Spring 2022

Investigating Lattice-Based Cryptography

Michaela Molina
San Jose State University

Investigating Lattice-Based Cryptography

A Project Report

Presented to
Dr. Chris Pollett

Department of Computer Science
San José State University

In Partial Fulfillment
Of the Requirements for the Class
CS 298

By
Michaela Molina
May 2022

The Designated Project Committee Approves the Master's Project Titled
Investigating Lattice-Based Cryptography

By

Michaela Molina

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE
SAN JOSÉ STATE UNIVERSITY
MAY 2022

Dr. Christopher Pollett	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Dr. Chris Pollett for his patience, encouragement, and knowledge during this challenging project. Thank you for spending your valuable time helping me each week for over a year and for answering my emails even on the weekends. I would also like to thank Dr. Thomas Austin and Dr. Robert Chun for their time and effort spent thoughtfully evaluating my project. I would like to thank the Computer Science Department faculty for the classes they taught. And finally I would like to thank my parents for funding my education.

ABSTRACT

Cryptography is important for data confidentiality, integrity, and authentication. Public key cryptosystems allow for the encryption and decryption of data using two different keys, one that is public and one that is private. This is beneficial because there is no need to securely distribute a secret key. However, the development of quantum computers implies that many public-key cryptosystems for which security depends on the hardness of solving math problems will no longer be secure. It is important to develop systems that have harder math problems which cannot be solved by a quantum computer.

In this project, two public-key cryptosystems which are candidates for quantum-resistance were implemented using Rust. The security of the McEliece system is based on the hardness of decoding a linear code which is an NP-hard problem, and the security of the Regev system is based off of the Learning with Errors problem which is as hard as several worst-case lattice problems [1], [2]. Tests were run to verify the correctness of the implemented systems and experiments were run to analyze the cost of replacing pre-quantum systems with post-quantum systems.

Index terms: Cryptography, Post-Quantum, Public-Key

TABLE OF CONTENTS

List of Figures

LIST OF TABLES

List of Graphs

1. INTRODUCTION

The communication of sensitive information has been an important task since before the digital age. Plaintext information that is sent from one party to another through an unsecured channel is easily subject to interception and is readable, manipulatable, and forgeable by any malicious party. Forms of cryptography have been used since the need first arose and provide a way to encode information so that malicious parties cannot read it, tamper with it, or forge it. Today, the need to secure information is even more prevalent and is present in everything from mundane applications such as text messaging to highly sensitive satellite communications. In particular, public key cryptography can be used for confidentiality by encrypting data with a public key and then sending it so that only the holder of the private key can decrypt it. The same system can also be used to verify integrity by sending a plaintext data along with a data encrypted with the private key or private key and having them compared at the receiving end. Authentication can be achieved by encrypting data with the private key so that anyone with the public key will know that only the holder of the private key could have sent it.

The security of a public-key cryptosystem depends on the difficulty of its underlying math problem. A harder math problem implies a more secure system. The development of quantum computers has meant that the difficulty of some of these math problems has become threatened by more powerful computers which can solve the problems easily. For example, Shor's algorithm is a quantum algorithm that can solve the factoring problem, which RSA is based on, in polynomial time. This will cause problems especially if quantum computers become mainstream. Therefore it has become important to develop cryptosystems which have security that holds even under attack from a quantum computer. The goal of this project was to investigate and implement some such systems. To achieve this, we used Rust to implement two public-key systems which are candidates for quantum-resistance. The first was the McEliece public key system in which security is based on the hardness of decoding a linear code which is NP-hard [1]. The second was the Regev public key system in which security is based off of the Learning with Errors problem which is as hard as several worst-case lattice problems [2]. We then ran experiments on these two systems and a library version of RSA, a non-quantum resistant public key system, to compare key size, key generation speed, encryption speed, decryption speed, and encrypted message size in pre-quantum versus post-quantum public key systems. We made graphs and analyzed the cost of replacing current systems with systems equipped for the post-quantum future.

The report is organized as follows: in Chapter 2 we discuss some related work. In Chapter 3 we explain the preliminary work necessary to understand the importance of the project. In Chapter 4 we give the technical background of the project. In Chapter 5 we go over the cryptosystem steps. In Chapter 6 we explain our implementation of each system. In Chapter 7 we go over an example of each system. In Chapter 8 we show the tests we did to make sure each system was correct. In Chapter 9 we go over our experiments. In Chapter 10 we mention some challenges, in Chapter 11 we give ideas for future work, and in Chapter 12 we give our conclusion.

2. Related Work

In [42] Valentijn provides a comprehensive overview of Goppa Codes and their use in the McEliece cryptosystem. She provides the mathematical background of linear codes and goes over the steps in creating and using the McEliece cryptosystem. She also gives explanations of the security of the system and names several attacks. In this project we use her partial examples of key generation, encryption, and decryption as a guideline. In [55] Berlekamp provides a summary of Valerii Denisovich Goppa's original work in which he introduces the new class of linear noncyclic error-correcting codes, Goppa codes, and proves four important properties about them. In [56] Patterson gives his algorithm for the algebraic decoding of Goppa Codes which we use in our McEliece decryption implementation. And in [57] Singh gives a thorough report of the mathematical background of linear codes, code-based cryptography, the McEliece cryptosystem, and the Niederreiter Cryptosystem, a variant of McEliece which can be used for digital signatures.

3. PRELIMINARY WORK

Before discussing our implementation we explain why our project is important. In this section we discuss symmetric keys, the brute-force search, public keys, quantum computers, and post-quantum cryptography. Many of our examples in this section come from [9].

3.1 Symmetric Keys: An Ancient History

Four thousand years ago in ancient Egypt, in a cemetery called Beni Hasan, a nobleman was buried by his family of powerful monarchs [3]. The tomb of Khnumhotep II drew the eye even in the midst of the magnificent necropolis; a courtyard and portico welcomed visitors, and once they were inside, an intricate ceiling curved gracefully [4]. Most spectacularly, a painting adorned the walls depicting Asiatic nomadic traders bringing offerings to the dead [4]. But the rock around the doorway possessed a unique feature-- its surface was smooth and flat, and it bore a fourteen-line inscription [4]. Some believe these hieroglyphs to be the earliest known example of a substitution cipher [5, 6, 7].

While an ancient tomb may be debatable as the first ever instance of cryptography, many more examples crop up throughout history that are less controversial. In 700 BC, over a thousand years after Khnumhotep II, Spartan generals used a tool called a scytalae to encrypt messages [5,7]. A strip of paper was wrapped around a cylinder and the message was written across the different strips [5,7]. When separated from the cylinder, the letters were scrambled, and only the general with an exact copy of the cylinder would be able to recover the message [5,7]. Five hundred years later, Polybius, another Greek, devised the Polybius checkerboard to represent a single letter with a pair of numbers [8]. Only a party who had the checkerboard could decode the message [8]. One hundred years after Polybius, Julius Caesar used the now well-known Caesar Cipher: every letter in a message was substituted by the letter some number of places ahead in the alphabet [6,8,9]. What do all of these systems have in common? A secret key.

The privacy of the covert activities of the Spartans, Polybius, and Julius Caesar depended on their secret keys staying secret. If an attacker discovered a secret key, the attacker would be able to read, tamper with, or forge any message the attacker desired. It makes sense that the invested parties would want to make discovering the secret key as difficult as possible. To accomplish this one might first consider how an attacker would try to find the secret key: assuming that all other aspects of the system were secure, such as implementation details and choice of honest individuals, the most obvious way to do so would be to guess one key at a time, try deciphering the message, and if the results made sense, the key was probably correct. Thus if the attacker is capable of trying all possible keys, the attacker is guaranteed to find the right one. This is called a *brute-force attack* or an *exhaustive key search*, and the number of all possible keys is called the *keyspace* [9]. The larger the keyspace, the more possibilities there are for the secret key, and the longer the attacker has to guess before they find it. Thus the larger the keyspace the higher the security of the system. In the case of Julius Caesar, a brute-force attacker would try shifting the letters of the message by one through twenty-six places in the alphabet; in the case of Polybius, a brute-force attacker would generate all possible checkerboards of numbers; and in the case of the Spartans, a brute-force attacker might go for a hike and collect as many different twigs and sticks as they could carry. If the forest was large, the last attacker may indeed face the largest keyspace and thus the hardest job.

Forest notwithstanding, we will now take a moment to demonstrate the concept of a keyspace using a version of Caesar's cipher, the substitution cipher, as an example [9]. In the simple substitution cipher, each letter can be assigned a unique number of places to be shifted by. In other words, each letter of the alphabet can be assigned another letter of the alphabet, with each letter having only one letter that maps to it. This equates to a permutation of the alphabet: and since there are twenty-six letters available, and if we start at the beginning of the alphabet, the number of options for "a" to be mapped to is twenty-six, the number of options for "b" to be mapped to is twenty-three, the number of options for "c" to be mapped to is twenty-two, and so on until the number of options for "z" to be mapped to is one. This creates $26 \cdot 25 \cdot 24 \cdot \ldots \cdot 1 = 26! \approx 403291460000000000000000000 \approx 2^{88}$ possible permutations of the alphabet, giving a keyspace of 2^{88}. Again assuming that no other attacks besides the brute-force search are possible, guessing the key in the time of Julius Caesar would certainly be a task. However, to bring the idea of the keyspace to the modern day, first assume that we have a modern 4.0 GHz computer as in [9]. Such a computer can test about 250 million$\approx 2^{28}$ keys per second [9]. If we divide $\frac{2^{88}\ possible\ keys}{\frac{2^{28}\ keys}{second}}$ we get $2^{88} keys \cdot \frac{1\ second}{2^{28} keys} = \frac{2^{88}}{2^{28}} seconds = 2^{60} seconds \approx 36534658200$ years to perform an exhaustive key search over this keyspace of size 2^{88}. Since we can expect to find the key after searching half of the keyspace [9], using our personal computer we can expect to find the key in 18267329100 years, which is certainly infeasible. We will keep the ideas of keyspace and brute-force search in mind as we introduce symmetric key systems developed in less ancient history.

While we might never know whether the hiker ever found a suitable piece of detritus, we can bring our discussion into more relevant territory. Systems such as the Caesar Cipher, the Spartan Scytalae, and Polybius' Checkerboard continued to be developed into the modern era and are called *symmetric ciphers* or *symmetric key cryptosystems* because the same key is used for encrypting and decrypting. For example, the One-Time Pad, or OTP, was used by Soviet spies in the 1930s to send messages from the United States back to Moscow [9]. The OTP key was a random string of bits the same length as the message; encrypting and decrypting was accomplished by a simple XOR, meaning that the keyspace was exponential in the size of the message [9]. Another example is the codebook cipher, which involved a dictionary that mapped plaintext words to ciphertext words and was used in World War I. Only a party with the same codebook could decrypt the message, making the keyspace equal to all possible codebooks.

