attacks will consist mainly of rudimentary attempts of the attacks that the authors have found significant enough to include in their documentation. This will hopefully open the door for a meaningful conclusion to be drawn regarding these specific methods of attacks.

Device and Implementation

Device Specifications

The device used to conduct these tests is a Dell Precision 5520. It uses an Intel® Core™ i7-7820HQ CPU @ 2.90GHz and has 8 GB of RAM. The authors of LAC used a device with an Intel® Core™ i7-4770S @ 3.10GHz and 7.6GB of RAM (Lu et al., 2019).

Implementation Specifics

The authors included duplicate tests using an "Optimized Implementation" and an implementation utilizing the Advanced Vector Extensions 2 (AVX2) instruction set. The main difference in these implementations seems to be how they handle polynomial multiplication, which is what the authors describe as "the most time-consuming operation in the implementation

of LAC" (Lu et al., 2019). My computer's processor is capable of using the AVX2 instruction set (the implementation which produces the faster results of the two), but many older processors are not. Seeing as my focus is to test performance on less capable machines, my tests will utilize the "Optimized Implementation."

CHAPTER III

RESULTS

Runtime Results

The following tables are the results from encryption tests run by myself and the authors. My tests appear in the blue boxes, the authors' in the yellow boxes, and the white boxes are the result of (my results / their results). Note that all shown times are measured in microseconds (µs).

Table 2. My encryption tests.

	Key Generation		Encryption		Decryption	
Mine	CPU Cycles	Time	CPU Cycles	Time	CPU Cycles	Time
LAC128	196596	48.4375	288626.6667	73.54167	128040	33.33333
LAC192	480684	170.3125	577328.6667	231.25	408098.6667	142.1875
LAC256	518213	175.5208	930115.6667	278.6458	439416.3333	139.5833

Table 3. Their encryption tests.

	Key Generation		Encryption		Decryption	
Theirs	CPU Cycles	Time	CPU Cycles	Time	CPU Cycles	Time
LAC128	124915	40.28	194118	67.24	81187	26.28
LAC192	335083	106.2	438204	144.63	292243	93.8
LAC256	382627	124.23	636997	204.8	302890	95.18

(Lu et al., 2019)

Table 4. Size of my values compared to the authors'.

	Key Generation		Encryption		Decryption	
	CPU Cycles	Time	CPU Cycles	Time	CPU Cycles	Time
LAC128	1.57383821	1.20252	1.486861943	1.093719	1.577099782	1.268392
LAC192	1.434522193	1.603696	1.317488354	1.598908	1.396436071	1.515858
LAC256	1.354355547	1.41287	1.46015706	1.360575	1.450745595	1.46652

Comparing the results of these tests is not surprising: my computer requires more time and CPU cycles to accomplish the same tasks as the authors' device. However, some analysis will reveal that my tests never reach times or cycles twice as large as the authors' (Table 4).

The trends of runtime found in encryption/decryption largely continue for encapsulation/decapsulation:

Table 5. My encapsulation tests.

	Key Generation		Encapsulation		Decapsulation	
Mine	CPU Cycles	Time	CPU Cycles	Time	CPU Cycles	Time
LAC128	197344	47.39583	287463.6667	79.16667	363815	104.6875
LAC192	476290.6667	158.3333	661235.3333	223.4375	753254.3333	371.3542
LAC256	573756.6667	171.875	978157.6667	301.0417	1343048.667	394.2708

Table 6. Their encapsulation tests.

	Key Generation		Encryption		Decryption	
Theirs	CPU Cycles	Time	CPU Cycles	Time	CPU Cycles	Time
LAC128	122691	39.67	209201	65.71	280125	88.07
LAC192	333649	105.63	445696	145.48	731472	235.42
LAC256	377123	123.59	643024	208.71	916835	297.01

(Lu et al., 2019)

Table 7. Size of my values compared to the authors'.

Key Generation		Encapsulation		Decapsulation	
CPU Cycles	Time	CPU Cycles	Time	CPU Cycles	Time
1.608463538	1.194753	1.374102737	1.204789	1.298759482	1.188685
1.427520138	1.498943	1.483601678	1.535864	1.029778766	1.577411
1.521404599	1.390687	1.521183761	1.442392	1.464874996	1.327467

Space Results

Some of these tests required my own custom files to be added to the implementation, which in turn affected the overall size of the folder. As a result, it was decided to include the results from tests with and without these files. Note that all shown sizes are in kilobytes (KB).

Table 8. With my custom function.