We must also start to consider other attacks on the systems besides the brute-force attack. All symmetric key systems employ either *confusion* or *diffusion* or both in order to work [9]. Confusion hides the relationship between the plaintext and the ciphertext, while diffusion distributes the plaintext statistics throughout the ciphertext [9]. The Caesar Cipher employed confusion, while the Spartans used diffusion. These two concepts are as relevant today as they were in the time of Julius Caesar; this is because other than the keyspace, the security of a symmetric key system depends on the effectiveness of the confusion and diffusion. We say that a cryptosystem is secure if the best-known attack requires as much work as an exhaustive key search [9]. So, if the confusion and diffusion mechanisms are not sufficient, their weaknesses may provide an easier way than the exhaustive key search to discover the plaintext, the key, or both [9]. This brings us to DES, which was a symmetric cipher standardized in

1977. DES is a block cipher, more specifically, a Feistel cipher, in which security depends on a function called a round function that is applied multiple times to a block of plaintext with the secret key and the previous output of the function as input. Most importantly, the round function in DES was so effective that no attack was developed that was faster than an exhaustive key search [9]. DES was retired in 2005 since it had a small keyspace that had become vulnerable to brute-force attacks by the increasing capabilities of commodity computers. Two more symmetric key systems that are still used today followed in the footsteps of DES: Triple DES was invented in 1981 and AES was invented in 1997, with key sizes of 112 and 128, 192, or 256 bits respectively [9]. Most importantly, the confusion and diffusion techniques of these ciphers are effective enough that no attack has been found that threatens their structure itself, leaving the best-known attack to be the brute force search. Recall that our simple substitution cipher had a keyspace of 2^{88} which could be searched successfully in 18267329100 years by a modern computer; by contrast, a 256-bit AES key has a keyspace of 2^{256}, which by similar computations could be searched successfully in about 1620393009164736827524337348641 years. Ronald Rivest, a famous cryptographer who will come up later in our story, has been heard to say that he believes a 256-bit AES key will be "secure forever" regardless of advancing technology [9].

So far we have established that Triple DES and AES have keyspaces of 2^{112} and 2^{256} respectively and their respective structures are strong enough that the fastest attack is still the exhaustive key search, meaning that the fastest attack would take well over 18267329100 years. The practical efficiency of 3DES and AES is respectable [9], and AES has the endorsement of an esteemed expert in the field. These ciphers seem too good to be true: what could be the problem? There is one so-called "achilles heel" of these ciphers, and that is how to securely distribute the symmetric key [9]. In the modern day, choosing a trustworthy individual to personally carry it is not an option.

3.2 Public Keys: A Necessarily Recent History

Public key cryptography was proposed by Whitfield Diffie and Martin Hellman in 1976 when they proposed a key exchange algorithm that would solve the above problem [9]. In the common definition, a public-key cryptosystem has a public key, which is available to the public, that is used for encrypting, and a private key, which is kept secret, that is used for decrypting. Since the public key is public, there is no need for secure distribution. However, generally speaking, any system which involves some important information being available to the public is considered a public key system [9], as is the case with the Diffie-Hellman key exchange, which does not actually have encryption and decryption capabilities. The Diffie-Hellman Key Exchange works as follows. Choose a prime p and a generator g that generates the prime field $\{1,2,\dots,p-1\}$. The public key is (p,g). To exchange a private key, the first party chooses an integer a and the second party chooses an integer b. The first party sends the quantity $g^a mod\ p$ to the second party, and the second party sends the quantity $g^b mod\ p$ to the first party. Thanks to the exponent rules, the first party can compute $(g^b mod\ p)^a = g^{ab} mod\ p$ and the second party can compute $(g^a mod\ p)^b = g^{ab} mod\ p$. The value $g^{ab} mod\ p$ is the secret key. Consider how an attacker might try to obtain the secret key: Since $g^a mod\ p$ and $g^b mod\ p$ are sent in the clear, the attacker can easily compute $g^{a+b} mod\ p$, but it does not equal $g^{ab} mod\ p$. Instead it appears that the attacker must find either a or b so as to compute $g^{ab} mod\ p$. Given the quantitites $g^a mod\ p$ and $g^b mod\ p$, finding a or b is called the discrete logarithm problem, and as far as

people know it is a very difficult problem to solve. However, the difficulty of the discrete algorithm problem is not known to be NP-complete, a class of problems in which we suppose that the best attack would be brute-force, and so the security of the Diffie Hellman Key Exchange interestingly cannot be quantified in the way that say, the security of a 256-bit AES system can, in which the structural integrity of the cipher itself is so strong that it is assumable that the fastest attack is to exhaustively search the 2^{256} keys in the keyspace. The strength of the discrete logarithm problem is thus of a different nature than the strength of the structure of AES, which does not involve a math problem. This same peculiarity applies to the next system we will discuss.

In 1977, shortly after Diffie and Hellman's key exchange system, a public-key system was invented that could encrypt and decrypt data. It was called RSA, after its inventors Rivest, Shamir, and Adleman and it works as follows. Choose two large primes p and q and set $N = pq$. Then choose e such that $\gcd(e, (p-1)(q-1)) = 1$ and find $d = e^{-1} mod\ (p-1)(q-1)$. The public key is (N, e) and the private key is d. Encryption of a message m is accomplished by doing $c = m^e mod\ N$ and decryption is accomplished by doing $c^d mod\ N$. How would an attacker try to get the private key of this system? Since the private key d is computed using the factors of N, the attacker can obtain it by factoring N. Integer factorization, like the discrete logarithm problem, is known to be hard but is not proven to be NP-complete; this means that like the discrete logarithm problem, the security of the factoring problem is not mathematically proven and still depends on no faster solution than a brute-force private key search being found. Thus like the security of the Diffie-Hellman key exchange, the security of RSA cannot be quantified in the same way as AES.

Finally, we reiterate difference between symmetric and asymmetric keys. In the former, other than large keyspace, the security of the system rests on the effectiveness of the confusion and diffusion tactics. In the latter, other than large keyspace, the security of the system rests on the underlying math problem having no fast solution. In the case of the symmetric system, there is no apparent way in which the integrity of the structure would suddenly become compromised; in the case of the public key systems and their math problems which might have a fast solution, this is not necessarily so. This is important to keep in mind when we later explore quantum computers.

3.3 Cryptography in Real Life: NIST Standards Today

Today, public key systems are used prolifically throughout digital security. While symmetric key systems are significantly more efficient than public key systems, public key systems are vital for symmetric key distribution and digital signatures [9]. The National Institute of Standards and Technology, or NIST, is a United States government agency that produces guidelines and standards that help federal agencies meet the requirements of the Federal Information Security Management Act, or FISMA [15]. FISMA is a United States federal law that requires federal agencies to meet certain security standards in order to protect national security interests [16]. In conjunction with FISMA, NIST develops Federal Information Processing Standards, or FIPS, which federal agencies must comply with, as well as the Special Publications 800-series, or (SP), which provides guidance and recommendations [15]. By looking at NIST standards we can understand how important public-key systems are today. For example, in FIPS 186-4 Digital Signature Standard, NIST specifies only three algorithms, DSA, RSA, and ECDSA [17]. DSA is a variant of the Schnorr and El Gamal signature schemes, which are based on the discrete logarithm problem; RSA

is the method we discussed above; and ECDSA is a variant of the DSA that uses elliptic curve cryptography [18, 19, 20, 21]. Under pair-wise key establishment schemes, NIST lists SP 800-71, in which key establishment using symmetric block ciphers is discussed; SP 800-56A Revision 3, which recommends several variations of the Diffie-Hellman scheme as well as the Menezes-Qu-Vanstone scheme which is based on Diffie-Hellman; and SP 800-56B Revision 2, which specifies key-establishment schemes using RSA [22, 23, 24, 25]. It is clear from these publications that digital signatures and key establishment today--and also national security--are dependent on public-key systems.

3.4 Quantum Computers: A Threat?

Contrary to what one might believe, the capabilities of quantum computers extend past just being "faster" computers. Quantum computers use atomic-scale units called qubits that can be simultaneously 0 and 1 [10]. This is in contrast to classical computers, in which a bit can be 0 or 1. The state of a quantum computers qubit being both 0 and 1 is called superposition [10]. In this state, a single qubit can perform two computations in parallel, which means that computations are much more efficient than in a classical computer [10]. While the immediate threat presented by such a computer may seem like its capability of performing a faster key search, this is not the case: Bennet, Bernstein, Brassard and Vazirani showed that quantum computation cannot speed up a brute-force search by more than a quadratic factor [11]. This means that AES, as long as its best known attack remains such a search, remains safe [12]; according to ComputerWorld, a quantum computer would be able guess a 256-bit key in the same time it takes for a regular computer to guess a 128 bit key, about 133478124979082724000 years [11]. For public key systems, this is not the case. In 1994 Peter Shor invented a special algorithm for quantum computers that could find the prime factors of an integer in polynomial time [13]. The algorithm, called Shor's algorithm, uses the quantum Fourier transform, a linear transformation on quantum bits, to accomplish this and can also be used to solve the discrete logarithm problem [13]. Recall that the security of the Diffie Hellman Key Exchange and RSA depends on the intractability of the discrete logarithm and integer factorization problems, which were only secure as long as no attack faster than brute-force attack was found. Well, the attack faster than brute force has been found. So is that it for public key systems? Will we have to manually deliver our symmetric keys like the ancient Romans? Will digital signatures become a thing of the past? Also not the case. The largest number Shor's algorithm has been able to successfully factor is 21 in 2012: this is because as more qubits are added to a quantum system, it becomes more difficult to control and the number of errors accumulates [10, 13]. Breaking RSA would reportedly take a quantum computer with 20 million qubits 8 hours, and to give some context, Google's Sycamore processor, created by Google's Artificial Intelligence division, has 53 qubits [14]. According to an article written in 2020, researchers estimate it will take between a decade and two decades to create a quantum computer of the required size [14]. Thus, while we may sleep soundly tonight knowing RSA is safe, we cannot deny that there is an urgent need to develop quantum-resistant public key algorithms so that we may continue to distribute symmetric keys in a timely fashion and digitally sign important documents.

3.5 Post-Quantum Cryptography: Preparing for the Future

How can we prepare for the day when quantum computers are capable of breaking RSA and Diffie-Hellman? The goal of post-quantum cryptography is to find cryptosystems in which security depends on math problems that quantum computers probably cannot solve. Currently there are six different approaches to post-quantum cryptography: lattice-based cryptography, multivariate cryptography, hash-based cryptography, code-based cryptography, supersingular elliptic curve isogeny cryptography, and symmetric key quantum resistance [27]. How do we know if the underlying math problem is hard enough? We say that problem two reduces to problem one if problem one is at least as hard to solve as problem two [26]. A proof of this reduction shows that a solution to problem one would imply a solution to problem two [26]. To prove that a math problem is hard enough and that a system is secure, it is necessary show one of these reductions to prove the equivalence of the system's problem with a problem that is already known to be hard [27]. These are called security reductions and they are used to classify the security of the system [27]. In particular, some math problems exist which are conjectured to be unsolvable in polynomial time by a quantum computer. For example, in Lattice-based cryptography the Shortest Vector Problem and the Closest Vector Problem are known to be NP-Hard and can be used to prove the security of some systems [27]. In code-based cryptography the Syndrome Decoding Problem is known to be NP-Hard [27]. Therefore by basing a cryptosystem's security on a problem that has a reduction from one of these problems, we can conjecture that that system is most likely secure even under attack from a quantum computer.

3.6 Two Candidates, One Unlikely

Two candidates for post-quantum cryptography appear in the form of the McEliece public key system and the Regev public key system. The McEliece public key system was developed in 1978 by Robert McEliece, a researcher best known for his work in error-correcting coding and information theory [28]. At the time, the system was pioneering because it was the first asymmetric encryption scheme to use randomization in the encryption step as well as the first linear code-based asymmetric system [28, 29]. The attractiveness of the system was found in the fact that its security was based on the problem of decoding a general linear code, a known NP-hard problem, and thus the fastest known attack by a computer would most likely necessitate a brute-force search [1]. In his original paper, McEliece calls on this property to explain that if the code parameters are large enough, an attack that sought to decode the general linear code by recovering the plaintext from the codeword would indeed be infeasible: for example, in an $n = 2^{10} = 1024$, $t = 50$ linear code, the dimension of the code would be at least $2^m - mt = 1024 - 50 = 524$, meaning that the brute-force attack of comparing the ciphertext to each possible codeword would have a work factor of about 2^{524} [1]. A positive result of this was that the brute-force attack work factor for the system was easy to increase quickly by turning up the parameters of the linear code slightly [1, 29, 30].

While innovative, there were some drawbacks to the system: the public key was inconveniently large, for the parameters mentioned reaching a size of 536576 bits or 67 Kilobytes, whereas today it is desirable to keep public keys under 1 KB; the encrypted message was much longer than the plaintext, which increased the chance of transmission errors; and the cryptosystem could not be used in reverse for authentication or signature, because of the randomness on the encryption side [31]. Because of these reasons, and despite remaining "remarkably stable" in its

resistance of cryptanalysis attempts over the last 40 years, the system never received acceptance in the cryptographic community [29, 30]. However, in 1978 when the system was invented, the field of quantum computing was still in its beginnings: current public-key systems did not come under threat until the early 1990s after several researchers including David Deutsch, Richard Jozsa, Dan Simon, and Peter Shor showed that quantum computers were capable of solving computational problems by using algorithms that regular computers could not [32]. The McEliece cryptosystem received new appreciation after 1995 when the United States Department of Defense organized the first workshop on quantum computing and quantum cryptography [32]. Despite not being intended as a quantum-resistant system, and many years after its emergence, the security of McEliece's system happened to satisfy one of the very requirements that made a system supposedly "quantum-resistant": the Syndrome Decoding Problem on which the system's security depended was a proven NP-Hard problem, meaning that it was one in the class of problems that experts supposed did not have a polynomial-time quantum solution. In this way, the McEliece system happened to have one of the sought-after "security reductions" discussed above. Today, even with researchers actively searching for systems with security reductions in the search for prospective post-quantum cryptography, McEliece's system has its own place as one of only seven such systems listed on Wikipedia's Post-Quantum Cryptography page [27].

The Regev public key system is a somewhat more recent development. It is significant because it is the first "classical", or non-quantum, cryptosystem in which security is based on a quantum hardness assumption [2]. Some discussion follows. In 2005, Oded Regev introduced the Learning With Errors problem, or LWE, and showed a reduction from two worst-case lattice problems, GAVSVP and SIVP, to LWE, once again satisfying the security reduction that is so important in cryptographic applications [1]. Similarly to how McEliece's system was not intended for the quantum-cryptography, the LWE problem derives from an adjacent field in that it is a generalization of the parity learning problem, a problem originating in the field of machine learning [33, 34]. However, unlike McEliece's 1978 system which was introduced with no quantum intentions, Regev's work falls farther into the quantum realm. Specifically, Regev's reduction was a quantum reduction, as opposed to a classical reduction, meaning that a fast algorithm that solves the LWE problem would imply only a quantum algorithm, not a classical algorithm, that solves GAPSVP and SIVP [1]. Regev elaborates more on this, explaining that it is currently conjectured that there is no classical polynomial time algorithm that closely approximates a shortest vector; and that if we take it one step further, so to speak, we can even conjecture that there is also no *quantum* polynomial time algorithm that accomplishes the same feat [1]. What Regev is saying here is that the second conjecture is a somewhat weaker statement because, as we have seen, quantum computers are sometimes capable of carrying out algorithms that classical computers are not. Based on this weaker second conjecture, Regev claims, the LWE problem is hard; and if a solution was found that solved the LWE problem, it would imply a quantum solution, but not a classical solution, to the GAPSVP and SIVP problems. This is the meaning of quantum reduction. However, "weaker" conjecture may be an exaggeration since many experts believe that the conjecture is true, making the LWE problem a solid candidate for a post-quantum system.

Nevertheless, if the reduction were one day made classical, this would make LWE even stronger. This is because proving that a solution to LWE implies a classical solution to SIVP or GAPSVP would mean that quantum

computers cannot exploit any aspect of the problem that regular computers cannot, and must resort to solving the problem in the way that a classical computer would [35]. Finally, one more significant aspect of Regev's proof, mentioned briefly above, is that it can be used conversely to *disprove* the conjecture that no quantum algorithms exist that can solve lattice problems faster than classical algorithms; because if a solution to LWE is found, then a quantum algorithm for solving GAPSVP and SIVP will also be found. Thus, Regev's reduction proof was highly significant and in 2018 he won the Godel Prize for his work [33]. Along with the proof, he proposed the classical public-key cryptosystem in which security is based on the hardness of the LWE problem and by extension the worst-case quantum hardness of GAPSVP and SIVP as discussed. He also provided a classical security proof. Security of the system is a result of the fact that the list of equations with error is computationally indistinguishable from a list of equations in which the elements are chosen uniformly [1]. While Regev's public key system does not appear on Wikipedia's list of prospects for post-quantum cryptography, perhaps due to its inefficiency, a key exchange protocol called Ring-LWE Signature appears nearby McEliece's system and has security which is based on the LWE problem specialized to polynomial rings over finite fields [27]. The RLWE problem also appeared in a 2017 algorithm called NewHope, a key-exchange protocol selected as a round-two contestant in the NIST Post-Quantum Cryptography Standardization competition, and was one of two algorithms used in the CECPQ1 experiment, an experimental post-quantum key agreement protocol developed by Google [36, 37, 38].

3.7 The Rust Programming Language: Adding Uniqueness

Rust is a systems programming language that first appeared in 2010 [53]. A systems programming language is a language that is used for programming a system such as an operating system as opposed to a user application. Rust is a multi-paradigm language, meaning that it can support procedural, functional, and object-oriented programming styles [53]. One of the most unique features of Rust is its enforcement of memory safety. Memory safety prevents attempts to access invalid memory through null or dangling pointers, which can cause program crashes or unexpected behavior. Unlike other languages, which enforce memory safety with garbage collectors or reference counting, Rust uses its compiler's borrow checker [53]. The borrow checker works by only allowing one mutable reference to a piece of data at a time, which prevents the invalid accesses mentioned above as well as concurrency bugs such as race conditions. Rust's memory safety also utilizes an ownership system in which each value belongs to one owner and the scope of a value is that of its owner; in this way, it can track object lifetime and variable scope [53]. Rust has roots in languages such as C++, OCaml, Haskell, and Erlang, and has been used by mainstream companies such as Amazon and Google as well as proposed for writing new Linux kernel code, an idea which Linus Thorvalds, the inventor of Linux, welcomes [53, 54]. Finally, according to the Stack Overflow Developer Survey in 2021, Rust is the most beloved programming language among over 80,000 developers for the sixth year in a row [54]. Rust has not been commonly used for experiments comparing pre-quantum and post-quantum cryptography, so its use will provide novelty as well as industry relevance.

4. BACKGROUND

The McEliece public key system is based off of linear codes and involves finite field arithmetic, and the Regev public key system is based off of the 'learning from parity with error' problem. In this section we give a short overview of linear codes, an introduction to finite fields, and an overview of the learning from parity with error problem.

4.1 Linear Codes

Coding theory is a branch of mathematics that was created in 1948 by Claude Shannon [39]. Coding theory is concerned with the problem of reproducing a message "at one point either exactly or approximately a message selected at another point" [39]. One of the main ideas of coding theory is a binary symmetric channel, or BSC, which is a channel that can transmit one bit at a time at a certain rate, but which is not completely reliable [39]. It is binary because it can only transmit 0 or 1 and it is symmetric because the probability of transmitting a 0 correctly is the same as the probability of transmitting a 1 correctly [40]. Also important to coding theory are the ideas of a binary symmetric source, which emits bits also at a certain rate, and a sender whose job it is to convey to the receiver through the BSC and as accurately as possible the source output [39]. The BSC rate may be faster than the source rate so that multiple bits can be transmitted per single bit emitted by the source [39]. Therefore the sender needs some strategy to encode the bits, and the receiver needs some strategy to decode the bits, such that the receiver can correct any garbled bits and retrieve the original message. This gives rise to the definition of a linear code. An (n, k) linear code is a scheme in which the source sequence is divided into blocks of k bits, and each block is encoded into a n-bit codeword which then is transmitted over the channel and then received possibly garbled [39]. The decoder then maps each n-bit possibly garbled codeword into a k-bit block which is an estimate of the original source sequence [39]. An image of this process is shown in Figure 1.

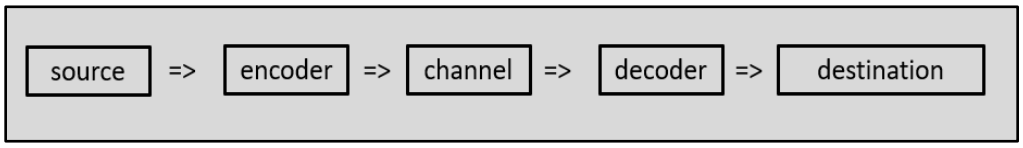


Figure 1. Process of a linear code

Within this context, the *generator matrix* is what maps k-bit blocks into n-bit codewords; the *parity check matrix* is a matrix such that if an n-bit vector is an element of the code, then the product of the parity check matrix and the vector is equal to the zero vector; and the *syndrome* is a vector that is computed by multiplying a codeword by the parity check matrix and which reveals the error vector that was introduced to the codeword by the channel [39].