	Size Before	On Disk	Size After	On Disk
LAC128	257	308	469	548
LAC192	257	308	452	532
LAC256	257	308	469	548

Table 9. Without my custom function.

	Size Before	On Disk	Size After	On Disk
LAC128	254	304	463	540
LAC192	254	304	447	524
LAC256	254	304	464	540

The sizes are fairly small by today's standards which allows this to fit on most any modern machine. One interesting thing to note is that the 128-bit implementation takes up more room than the 192-bit implementation, and nearly always as much as the 256-bit implementation. It is unclear to me at the moment whether this leads to a meaningful conclusion, but the discovery of this fact is at least worth mentioning.

Correctness Results

After generating 1 keypair, 1000 messages were randomly generated, encrypted, and then decrypted (hex outputs, 128 implementation):

m	2115A78259B2DD38D3DBC36F2C62BCA7E24CBE225765E1685802E89627AC26B
p	2115A78259B2DD38D3DBC36F2C62BCA7E24CBE225765E1685802E89627AC26B
m	D437CCE91ED1ED896EB4BEFC95F82C4A6C33DB812CF3C01B7551CD43B76BE
p	D437CCE91ED1ED896EB4BEFC95F82C4A6C33DB812CF3C01B7551CD43B76BE
m	FFBE3BC8BE47BEE228B2E81F99717A84D44FE5EB24BBB7C588A973E872F82C3
p	FFBE3BC8BE47BEE228B2E81F99717A84D44FE5EB24BBB7C588A973E872F82C3
m	C1243FFBD0336F4A838FFDC2D2BFA8F98E4C63A0E546676D8F4328A862AE284D
p	C1243FFBD0336F4A838FFDC2D2BFA8F98E4C63A0E546676D8F4328A862AE284D
m	FC4BCDA36F9BD7AA4A14222AB37E59ED61897855296B1AEEC09838ACADEDCD
p	FC4BCDA36F9BD7AA4A14222AB37E59ED61897855296B1AEEC09838ACADEDCD
m	54A54D9EA0E8D06519DA19BC9892C537A1B1F9315EDB31993C631DF9AB1DB
p	54A54D9EA0E8D06519DA19BC9892C537A1B1F9315EDB31993C631DF9AB1DB
m	E45E1EB737B92368F8D5583BBEC33584F97F8C69A69D2194A7A9675CBA98C
p	E45E1EB737B92368F8D5583BBEC33584F97F8C69A69D2194A7A9675CBA98C
m	281CDDF013878FC32EA21E1A52DA74B461544418244154C9F36CFF85754
p	281CDDF013878FC32EA21E1A52DA74B461544418244154C9F36CFF85754
m	A0C5D5B79C3BA07047962E5871587B13D671944EAA8A8D7141FE9187F4A7E9BD
p	A0C5D5B79C3BA07047962E5871587B13D671944EAA8A8D7141FE9187F4A7E9BD
m	34638AC448F39041542A8A4351275ABE849A27778EC4E1259F7F2F2C88AE
p	34638AC448F39041542A8A4351275ABE849A27778EC4E1259F7F2F2C88AE
m	188DCC6542936F44C72CE9A47A8E53D9C24BF82C3FAE89C6E666B87F2532B7
p	188DCC6542936F44C72CE9A47A8E53D9C24BF82C3FAE89C6E666B87F2532B7
m	6AF3AEB8D16528F98D4D27A17455CC31DCAE4B28ADFD271B5DD9C21D4985B2A
p	6AF3AEB8D16528F98D4D27A17455CC31DCAE4B28ADFD271B5DD9C21D4985B2A
m	33AEC51FE36AF49FCE81970A843CDD6E86794AAF54524F2C592A3268D32F
p	33AEC51FE36AF49FCE81970A843CDD6E86794AAF54524F2C592A3268D32F
m	E19B2B609839DB2C96F37820ABC18A1A4E82C97BA0C858088CD1FB9C61064
p	E19B2B609839DB2C96F37820ABC18A1A4E82C97BA0C858088CD1FB9C61064
m	D6C1AC95E12C62D34CE0809444125540463F49379AAB816DF598D47A494B27C8
p	D6C1AC95E12C62D34CE0809444125540463F49379AAB816DF598D47A494B27C8

Figure 2. Same keypair for each encryption.