4.2 Finite Fields

A finite field is a set of elements together with two binary operations such that for all elements of the set, the field axioms are obeyed [40]. If the binary operations are called addition and multiplication, the field axioms are: closure under addition, associativity of addition, additive identity, additive inverse, commutativity of addition,

closure under multiplication, associativity of multiplication, distributive laws, commutativity of multiplication, multiplicative identity, no zero divisors, and multiplicative inverse [40]. The number of elements in a finite field is always a power of a prime, and the field is called GF(p^n), where GF stands for Galois field after the mathematician who first studied finite fields [40]. A polynomial over a finite field is a polynomial with coefficients in that field.

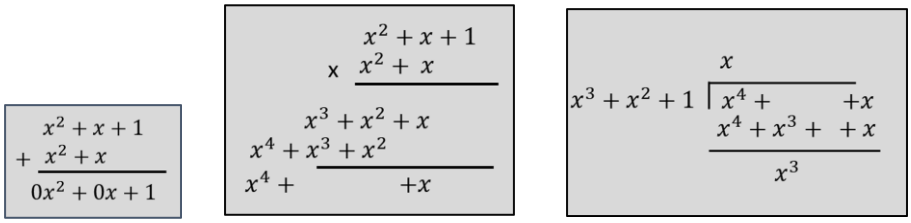
4.2.1 The Finite Field GF(2)

Finite fields of the form GF(p^n) where $n = 1$ are a special case of GF(p^n) finite fields. In these special cases, elements are taken to be the set $Z_p = \{0,1, \dots, p-1\}$ and the two field operations are addition and multiplication modulo n. The finite field GF(2) is one of these cases. It follows that the elements of GF(2) are 0 and 1, with addition taken modulo 2.

4.2.2 The Finite Field GF(2^m)

The finite field GF(2^m), $m > 1$ is created by taking the set of all polynomials over GF(2) of degree less than m and defining addition as polynomial addition and multiplication as polynomial multiplication modulo some irreducible polynomial over GF(2) of degree m called $f(x)$. Since the coefficients are in GF(2), all coefficient arithmetic is done in GF(2).

For example, to create the field GF(2^3), first choose an irreducible polynomial of degree 3, say $f(x) = x^3 + x + 1$, and list all the polynomials over GF(2) of degree less than 3. These polynomials are 0, 1, x, $x+1$, x^2, x^2+1, x^2+x, and x^2+x+1. As expected there are $2^3 = 8$ elements in the field. Examples of addition and multiplication are shown in Figure 2.



a) addition b) multiplication c) reduction

Figure 2. Examples of arithmetic in GF(2^3)

Since the coefficients of polynomials in a field GF(2^m) are 0 or 1, elements of this field can be represented as binary strings of length m. For example, the elements of GF(2^3) can be represented as 0, 1, 10, 11, 100, 101, 110, and 111.

4.2.3 The Finite Field GF(2^{mt})

The field GF(2^{mt}) is created by taking the set of all polynomials over GF(2^m) with degree less than t and defining addition as polynomial addition and multiplication as polynomial multiplication modulo some irreducible polynomial over GF(2^m) of degree t called $g(x)$. Since the coefficients are in GF(2^m), all coefficient arithmetic is done in GF(2^m). For example, to create the field GF($2^{(3)(2)}$), first choose an irreducible polynomial of degree 2, say $g(x) = x^2 + x + 1100$, and list all the polynomials over GF(2^3) of degree less than 2. These polynomials are

shown in Figure 3 with binary strings used to represent the elements of GF(2^3). Using $f(x) = x^3 + x + 1$ to create the coefficient field, examples of addition and multiplication are shown in Figure 4.

$$0x+0, 0x+1, 0x+10, 0x+11, 0x+100, 0x+101, 0x+110, 0x+111,$$

$$1x+0, 1x+1, 1x+10, 1x+11, 1x+100, 1x+101, 1x+110, 1x+111$$

$$10x+0, 10x+1, 10x+10, 10x+11, 10x+100, 10x+101, 10x+110, 10x+111,$$

$$11x+0, 11x+1, 11x+10, 11x+11, 11x+100, 11x+101, 11x+110, 11x+111,$$

$$100x+0, 100x+1, 100x+10, 100x+11, 100x+100, 100x+101, 100x+110, 100x+111,$$

$$101x+0, 101x+1, 101x+10, 101x+11, 101x+100, 101x+101, 101x+110, 101x+111,$$

$$110x+0, 110x+1, 110x+10, 110x+11, 110x+100, 110x+101, 110x+110, 110x+111,$$

$$111x+0, 111x+1, 111x+10, 111x+11, 111x+100, 111x+101, 111x+110, 111x+111,$$

Figure 3. Elements of GF($2^{(3)(2)}$)

$$\begin{array}{r} 110x+1001 \\ +\ 1111x+0001 \\ \hline 1001x+1000 \end{array}$$

a) addition

$$\begin{array}{r} 110x+1001 \\ \times\ 1111x+0001 \\ \hline 110x+1001 \\ (1111)(110)x^2+(1111)(1001)x \\ \hline 110x+1001 \\ 101x^2+0011x \\ \hline 0101x^2+0101x+1001 \end{array}$$

b) multiplication

$$\begin{array}{r} 101 \\ x^2+x+1000 \ \overline{)\ 101x^2+0101x+1001} \\ \underline{101x^2+101x+(1100)(101)} \\ 1001 \\ 110 \\ \hline 1111 \end{array}$$

c) reduction

Figure 4. Examples of arithmetic in GF($2^{(3)(2)}$)

4.3 The Learning from Parity With Errors Problem

For an integer $n \geq 1$ and a real number $\epsilon > 0$ the learning from parity with error problem is to find $\boldsymbol{s} \in Z_2^n$ given the following list of "equations with errors":

$$\langle \boldsymbol{s}, \boldsymbol{a_1} \rangle \approx_\epsilon b_1 (mod\ 2)$$
$$\langle \boldsymbol{s}, \boldsymbol{a_2} \rangle \approx_\epsilon b_2 (mod\ 2)$$
$$\ldots$$

where the a_i are chosen uniformly from Z_2^n and $\langle \boldsymbol{s}, \boldsymbol{a_i} \rangle = \sum_j s_j (a_i)_j$ is the inner product modulo 2 of $\boldsymbol{s}$ and $\boldsymbol{a_i}$, and each equation is correct with probability $1 - \epsilon$. The inputs to the problem are the pairs $(\boldsymbol{a_i}, b_i)$ where each $\boldsymbol{a_i}$ is chosen independently and uniformly from Z_2^n and where each b_i is independently chosen to be equal to $\langle \boldsymbol{s}, \boldsymbol{a_i} \rangle$ with probability $1 - \epsilon$. The goal is to find $\boldsymbol{s}$ [2].

5. CRYPTOSYSTEM STEPS

In this chapter we go over key generation steps, encryption steps, and decryption steps for the McEliece and Regev public key systems.

5.1 McEliece Key Generation

Many of the steps below are taken from [42] and [52].

1. Choose the parameters n and t with n a power of 2.

2. Set $m = \log_2(n)$. Generate an irreducible polynomial over GF(2) of degree m called $f(x)$and an irreducible polynomial over GF(2^m) of degree t called $g(x)$.

3. Find the parity check matrix for the (n, k) linear code. k is not known at this point but is not necessary to find the parity check matrix. First create the set L by listing all the elements of GF(2^m) in some arbitrary order. There will be n elements and say $L = \{\alpha_1, \alpha_2, \dots, \alpha_n\}$ for α_i in GF(2^m). The parity check matrix is $H = xyz$ where

$$x = \begin{bmatrix} g_t & 0 & 0 & \dots & 0 \\ g_{t-1} & g_t & 0 & 0 & 0 \\ \dots & \dots & \dots & 0 & 0 \\ g_1 & g_2 & g_3 & \dots & g_t \end{bmatrix}, y = \begin{bmatrix} \alpha_1^0 & \alpha_2^0 & 0 & \dots & \alpha_n^0 \\ \alpha_1^1 & \alpha_2^1 & 0 & \dots & \alpha_n^1 \\ \dots & \dots & \dots & \dots & \dots \\ \alpha_1^{t-1} & \alpha_2^{t-1} & 0 & \dots & \alpha_n^{t-1} \end{bmatrix}, \text{ and } z = \begin{bmatrix} \frac{1}{g(\alpha_1)} & 0 & 0 \\ 0 & \frac{1}{g(\alpha_2)} & 0 \\ \dots & \dots & \dots \\ 0 & \dots & \frac{1}{g(\alpha_n)} \end{bmatrix}.$$

x is t x t, y is t x n, and z is n x n, so H will be t x n.

4. Find the generator matrix for the (n, k) linear code. The generator matrix is the transpose of the nullspace of the parity check matrix: $G = nullspace(H)^T$. The dimension k is equal to the number of rows in G.

5. Generate a k x k dense nonsingular matrix S. Make sure that the inverse is equal to 1 or -1 so that the inverse will be whole numbers [43]. Also generate a n x n random permutation matrix P. Scramble the generator matrix by doing $G' = SGP$.

6. The public key is G'. The private key is S, G, P, $g(x)$, $f(x)$, and L.

5.2 McEliece Encryption

1. To encrypt a binary string m of length k first do $y = mG'$. Then y will be of length n. Add up to t errors to y by adding a length n vector e of all zeros and up to t ones. $y' = y + e$ and y' is the ciphertext for sending.

5.3 McEliece Decryption

1. If y' is the received ciphertext, first compute $y'P^{-1}$. This quantity will be equal to $(y + e)P^{-1} = yP^{-1} + eP^{-1}$.

2. Use Patterson's algorithm to find eP^{-1}.

a) If y stands for $y'P^{-1}$, compute the syndrome $s(x) = \sum_{i=1}^{n} \frac{y_i}{x - \alpha_i} mod\ g(x)$

b) Compute $v(x) \equiv \sqrt{s(x)^{-1} - x}\ mod\ g(x)$

c) Find $a(x)$ and $b(x)$ so that $a(x) \equiv b(x)v(x)\ mod g(x)$ and $\deg(a) \leq \left\lfloor \frac{t}{2} \right\rfloor$ and $\deg(b) \leq \lfloor (t-1)/2 \rfloor$.

d) Set $\sigma(x) = a(x)^2 + x \cdot b(x)^2$.

e) Plug in each element of L to $\sigma(x)$. If the result is zero and the element plugged in was α_i,

there was an error added to index i. Create eP^{-1} by creating a vector of all zeros except with ones in the indices where $\sigma(\alpha_i) = 0$.

3. Find yP^{-1} by adding eP^{-1} found in the previous step to $y'P^{-1}$. Since $yP^{-1} = mSGPP^{-1}$ this results in mSG.

4. Find mS by row reducing $[G^T|(mSG)^T]$.

5. Find m by computing mSS^{-1}.

5.4 Regev Key Generation

All arithmetic in this system is done modulo p.

1. Choose the security parameter $n \in Z$.

2. Choose two more parameters p, m such that $p \geq 2$ is a prime number in the range $[n^2, 2n^2]$ and $m = (1+\epsilon)(n+1)\log_2 p$ for some constant $\epsilon > 0$. Also create a probability distribution $\chi = \overline{\Psi}_{\alpha(n)}$ on Z_p. To do this first create a normal distribution with mean 0 and standard deviation $\frac{\beta}{2\pi}$ where β is $\alpha(n) = \frac{1}{\sqrt{n}\log^2 n}$. Then taking a sample from χ is the same as taking sample from the normal distribution, reducing the result modulo 1, multiplying by p, and rounding to the nearest integer modulo p. The graph of this distribution is shown in Figure 5 [2].

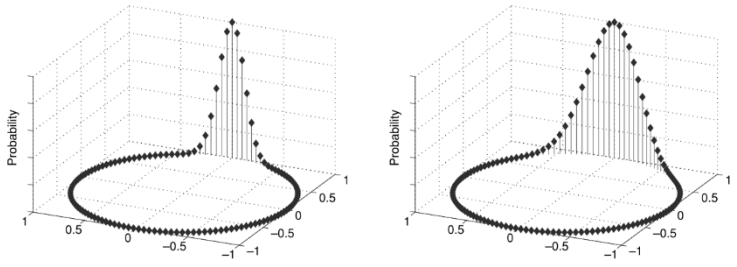


Figure 5. Graph of error distribution from [2]

4. Choose a vector in Z_p^n randomly and call it $\boldsymbol{s}$. This is the private key.

5. To create the public key, first choose m vectors in Z_p^n randomly and call them $\boldsymbol{a_1}, \ldots, \boldsymbol{a_m}$. Then choose m elements in Z_p according to χ and call them $\boldsymbol{e_1}, \ldots, \boldsymbol{e_m}$. Then compute $b_1, \ldots, b_n$ by setting $b_i = \langle \boldsymbol{a_i}, \boldsymbol{s} \rangle + e_i$ where $\langle \boldsymbol{a_i}, \boldsymbol{s} \rangle$ is the dot product of $\boldsymbol{a_i}$ and s reduced modulo p. The public key is m tuples $(\boldsymbol{a_i}, b_i)$ for $0 \leq i < m$.

5.5 Regev Encryption

1. Randomly choose a set S from all subsets of the set $\{1, \ldots, m\}$. Encrypt a 0 as $(\sum_{i \in S} \boldsymbol{a_i}, \sum_{i \in S} b_i)$ and a 1 as $(\sum_{i \in S} \boldsymbol{a_i}, \left\lfloor \frac{p}{2} \right\rfloor + \sum_{i \in S} b_i)$ where all numbers are reduced modulo p.

5.6 Regev Decryption

1. To decrypt a pair $(\boldsymbol{a}, b)$ compute $b - \langle a, b \rangle$. Check if $b - \langle a, b \rangle$ is closer to 0 than to $\left\lfloor \frac{p}{2} \right\rfloor$ modulo p. This is the same as computing the circular distance of $b - \langle a, b \rangle$ to 0 and $\left\lfloor \frac{p}{2} \right\rfloor$ and noting the smaller quantity. If it is closer to 0, the bit is 0. If it is closer to $\left\lfloor \frac{p}{2} \right\rfloor$, the bit is 1.

6. CRYPTOSYSTEM IMPLEMENTATION

In this section we will go over our implementation of the McEliece and Regev cryptosystems. First we will show the list of files in each system and give a brief description of each. Since McEliece was a large system, we divided it into five files, listed below with a brief explanation. Regev was a smaller system so we used only one file.

McEliece system files:

`GF_2m.rs`	contains finite field operations for the field GF(2^m)
`GF_2mt.rs`	contains finite field operations for the field GF(2^{mt})
`matrix.rs`	contains generator matrix method, rref, and nullspace methods. Also contains miscellaneous matrix methods.
`encrypt_decrypt.rs`	contains methods to encrypt and decrypt a binary vector, and the steps of Patterson's algorithm.
`system.rs`	contains higher level methods to create a cryptosystem and implementation of cipher block chaining

Regev system file:

`main.rs`	contains encrypting and decrypting methods, system method, and helper method

A full list of methods implemented in each file is shown in Figures 6 and 7. In sections one and two, we will go over our implementation of the finite field operations used in McEliece. Following that, we will include descriptions of the rest of the methods in the context of our implementation of key generation, encryption, and decryption.

6.1 GF(2^m) Implementation

Since elements of GF(2^m) can be represented by binary strings, they can be represented by `u64`s in Rust. A `u64` is an unsigned integer with 64 bits to represent it. The `u64` representing the element can be visualized and printed as a binary number instead of a decimal number. A printout of an element of GF(2^m) is shown in Figure 6.

```
0b0000000000000000000000000000000000000000000000000000000000000111
```

Figure 6. $x^2 + x + 1$ represented as a u64 in Rust

If the elements of GF(2^m) are represented as `u64`s in Rust, bit shifts can easily be used to implement multiplication and division of polynomials by x. For example, if $(x^2 + x) = 110$ is multiplied by x, the result will be $(x^2 + x)x = x^3 + x^2 = 1100$ and if $(x^2 + x) = 110$ is divided by x, the result will be $\left(\frac{x^2+x}{x}\right) = x + 1 = 011$ so that for any polynomial `a:u64`, `a*x = a<<1` and `a/x = a>>1`. This strategy is used in the following operations in GF(2^m). Most methods were implemented following pseudocode by Hankerson et al. in [44].

```
GF_2m.rs:
      Primary methods:
      - add()
      - multiply()
      - reduce()
      - gcd()
      - inverse()
      - pow()
      - is_irreducible()
      Helper methods:
      - vec()
      - swap()
      - bit()
      - rz()
      - deg()
      - string()
      - shift_right()
      - shift_left()
      - shift_right_i()
      - shift_left_i()
      - get_random_polynomial()
      - get_irreducible_polynomial()
      - create_L()
```

```
GF_2mt.rs:
      Primary methods:
      - add()
      - gcd()
      - inverse()
      - reduce()
      - square()
      - sqrt()
      - is_irreducible()
      Helper methods:
      - get_bx_ax()
      - multiply_by_constant()
      - is_one()
      - is_constant()
      - is_zero()
      - deg()
      - string()
      - compute()
      - shift_right()
      - shift_left()
      - shift_right_i()
      - shift_left_i()
      - get_random_polynomial()
      - get_irreducible_polynomial()
      - create_L()
```

```
encrypt_decrypt.rs:
      Primary methods:
      - encrypt()
      - decrypt()
      - error_vector()
      - sigma()
      - syndrome()
      - choose_error_vector()
```

```
matrix.rs:
      Primary methods:
      - generator()
      - dense_nonsingular_matrix()
      - permutation_matrix()
      - multiply()
      - x()
      - y()
      - z()
      - nullspace_t()
      Helper methods:
      - rref_for_decrypt()
      - rref_mod_2()
      - mod2()
      - mod2_matrix()
      - row_is_zero()
      - is_col()
      - find_col()
      - append()
      - binary_matrix()
```

```
system.rs:
      Primary methods:
      - create_system()
      - initialization_vector()
      - encrypt()
      - decrypt()
      Helper methods:
      - matrix()
      - matrix_blocks_into_vec()
      - vec_into_matrix_blocks()
      - xor_matrix()
      Structs:
      - public_key
      - private_key
```

Figure 7. List of methods in McEliece system

```
main.rs:
      Primary methods:
      - create_system()
      - draw_vector_from_Znp()
      - get_list()
      - inner_product()
      - draw_element_from_X()
      - alpha()
      - encrypt_bit()
      - choose_subset()
      - add_vector()
      - decrypt_bit()
      - circular_distance()
      Helper methods:
      - all_possible_messages()
      - vec()
      - is_equal()
      - encrypt_vector()
      - decrypt_vector()
      - encrypt()
      - decrypt()
```

Figure 8. List of methods in Regev system

6.1.1 add()

Since the elements of GF(2^m) are represented as `u64`s, addition in GF(2^m) can be implemented as a bitwise XOR: `a:u64 + b:u64 = a ^ b`.

6.1.2 multiply()

Multiplication is implemented following the right-to-left shift-and-add technique in [44]. The technique multiplies two elements of GF(2^m) together step by step while keeping the degree of the result less than m. This is useful because two elements of GF(2^m) with m up to degree 63 can be multiplied together without overflowing the `u64`. A description is below.

First, the following equivalence is established: if the irreducible polynomial used for reducing elements of the field is $f(x) = x^m + r(x)$ where $r(x)$ is some polynomial of degree $m-1$ or less, then $x^m \bmod f(x) = r(x)$. In the multiplication method, this equivalence is used to replace an instance of x^m with $r(x)$. Since $r(x)$ has degree at most $m-1$, any term of a polynomial that has a factor x^m can be reduced by at least one degree by factoring out x^m and replacing x^m with $r(x)$. Continuing with the procedure, if $a(x)$, $b(x)$ are polynomials in GF(2^m) and $a(x) = a_{m-1}x^{m-1} + \cdots + a_1x + a_0$ then $a(x)b(x)$ can be computed by distributing b(x) over a(x): $a(x)b(x) = a_{m-1}x^{m-1}b(x) + \cdots + a_1xb(x) + a_0b(x)$. Moving from right to left, and using the fact that 1 bit shift left is the same as multiplying by x, the product can be computed by first letting `result:u64` be zero, and then if the zeroth coefficient of $a(x)$ is 1, add $b(x)$ shifted left zero times to `result`, and then if the first coefficient of

$a(x)$ is 1 add $b(x)$ shifted left 1 time to `result`, and then if the third coefficient of $a(x)$ is 1 add $b(x)$ shifted left 2 times to `result`, and finally if the $m-1th$ coefficient of $a(x)$ is 1 add $b(x)$ shifted left $m-1$ times to `result`. Whenever the degree of $b(x)$ reaches m, before adding it to the result, reduce it by using the equivalence mentioned above: if $b(x) = x^m + b_{m-1}x^{m-1} + \cdots + b_0$, replace x^m with $r(x)$ by cancelling out the mth bit of $b(x)$ and adding $r(x)$ to the new $b(x)$. Any ith bit can be canceled out of a `u64` by a bitwise AND with 2^i. Since $b(x)$ is always less than degree m before adding it to the result, the result also stays less than degree m.

6.1.3 pow()

Raising an element of GF(2^m) to the power i is implemented by multiplying the polynomial by itself i times. There is no need to reduce after since our multiplication method does that.