There were no discrepancies in any of the message (*m*)/decrypted plaintext (*p*) pairs tested, as can be seen by the limited tests shown above. After these tests, I believed it valuable to do a similar test with some more practical messages (ASCII outputs, 128 implementation):

m	0000AAAAAAACAAAAA$$$$AAAAAAABBBC
p	0000AAAAAAACAAAAA$$$$AAAAAAABBBC
m	This is a test of the LAC system
p	This is a test of the LAC system
m	Correctness is important %_&-=+/
p	Correctness is important %_&-=+/

Figure 3. Same keypair as before.

Note that the same keypair was used for each test. This is to mimic the real-world scenario where a single keypair is generated by a user, *pk* is shared publicly, and many people then use the same *pk* to send messages to the user, who will continually use the same *sk* to decrypt those incoming messages.

Attacker Analysis

The authors have named multiple attacks that have either proven effective against schemes of a similar nature, or that were proposed by those who analyzed their Round 1 submission of the NIST call. The following is analysis of two particular attacks that I believe to be interesting.

Primal Attack

The primal attack seeks to "[build] a lattice with a unique-SVP instance from the LWE samples; then, [using] BKZ algorithm to recover this unique shortest vector (Lu et al., 2019)." In

essence, attacks of this nature seem to attempt to exploit very fundamental building blocks of the scheme. Thus far, the authors have upheld the case for LAC's security against these and others that build off of the primal attack. I take particular interest in these "generic attacks" because, if they prove to be efficient, they would not only compromise the security of LAC, but they would seemingly cause issues for many LWE-based cryptosystems, which has the potential of eliminating an entire subfield of contenders from the wider field that hope to become part of the new standards.

High Hamming Weight Attack

One of the other attacks identified by the authors takes advantage of something called Hamming weight, which is essentially the number of nonzero symbols that appear in a given string. In the case of LAC, those strings are "the secrets and errors ($\boldsymbol{r}, \boldsymbol{e}_1$) in some ciphertexts" (Lu et al., 2019). The goal of an attack is to exploit higher Hamming weights to guess the decryption error rate in a given system, then use that information in an attempt to gain more knowledge of the private key (Lu et al, 2019).

The authors reference a specific implementation of such an attack in their own attack analysis, and I attempted my own tests with it (D'Anvers et al., 2018). Using the software provided by D'Anvers et al. in their paper (referenced below), I obtained the following:

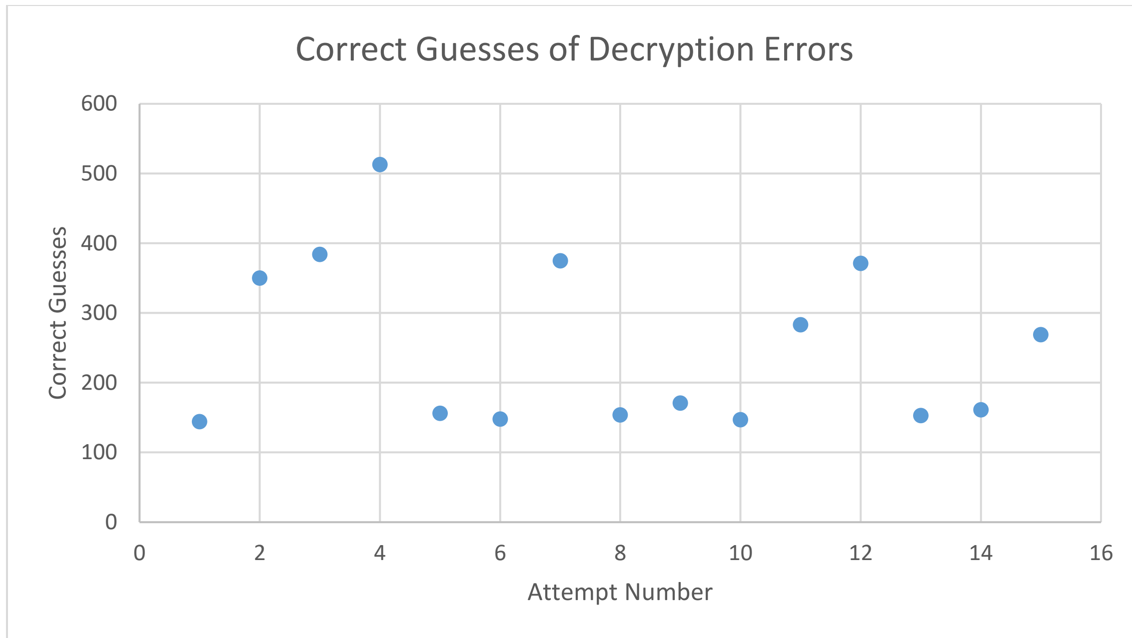

Figure 4. Attack results on the 256-bit implementation.

The test was done using the same keypair as the previous ones. An average of 251.93 errors were correctly guessed after 15 tests, where each test attempts to guess 1024 bits. This means that correct guesses were made almost a fourth of the time. This remained somewhat consistent after another 10 tests (where correct guesses were made about 30% of the time), however, during these tests it was discovered that decryption errors were only made around 30% of the time as well. Note that the attack algorithm is able to detect non-error cases, so this is less concerning than it originally appears.

This is important, as any information that an attacker is able to successfully draw out of a ciphertext, key, or any other part of a cryptosystem should be minimal: any amount of information could form a basis for more to be found. Typically, the margin of concern is when an attacker can be correct at a rate of $1/n \pm a$, where n is the number of possible outcomes and a

is some negligible function. It is therefore noteworthy that the rate of errors found is nearing 1/3, as there are only 3 possible bits to guess, those being -1, 0, and 1.

CHAPTER IV

CONCLUSION

The tests above showcase the capabilities of this scheme when run on a machine with average capabilities. The question remains, how does this compare with current schemes? A good comparison can be made with the modern RSA, one of the most widely used cryptosystems of today. Thus far, time constraints have allowed me to conduct only very limited research down this path, so I do not wish to make any decisive statements regarding such comparisons based on the results that I have gathered so far. What I can say is that the fact many in the field believe in the efficiency of the math behind the system coupled with its advancement to round two in NIST's call means it may be one of the best options there are (Micciancio, 2009).

I do believe that my results in testing the speed of the LAC cryptosystem compare well with those given by the authors, and Tables 4 and 7 show great consistency in those results. Its size lends itself to be stored and run on most any device, which is important in today's age of smartphones and tablets, and the apparent correctness is a great boon.

My one concern is with my results from the Hamming weight attack tests: my results seem to imply that this style of attack is a strong candidate for finding a great deal of information on a consistent basis. If I were to give any advice, it would be to further analyze these attacks and prepare a better defense against them. The authors spent well over a page discussing this attack in their latest submission, so it is clearly on their minds, but they seem to claim they are "immune" to such methods of attack (Lu et al., 2019). I believe this issue requires further investigation.

REFERENCES

Katz, Jonathan, and Yehuda Lindell. Introduction to Modern Cryptography. CRC Press/Taylor & Francis, 2015.

NIST "Call for Proposals." CSRC, 3 Jan. 2017, csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals.

Shor, Peter W. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." SIAM Journal on Computing 26.5 (1997): 1484–1509. Crossref. Web.

Lu, Xianhui, et al. "Practical Ring-LWE Based Public-Key Encryption with Byte-Level Modulus." Cryptology EPrint Archive, vol. 2018, 2019.

Micciancio, Daniele. "Lattice-Based Cryptography." *Post-Quantum Cryptography*, by Oded Regev, Springer, Berlin, Heidelberg, 2009, pp. 147–191.

D'Anvers, Jan-Pieter, et al. "On the Impact of Decryption Failures on the Security of LWE/LWR Based Schemes." Cryptology EPrint Archive, 2018.

APPENDIX

Figure 1. Example benchmarks from LAC documentation (Lu et al., 2019).

Categories	Key generation		Encryption		Decryption		Decryption(Const-BCH)	
	CPUCycles	Times	CPUCycles	Times	CPUCycles	Times	CPUCycles	Times
LAC128	61242	19.98	80173	25.91	25004	7.83	64238	20.77
LAC192	120528	38.87	130286	42.34	63266	26.41	134289	39.95
LAC256	136313	54.23	191543	63.14	72326	30.56	112654	48.99

Categories	Key generation		Encryption		Decryption		Decryption(Const-BCH)	
	CPUCycles	Times	CPUCycles	Times	CPUCycles	Times	CPUCycles	Times
LAC128	61242	19.98	80173	25.91	25004	7.83	64238	20.77
LAC192	120528	38.87	130286	42.34	63266	26.41	134289	39.95
LAC256	136313	54.23	191543	63.14	72326	30.56	112654	48.99

Figure 2. Same keypair for each encryption.