6.1.4 reduce()

The reduce method reduces a polynomial of degree up to 63, the maximum index of the `u64`, modulo some polynomial $f(x)$ of degree m. The equivalence used in `multiply()` is also used here. First, for each term with power m or higher, take out a factor of x^m and replace it by $r(x)$. For example, if $a(x) = a_{63}x^{63} + a_{62}x^{62} + \cdots + a_m x^m + a_{m-1}x^{m-1} + \cdots + a_0$ take out x^m to form $(a_{63}x^{63-m} + a_{62}x^{62-m} + \cdots + a_m x^{m-m})x^m + a_{m-1}x^{m-1} + \cdots + a_0$ and replace x^m by $r(x)$ to get $(a_{63}x^{63-m} + a_{62}x^{62-m} + \cdots + a_m x^{m-m})r(x) + a_{m-1}x^{m-1} + \cdots + a_0$. Then work from left to right, first adding $r(x)$ times $a_{63}x^{63-m}$ to $a(x)$ and canceling out the 63rd bit, and then adding $r(x)$ times $a_{62}x^{62-m}$ to $a(x)$ and cancelling out the 62nd bit, and continue until finally adding $r(x)$ times $a_m x^{m-m}$ to $a(x)$ and canceling out the mth but. Since $r(x)$ has degree at most $m-1$, the degree of each term multiplied by $r(x)$ will be at least one degree smaller than previously. In this way the polynomial shrinks from left to right at least one bit at a time. For the implementation itself, $r(x)$ is shifted to the left by 63-m and a loop iterates from 63 down to $m-1$. For each index check if that coefficient is one. If it is, add $r(x)$ to $a(x)$ and cancel out the 1 in that index. Otherwise do nothing. Then shift $r(x)$ to the right for the next term. The resulting polynomial will have degree less than m.

6.1.5 gcd()

The greatest common divisor of two polynomials is found using the extended Euclidean algorithm. For two polynomials $a(x)$ and $b(x)$ first set $u = a,\ v = b,\ g_1 = 1,\ g_2 = 0,\ h_1 = 0, and\ h_2 = 1$. These variables are used in two invariants that hold throughout the algorithm:

$$ag_1 + bh_1 = u$$
$$ag_2 + bh_2 = v$$

At the beginning of the algorithm these will be equal to $a(1) + b(0) = u$ and $a(0) + b(1) = v$. They hold since u was set to $a(x)$ and v was set to $b(x)$. Then while u is not equal to zero, first make sure that u has the larger degree because u acts as the dividend and v as the divisor. If this is not the case, swap the values of g_1 and g_2, h_1 and h_2, and u and v so that it is the case. Then find the difference between the degrees of the leading terms of u (dividend) and v (divisor) and call it j. To cancel out the leading term of u in long division, v would be multiplied by x^j and added to u. In this case, to maintain the invariant, the entire second equation is multiplied by x^j and added to the first:

$$(ag_2 + bh_2 = v)x^j + ag_1 + bh_1 = u$$
$$=> ag_1 + ag_2x^j + bh_1 + bh_2x^j = u + vx^j$$
$$=> a(g_1 + g_2x^j) + b(h_1 + h_2x^j) = u + vx^j$$

When this is done, u shrinks by at least one degree and both invariants are maintained. Continue in this way until $u = 0$, always keeping u as the larger polynomial. When $u = 0$, v will contain the last nonzero remainder, which is the gcd, and g_2 and h_2 will be the factors.

6.1.6 inverse()

The inverse is found using the same algorithm as `gcd()`, except the loop stops when u is equal to 1. When u is equal to 1, then the first equation in the invariant will be $ag_1 + bh_1 = 1$ and so g_1 contains the inverse of a modulo b, and h_1 contains the inverse of b modulo a, because for example in the first case $ag_1 + bh_1 = 1 => ag_1 = bh_1 + 1 => ag_1 = 1 \, mod \, b$.

6.1.7 is_irreducible()

We include this method here because an appropriate irreducible polynomial is necessary for creating the field GF(2^m). To check whether a polynomial is irreducible, we use the Ben-Or irreducibility test discussed by Shuhong et al. in [46]. The pseudocode is shown below.

```
// Ben-Or irreducibility test pseudocode
for i:=1 to n/2 do
      g := gcd(f, x^(q^i) - x mod f);
      if g != 1 return false
return true
```

To find an irreducible polynomial of degree m over GF(2) we set `q` to 2 and `n` to m. As is, this method is exponential in m and requires the modular exponentiation of a polynomial with a very large exponent. To speed up this method, we used repeated squaring as suggested by Shuhong at al. in [46] and as explained in [9]. In repeated squaring, we double the exponent at each step followed by a reduction [9]. In this way we reduce intermediate results and avoid very large polynomials [9].

6.1.8 get_random_polynomial()

In this method, we supply random polynomials for `get_irreducible_polynomial()`. While trinomials and pentanomials are often desirable in the construction of finite fields due to the resulting efficiency of field operations, in our case the security of the system depends on the randomness of the polynomial [47, 1]. Choosing only trinomials and pentanomials would therefore decrease security, so we generate a random polynomial as briefly follows. First set `result = 1 as u64`. Then shift `result` to the left `m` times. This will be the polynomial x^m. Then, for indices `i= 0` through `m-1` choose the coefficient 0 or 1 randomly. If the coefficient is 1, add x^i to `result`. The resulting polynomial will be a random polynomial of degree m over GF(2).

6.1.9 get_irreducible_polynomial()

Finally we discuss how we generate irreducible polynomials. According to McEliece, about $\frac{1}{n}$ degree n polynomials are irreducible, so if we check random polynomials, we should find an irreducible one in about n tries [1]. We use this idea here: in a loop, we call `get_random_polynomial()` and check if it is irreducible with `is_irreducible()`. If it is, we return it. Otherwise, continue and try a new polynomial.

6.2 GF(2^{mt}) Implementation

Since elements of GF(2^m) are represented as `u64`s in Rust, an element of GF(2^{mt}) can be represented as a vector of `u64`s. A low order index represents a low order power and a high order index represents a high order power. However, the vectors are visualized with the high index on the left and the low index on the right in the same way the `u64`s represent the elements of GF(2^m). This is shown in Figure 9.

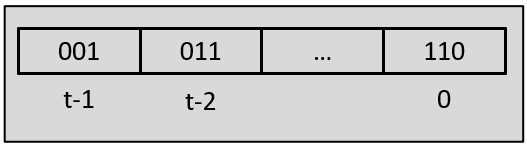


Figure 9. Element of GF(2^{mt})

Also in the same way as in GF(2^m), multiplying an element of GF(2^{mt}) is the same as a left shift where a left shift is implemented as inserting a 0 in the 0 index, and dividing an element of GF(2^{mt}) is the same as a right shift where a right shift is implemented as removing an element from the index 0. Arithmetic operations are implemented similarly to GF(2^m), except all coefficient arithmetic must be done in GF(2^m) and therefore calls on the methods implemented for that field.

6.2.1 add()

Addition is implemented in the same way as GF(2^m). Iterate the two polynomials simultaneously from low index to high index and call on `GF_2m:add()` to add their coefficients in the coefficient field. Append the result to a new vector.

6.2.2 reduce()

Reduce is implemented in the same way as in GF(2^m) except with coefficient arithmetic done in GF(2^m). If $a(x) = a_d x^d + a_{d-1}x^{d-1} + \cdots + a_t x^t + a_{t-1}x^{t-1} + \cdots + a_0$ factor out x^t from all possible terms to form $(a_d x^{d-t} + a_{d-1}x^{d-1-t} + \cdots + a_t x^{t-t})x^t + a_{t-1}x^{t-1} + \cdots + a_0$. Replace x^t by $r(x)$ to get $(a_d x^{d-t} + a_{d-1}x^{d-1-t} + \cdots + a_t x^{t-t})r(x) + a_{t-1}x^{t-1} + \cdots + a_0$. Then distribute $r(x)$ starting from the left. To accomplish this first shift $r(x)$ to the left d times and then multiply it through by a_d. Since a_d and the coefficients of $r(x)$ are in GF(2^m), call on `GF_2m::multiply()` to multiply and reduce the coefficients in the same step. As in GF(2^m), $a(x)$ will shrink at least one degree at a time. In this case as it shrinks, the topmost element can be popped off instead of canceled out.

6.2.3 gcd()

The greatest common divisor is implemented using the extended euclidean algorithm with coefficient arithmetic done in GF(2^m). The algorithm is exactly the same as in GF(2^m) except methods from that field must be called to do coefficient work. It also follows that canceling out the first term of u has some additional steps because this time the coefficients of the leading terms of u and v are elements of GF(2^m):

$$ag_1 + bh_1 = u$$
$$ag_2 + bh_2 = v$$

Where $u = u_i x^i + \cdots u_0$ and $v = v_j x^j + \cdots v_0$ for u_i, v_i in GF(2^m). To cancel out $u_i x^i$ using $v_j x^j$ first find $c = v_j^{-1} mod\ f(x)$. Then $cv_j = 1\ mod\ f(x)$. Multiply that entire equation by u_i to form $(cv_j = 1\ mod\ f(x))u_i$ and then $cv_j u_i = u_i\ mod\ f(x)$. So to cancel out $u_i x^i$, $v_j x^j$ can be multiplied by $cu_i x^{i-j}$ and added to u and the leading coefficient of u will be canceled out. However, since the invariants must be maintained, multiply the entire second equation by $cu_i x^{i-j}$ and add it to the first. Call on `GF_2m::multiply()` `GF_2m::add()` and to accomplish this. u will shrink by at least one degree each iteration, and the loop will stop when u is zero. v will contain the last nonzero remainder and g_2 and h_2 will contain the factors.

6.2.4 inverse()

Inverse is implemented again using the extended Euclidean algorithm and is the same as inverse for GF(2^m) except with arithmetic done in GF(2^m) . It also has the modification that loop stops not when $u = 1$ but when u is any constant. This works because any constant can be multiplied by its multiplicative inverse in the field GF(2^m) to get 1. Multiply the other side of the equation by the same number to get the inverse:

$$(ag_1 + bh_1 = c_1) * c_1^{-1} mod\ f(x)$$

And if $c_2 = c_1^{-1} mod\ f(x)$,

$$ag_1 c_2 + bh_1 c_2 = 1.$$

6.2.5 square()

Squaring a polynomial over a finite field is the same as squaring the coefficients in their finite field and then inserting a zero between all elements followed by a reduction [44]. An example is shown in Figure 10.

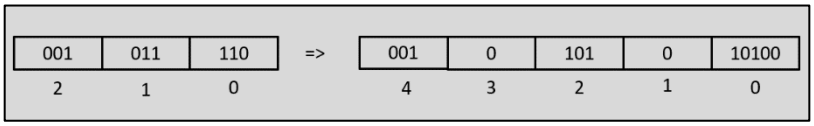


Figure 10. Squaring a polynomial over a finite field

6.2.6 sqrt()

The square root of an element x in a finite field GF(2^{mt}) is $\sqrt{x} = x^{2^{tm-1}}$ [45]. Square root is implemented by calling `square()` on the element $t \cdot m - 1$ times.

6.2.7 is_irreducible()

To create the field GF(2^{mt}) an irreducible polynomial of degree t over GF(2^m) is required. To check whether a polynomial of degree t over GF(2^m) is irreducible, we use the Ben-Or irreducibility test as in GF(2^m) [46]. The pseudocode is shown again for reference.

```
// Ben-Or irreducibility test pseudocode
for i:=1 to n/2 do
      g := gcd(f, x^(q^i) - x mod f);
      if g != 1 return false
```

```
return true
```

In this case, q is equal to 2^m and to implement repeated squaring, we square the polynomial m times per iteration instead of once as in the previous case.

6.2.8 get_random_polynomial()

To generate a random polynomial we use the same technique as in the previous section. First we set `result` to `vec![1 as u64]` and then shift it left t times to create x^t. Then for each index `i` from `0` to `t-1`, we generate a random number in the range `[0, 2^m)` as the coefficient and set `result[i]` equal to that number.

6.2.9 get_irreducible_polynomial()

This method works the same way as in the previous section. In a loop, we generate random polynomials and test each one for irreducibility until we find one.

6.3 McEliece Implementation

In our McEliece implementation the Rust library `peroxide` is used for operations with matrices. A `Matrix` in `peroxide` holds `f64`s, so elements of GF(2^m) can be converted from `u64`s to `f64`s to go in the matrix.

6.3.1 McEliece Key Generation Implementation

Key generation is accomplished by the method `system::create_system().`

1. Choosing parameters: The `system::create_system()` method takes in the system parameters `n:u64` and `t:u64` and can be called from `main()`. The method first checks that `n` is a power of `2`. If it is not, the program exits. Otherwise, it sets `m = n.log2()` and continues. Then it checks that $2^m > mt$. If this condition is not met, the dimension of the linear code will be zero. If it is not met, the program exits. Otherwise, it continues.

2. Generating irreducible polynomials: In `create_system()`, `GF_2m::get_irreducible_polnomial()` is called to create $f(x)$ and `GF_2mt::get_irreducible_polynomial()` is called to create $g(x)$. These methods were explained above.

3. Finding the parity check matrix for the (n, k) linear code: After the irreducible polynomials are found, `L` is created by calling `GF_2m::create_L()`. In `create_L()`, a loop is used to iterate the range `[0,2.pow(m))` which are the elements of GF(2^m). For each element `x`, `GF_2mt::compute()` is called to compute `g(x)`. This is done to make sure that $g(x)$ is really irreducible, because if it is irreducible, it should have no roots in GF(2^m). Finally all the elements are put in a vector in an arbitrary order and returned. Next, to create the matrix x, a `t` by `t` matrix of zeros is first created using `x=peroxide::zeros(t, t)`. Then the coefficients of $g(x)$ except for g_0 are copied to the last row of x. g is shifted one to the left then the coefficients are copied to the second to last row. This process continues until the matrix is filled. To create y, a `t` by `n` matrix of zeros is created and the first row is set to all ones. Then for `r` in `1..t` and `c` in `0..n` each element is set to `GF_2m::pow(L[c],r,f)`. To create z, an `n` by `n` matrix of zeros is created. Then a loop iterates over the rows and columns and and when `r==c`, the element at `z[(r,c)]` is set to `f1::inverse(f2::compute(&g, f, L[r]), f)`. Finally, xyz must be computed. However, since the elements of all three matrices are in GF(2^m), and decimal multiplication and addition is not the

same as GF(2^m) multiplication and addition, the `peroxide` built-in matrix multiplication method cannot be used. Instead, a special method is used which does the arithmetic in the proper field. An image is shown in Figure 11.

```
pub fn multiply(x:&Matrix, y:&Matrix, xrows:usize, xcols:usize, yrows:usize, ycols:usize, f:u64) -> Matrix {
    if xcols != yrows {
        println!("matrix multiplication error");
        process::exit(1);
    }
    let mut z = zeros(xrows, ycols);
    for r in 0..xrows {
        for c in 0..ycols {
            for t in 0..xcols {
                z[(r,c)] = f1::add(z[(r,c)] as u64, f1::multiply(x[(r,t)] as u64, y[(t,c)] as u64, f)) as f64;
            }
        }
    }
    z
}
```

Figure 11. Matrix multiplication using field multiplication and addition where GF_2m is used as f1

After finding H, the elements of H will be `f64`s, each representing an element of GF(2^m). To continue with the rest of the steps this matrix must be converted to a binary matrix, so each element is converted into a `u64` and then to a binary string including leading zeros using `GF_2m::vec()`. `GF_2m::vec()` creates a new `Vec` of length `deg(f)` and loops over the indices of the `Vec` from high order to low order. If `u % 2` is zero, set the `vec[index]` to zero. Otherwise set `vec[index]` to one. Then shift the `u64` to the right. Continue until the vector is filled. This results in a vector where the low order bits are in the high order index. This is so that the vector can be read from left to right and still represents the `u64` in binary. In the matrix, each vector is then written downwards from high order to low order so that the dimensions of H become `tm` by `n`.

4. Finding the generator matrix for the (n, k) linear code $G = nullspace(H)^T$: The `peroxide` matrix library did not have a method to find the nullspace of a matrix so a method was implemented by hand based off of Kahn Academy [48]. To find the nullspace of a matrix, the matrix is first put into reduced row echelon form. The `peroxide` matrix library also does not have a method to row reduce a matrix so a method was also implemented by modifying pseudocode from Wikipedia [49] to work modulo 2. To put a matrix into reduced row echelon form, `lead`, representing the lead column, is set to zero and a loop goes over the rows of the matrix from top to bottom with variable `r`. For each row `r`, a variable `i` is initialized to zero. `i` is sent down the matrix until it finds a row where the element at row `i`, column `lead` is one. The row found is put into row `r` by swapping. Then the whole column `lead` is cleared except for the `1` at row `r`. Since arithmetic is modulo 2, this is achieved by adding row `r` to which ever row does not have a zero at row `r`, column `lead`. Then the `lead` column is incremented and the process repeats until `lead` goes past the end of the matrix. At the end of the method, the matrix is in reduced row echelon form. However, reduced row echelon form is not necessarily an identity matrix. To finish creating an identity matrix, the code walks through the matrix from left to right to find columns that do not satisfy the identity matrix. It then calls the method `find_col()` to find and swap an appropriate column. The pairs of columns swapped are recorded in a vector `swap` to be swapped back later. A this point the matrix has some kind of identity matrix on the left. If the identity matrix is `mt` by `mt`, the number of columns which is the dimension of the code will be `n-mt`. It the identity matrix is not `mt` by `mt`, there must be a zero row at the bottom of the matrix. For every zero row at the bottom of the matrix there will be one additional column in the nullspace and thus the dimension of the code is

increased by one for each zero row. At this point a new matrix `nullspace` is created to hold the nullspace. The number of rows in `nullspace` must be equal to the number of columns of the original matrix since there must be a row for every variable in the system of equations being solved. The number of columns is equal to however many columns are not a part of the leftward identity matrix which is some number greater than or equal to `n-mt`. So a matrix of zeros of that size is created. Then, the columns to the right of the identity matrix are copied into `nullspace`. Finally, if there is a block of zero rows at the bottom of `nullspace`, fill it with an identity matrix. As the last step, go through the vector `swap` in reverse order and swap the rows back (not the columns as previously done). This matrix represents the columns of the nullspace of the original matrix. The matrix is transposed and returned and the generator matrix is this matrix with dimension `k>=n-mt` and length `n`.

5. Generating a k x k dense nonsingular matrix S: Since the Rust library often crashes when finding the determinant, we use the facts that the identity matrix has a determinant of one and that adding a row to another row does not change the determinant to create our own nonsingular matrix [49]. However, since the matrix elements are in GF(2) we cannot add a row to another row if it would cause any of the elements to reach 2. So to create the matrix we start by creating an identity matrix. Then we loop through each row and add it to whichever row will remain in GF(2) upon addition. We keep track of the matrix density and after each addition increase it accordingly. We continue the loop until density stops increasing.

Generating a n x n random permutation matrix P: First an `n` by `n` matrix is created. A vector called `ones` is generated containing values in the range `[0,n)` and then shuffled. A loop iterates through the range `[0,n)` and sets `P[(i,ones[i])] = 1`.

6. The public key is G': A `struct` called `public_key` is used to hold G', length, dimension, and t. The public key needs to contain `t` so that the encrypter knows how many errors they can include.

The private key is S, G, P, $g(x)$, $f(x)$, and L: A `struct` called `private_key` is used to hold S, G, P, L, g, f, length, and dimension. An image is shown in Figure 12. A tuple `(public_key, private_key)` is returned from `create_system()`.

```
pub struct public_key {
    generator: Matrix,
    length: u64,
    dimension: u64,
    t: u64
}
```

```
pub struct private_key {
    S: Matrix,
    G: Matrix,
    P: Matrix,
    L: Vec<u64>,
    g: Vec<u64>,
    f: u64,
    length: u64,
    dimension: u64
}
```

Figure 12. Structs used to hold the public and private keys in McEliece

6.3.2 McEliece Encryption Implementation

1. Encrypting a binary string `m`: In `encrypt()`, the `peroxide` matrix library is be used to compute `mG`. A random error vector of weight `t` is chosen from among all possible error vectors, which is $C(length, t)$ combinations. We

do this by first creating a vector with `length` zeros. Then we use a loop and choose a random index at each iteration. We keep track of the indices chosen and make sure that we do not choose the same one twice. For each index we set that element equal to one. The error vector is added to `mG`.