m	2115A78259B2DD38D3DBC36F2C62BCA7E24CBE225765E1685802E89627AC26B
p	2115A78259B2DD38D3DBC36F2C62BCA7E24CBE225765E1685802E89627AC26B
m	D437CCE91ED1ED896EB4BEFC95F82C4A6C33DB812CF3C01B7551CD43B76BE
p	D437CCE91ED1ED896EB4BEFC95F82C4A6C33DB812CF3C01B7551CD43B76BE
m	FFBE3BC8BE47BEE228B2E81F99717A84D44FE5EB24BBB7C588A973E872F82C3
p	FFBE3BC8BE47BEE228B2E81F99717A84D44FE5EB24BBB7C588A973E872F82C3
m	C1243FFBD0336F4A838FFDC2D2BFA8F98E4C63A0E546676D8F4328A862AE284D
p	C1243FFBD0336F4A838FFDC2D2BFA8F98E4C63A0E546676D8F4328A862AE284D
m	FC4BCDA36F9BD7AA4A14222AB37E59ED61897855296B1AEEC09838ACADEDCD
p	FC4BCDA36F9BD7AA4A14222AB37E59ED61897855296B1AEEC09838ACADEDCD
m	54A54D9EA0E8D06519DA19BC9892C537A1B1F9315EDB31993C631DF9AB1DB
p	54A54D9EA0E8D06519DA19BC9892C537A1B1F9315EDB31993C631DF9AB1DB
m	E45E1EB737B92368F8D5583BBEC33584F97F8C69A69D2194A7A9675CBA98C
p	E45E1EB737B92368F8D5583BBEC33584F97F8C69A69D2194A7A9675CBA98C
m	281CDDF013878FC32EA21E1A52DA74B461544418244154C9F36CFF85754
p	281CDDF013878FC32EA21E1A52DA74B461544418244154C9F36CFF85754
m	A0C5D5B79C3BA07047962E5871587B13D671944EAA8A8D7141FE9187F4A7E9BD
p	A0C5D5B79C3BA07047962E5871587B13D671944EAA8A8D7141FE9187F4A7E9BD
m	34638AC448F39041542A8A4351275ABE849A27778EC4E1259F7F2F2C88AE
p	34638AC448F39041542A8A4351275ABE849A27778EC4E1259F7F2F2C88AE
m	188DCC6542936F44C72CE9A47A8E53D9C24BF82C3FAE89C6E666B87F2532B7
p	188DCC6542936F44C72CE9A47A8E53D9C24BF82C3FAE89C6E666B87F2532B7
m	6AF3AEB8D16528F98D4D27A17455CC31DCAE4B28ADFD271B5DD9C21D4985B2A
p	6AF3AEB8D16528F98D4D27A17455CC31DCAE4B28ADFD271B5DD9C21D4985B2A
m	33AEC51FE36AF49FCE81970A843CDD6E86794AAF54524F2C592A3268D32F
p	33AEC51FE36AF49FCE81970A843CDD6E86794AAF54524F2C592A3268D32F
m	E19B2B609839DB2C96F37820ABC18A1A4E82C97BA0C858088CD1FB9C61064
p	E19B2B609839DB2C96F37820ABC18A1A4E82C97BA0C858088CD1FB9C61064
m	D6C1AC95E12C62D34CE0809444125540463F49379AAB816DF598D47A494B27C8
p	D6C1AC95E12C62D34CE0809444125540463F49379AAB816DF598D47A494B27C8

Figure 3. Same keypair as before.

m:	0000AAAAAAACAAAAA$$$$AAAAAAABBBC
p:	0000AAAAAAACAAAAA$$$$AAAAAAABBBC
m:	This is a test of the LAC system
p:	This is a test of the LAC system
m:	Correctness is important %_&-=+/
p:	Correctness is important %_&-=+/

Figure 4. Attack results on the 256-bit implementation.

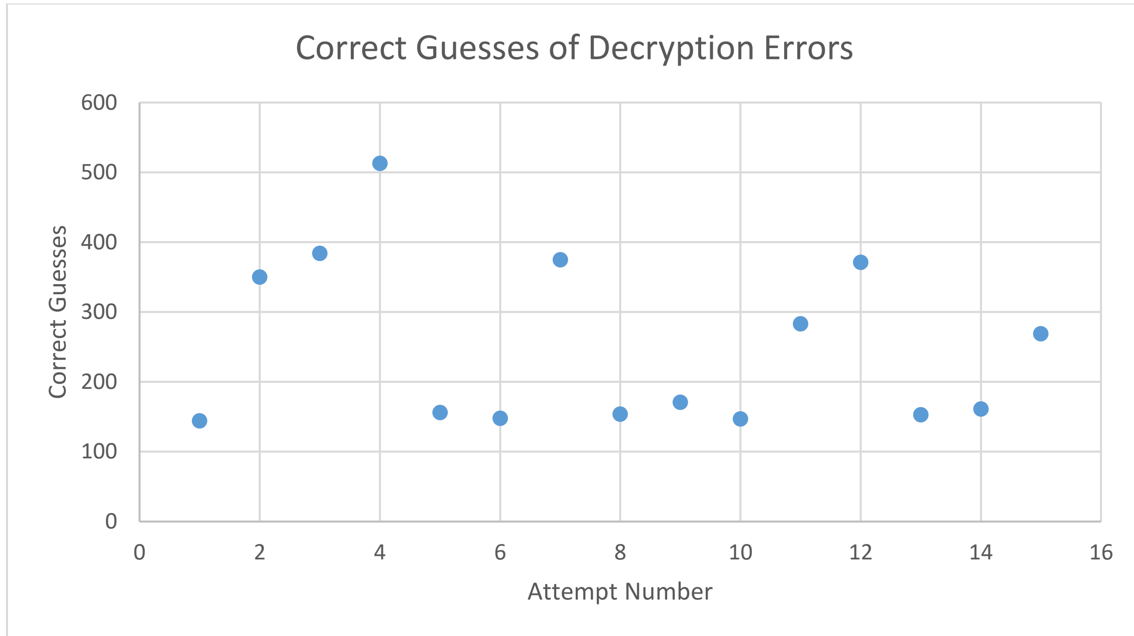

Table 1. Empty Size Table.

	Size Before	On Disk	Size After	On Disk
LAC128				
LAC192				
LAC256				

Table 2. My encryption tests.

	Key Generation		Encryption		Decryption	
Mine	CPU Cycles	Time	CPU Cycles	Time	CPU Cycles	Time
LAC128	196596	48.4375	288626.6667	73.54167	128040	33.33333
LAC192	480684	170.3125	577328.6667	231.25	408098.6667	142.1875
LAC256	518213	175.5208	930115.6667	278.6458	439416.3333	139.5833

Table 3. Their encryption tests.

	Key Generation		Encryption		Decryption	
Theirs	CPU Cycles	Time	CPU Cycles	Time	CPU Cycles	Time
LAC128	124915	40.28	194118	67.24	81187	26.28
LAC192	335083	106.2	438204	144.63	292243	93.8
LAC256	382627	124.23	636997	204.8	302890	95.18

Table 4. Size of my values compared to the authors'.

	Key Generation		Encryption		Decryption	
	CPU Cycles	Time	CPU Cycles	Time	CPU Cycles	Time
LAC128	1.57383821	1.20252	1.486861943	1.093719	1.577099782	1.268392
LAC192	1.434522193	1.603696	1.317488354	1.598908	1.396436071	1.515858
LAC256	1.354355547	1.41287	1.46015706	1.360575	1.450745595	1.46652

Table 5. My encapsulation tests.

	Key Generation		Encapsulation		Decapsulation	
Mine	CPU Cycles	Time	CPU Cycles	Time	CPU Cycles	Time
LAC128	197344	47.39583	287463.6667	79.16667	363815	104.6875
LAC192	476290.6667	158.3333	661235.3333	223.4375	753254.3333	371.3542
LAC256	573756.6667	171.875	978157.6667	301.0417	1343048.667	394.2708

Table 6. Their encapsulation tests.

	Key Generation		Encryption		Decryption	
Theirs	CPU Cycles	Time	CPU Cycles	Time	CPU Cycles	Time
LAC128	122691	39.67	209201	65.71	280125	88.07
LAC192	333649	105.63	445696	145.48	731472	235.42
LAC256	377123	123.59	643024	208.71	916835	297.01

Table 7. Size of my values compared to the authors'.

Key Generation		Encapsulation		Decapsulation	
CPU Cycles	Time	CPU Cycles	Time	CPU Cycles	Time
1.608463538	1.194753	1.374102737	1.204789	1.298759482	1.188685
1.427520138	1.498943	1.483601678	1.535864	1.029778766	1.577411
1.521404599	1.390687	1.521183761	1.442392	1.464874996	1.327467

Table 8. With my custom function.

	Size Before	On Disk	Size After	On Disk
LAC128	257	308	469	548
LAC192	257	308	452	532
LAC256	257	308	469	548

Table 9. Without my custom function.

	Size Before	On Disk	Size After	On Disk
LAC128	254	304	463	540
LAC192	254	304	447	524
LAC256	254	304	464	540