6.3.3 McEliece Decryption Implementation

1. Computing $y'P^{-1}$: The `peroxide` matrix libary is used to find `P.inv()` and compute `yP_inv = y*P.inv()` where the parameter of y' is called `y`.
2. Using Patterson's algorithm to find eP^{-1}:
    a) The syndrome $s(x) = \sum_{i=1}^{n} \frac{y_i}{x-\alpha_i} mod\ g(x)$ is computed by calling `encrypt_decrypt::syndrome()`. In `syndrome()`, the denominators are computed by calling `GF_tm::add()` because in this case x is a member of GF(2^{mt}) and α_i is a constant in GF(2^{mt}) so addition is computed in GF(2^{mt}). Then, since y_i is either one or zero, for every y_i that is a one, the fractional quantity becomes the inverse of an element in GF(2^{mt}), modulo $g(x)$. To accomplish this, `GF(2_mt)::inverse()` is called. The inverses, which are also elements of GF(2^{mt}), are summed by using `GF_2mt:add()`. The sum is also an element of GF(2^{mt}).
    b) Computing $v(x) \equiv \sqrt{s(x)^{-1} - x}\ mod\ g(x)$: The syndrome is an element of GF(2^{mt}) so its inverse is computed by calling `GF_2mt::inverse()`. x in GF(2^{mt}) is added by calling `GF_2mt::add()`. The square root is computed by calling `GF_2mt::sqrt()`.
    c) Finding $a(x)$ and $b(x)$ so that $a(x) \equiv b(x)v(x)\ mod g(x)$ and $\deg(a) \leq \left\lfloor \frac{t}{2} \right\rfloor$ and $\deg(b) \leq \lfloor (t-1)/2 \rfloor$: These polynomials are found using a special method `GF_2mt::get_bx_ax()` which is an extended Euclidean algorithm that stops early, at the conditions required.
    d) Setting $\sigma(x) = a(x)^2 + x \cdot b(x)^2$: The polynomial σ is computed by `encrypt_decrypt::sigma()` which calls methods from `GF_2mt` even though the polynomials are not elements of GF(2^{mt}). `GF_2mt::square()` is called on `ax` and `bx` (no reduction), `bx2` is shifted left by one index, and `GF_2mt::add()` is called to add `ax2` and `b2x` and get `sigma`.
    e) Plugging in each element of L to $\sigma(x)$: `GF_2mt::compute()` is called to plug in the element. `compute()` just calls the `GF_2m` arithmetic methods `pow()`, `multiply()`, and `add()` to compute $\sigma(x)$ and return the value which is in GF(2^m). The error vector is created by `error_vector()` which takes in `sigma` and `L`.
3. Finding $yP^{-1} = mSG$: This quantity is found by adding `error_vector` to `y` modulo 2.
4. Finding mS by row reducing $[G^T|(mSG)^T]$: Since the `peroxide` library does not have an append method, `matrix::append()` was implemented. Then `G.transpose()`, with the transpose found using the library method, is appended to `mSG.transpose()` and a method `rref_for_decrypt()` is run which puts the matrix into reduced

row echelon form without clearing the rightmost column. Then `k` elements are taken from the top of the rightmost column.

5. Finding m by computing mSS^{-1}: `S.inv()` is found using the library and the product is computed the library: `m = mS*S.inv()`.

6.4 Regev Implementation

Random numbers are generated using the Rust library `std::rand::Rng`. Vectors of integers are represented as `Vec`s of `u64`s.

6.4.1 Regev Key Generation Implementation

Key generation is accomplished by the method `create_system()`.

1. Choosing parameters: the `create_system()` method takes in the system parameters `n` and `E` and can be called from `main()`. First the range of `p` is set by letting `lower=n*n` and `upper=2*lower`. Then `p` is generated by `p = rand::thread_rng().gen_range(lower..upper)`. `m` is set by doing `m = ((1.0 + E) * (n as f64 + 1.0) * (p as f64).log2())` and then `round()` is used to round it since `m` is supposed to be an integer. It is not specified in the instructions how to make it an integer. `B` is set to `alpha(n)` which implements α. A private key is created by calling the method `private_key()`. The method `private_key()` calls `draw_vector_from_Znp()`. The method `draw_vector_from_Znp()` returns a vector of length `n` filled with random elements from Z_p to use as the private key. `create_system()` continues by calling `public_key()`. The method `public_key()` loops from `0` to `m`. For each loop a vector is drawn from Z_p^n using `draw_vector_from_Znp()`, and then `b` is computed by `calling b = inner_product(&a, &s, p) + draw_element_from_X(p, B)`. In `inner_product()` the lengths of both vectors are compared to see if they are equal. If they are not, the program exits. Otherwise it continues, and `sum` is set to `zero` and for the range `0..a.len()` the product of `a[i]` and `b[i]` is added to `sum`. The sum which is a `u64` is reduced by `p` and then returned. `draw_element_from_X()` is the method that draws the element from the distribution χ. It takes `p` and `B`. First `mean` is set to 0 and `std_dev` to `B/(2*PI).sqrt()`. Then the Rust library `rand_distr::{Normal, Distribution}` is used to create a normal distribution with mean `mean` and standard deviation `std_dev`. That normal distribution is sampled from which results in an `f64`, which is reduced modulo `1.0` with `rem_euclid(1.0)`, multiplied by `p`, rounded it to the nearest integer using `round()` and then reduced modulo `p`. The resulting number which is a `u64` is returned. Returning to `public_key()`, `a` is a random vector from Z_p^n and `b` is an element computed by `b = inner_product(&a, &s, p) + draw_element_from_X(p, B)` and is a `u64` reduced modulo `p`. Then a `pair` instance is created where a `pair` is a `struct` used to hold together `a:Vec<u64>` and `b:u64` which are the structures used in the public key and the ciphertext. A picture of the `struct` is shown in Figure 14.

```
pub struct pair {
    a: Vec<u64>,
    b: u64
}
```

Figure 14. Struct used to hold elements of the public key and the ciphertext in Regev

```
pub struct private_key {
    s: Vec<u64>,
    p: u64
}

pub struct public_key {
    m: u64,
    n: u64,
    p: u64,
    list: Vec<pair>
}
```

Figure 13. Structs used to hold private and public key in Regev

For `i` in `0..m` a `pair` is created and pushed to a vector of `pair`s. Finally in `create_system()` an instance of the `public_key struct` is created and an instance of the `private_key struct` is created. These `struct`s are shown in Figure 13. A tuple `(public_key, private_key)` is returned.

6.4.2 Regev Encryption Implementation

The method `encrypt_vector()` is called to encrypt a message which is a binary vector. A new vector called `encrypted_bits` is created and a loop goes over the bits in the message. Each bit is encrypted one by one by calling `encrypt_bit()`. `encrypt_bit()` first calls `choose_subset(m)` which returns a random subset out of all subsets of $\{1..m\}$. `choose_subset()` works by generating a random number in the range `0..2.pow(m)`. Since the parameters of the system are large enough that this upper bound does not fit in a `u64`, this is done by creating a vector of length `m` and setting each element randomly to zero or one. Then for each element `i` in `1..m`, if the `i`th bit of the random number is one, `i` is pushed to the set. The resulting set is returned which is a vector of `u64`s. After choosing the subset in `encrypt_bit(),` the two sums $\sum_{i \in S} \boldsymbol{a_i}$ and $\sum_{i \in S} b_i$ are created by looping over the elements of the subset and for each `element` adding `public_key[element].a` to the first sum and `public_key[element].b` to the second sum. The vectors in the first sum are added using `add_vector()` which adds a vector modulo `p` and `u64` addition modulo `p` is used for the second sum. Finally if the bit is 0 the pair is returned as is, otherwise the extra value `y += (p/2).floor()` is added to the second sum and then the pair is returned. Each encrypted bit is appended to the vector `encrypted_bits()`. The vector is returned and is a vector of `pair`s.

6.4.3 Regev Decryption Implementation

In `decrypt_vector()`, a vector of `pair`s is taken and decrypted one by one by calling `decrypt_bit()`. In `decrypt_bit()`, `inner_product()` is first called to compute the inner product of `pair.a` and `s` and `point` is set to `pair.b-inner_product`. Then `q` is set to `p/2.floor()` and `distance_to_0` is computed as the circular

distance from `point` to `0` using our method `circular_distance()`. The method is shown in Figure 15 and is taken from Stack Overflow [50].

```
fn circular_distance(point:u64, target:u64, modulus:u64) -> u64 {
    let diff = (point as i64 - target as i64).abs() as u64;
    return std::cmp::min(diff, modulus-diff);
}
```

Figure 15. Circular distance function used in Regev decryption

Then `distance_to_q` is computed as `the` circular distance from point to q using `circular_distance()`. If `distance_to_0` is less than `distance_to_q`, `0` is returned, otherwise `1` is returned. In `decrypt_vector()`, the decrypted bits are appended to a vector one by one and returned.

7. EXAMPLES

In this section we show examples of key generation, encryption, and decryption for the McEliece and Regev systems.

7.1 McEliece Key Generation Example

1. Choose n and t with a n a power of 2 and $2^m > mt$:

$(n, t) = (8,2)$

2. Generate an irreducible polynomial of degree m over GF(2), call it $f(x)$. Also generate an irreducible polynomial of degree t over GF($2^m = n$) and call it $g(x)$:

$f(x) = x^3 + x + 1,\ g(x) = x^2 + x + 111$

3. Find the parity check matrix for the linear code corresponding to $g(x)$.

Choose L: $L = \{\alpha_1, \dots, \alpha_n\} = \{011,100,110,001,000,101,010,111\}$

Then use the following formulas to create x, y, and z:

$$x = \begin{bmatrix} g_t & 0 & 0 & \dots & 0 \\ g_{t-1} & g_t & 0 & 0 & 0 \\ \dots & \dots & \dots & 0 & 0 \\ g_1 & g_2 & g_3 & \dots & g_t \end{bmatrix}, y = \begin{bmatrix} \alpha_1^0 & \alpha_2^0 & 0 & \dots & \alpha_n^0 \\ \alpha_1^1 & \alpha_2^1 & 0 & \dots & \alpha_n^1 \\ \dots & \dots & \dots & \dots & \dots \\ \alpha_1^{t-1} & \alpha_2^{t-1} & 0 & \dots & \alpha_n^{t-1} \end{bmatrix}, \text{ and } z = \begin{bmatrix} \frac{1}{g(\alpha_1)} & 0 & 0 \\ 0 & \frac{1}{g(\alpha_2)} & 0 \\ \dots & \dots & \dots \\ 0 & \dots & \frac{1}{g(\alpha_n)} \end{bmatrix}.$$

So,

$$x = \begin{bmatrix} 001 & 000 \\ 001 & 001 \end{bmatrix}, y = \begin{bmatrix} 001 & 001 & 001 & 001 & 001 & 001 & 001 & 001 \\ 011 & 100 & 110 & 001 & 000 & 101 & 010 & 111 \end{bmatrix}, \text{ and}$$

$$z = \begin{bmatrix} 001 & 000 & 000 & 000 & 000 & 000 & 000 & 000 \\ 000 & 010 & 000 & 000 & 000 & 000 & 000 & 000 \\ 000 & 000 & 110 & 000 & 000 & 000 & 000 & 000 \\ 000 & 000 & 000 & 100 & 000 & 000 & 000 & 000 \\ 000 & 000 & 000 & 000 & 100 & 000 & 000 & 000 \\ 000 & 000 & 000 & 000 & 000 & 010 & 000 & 000 \\ 000 & 000 & 000 & 000 & 000 & 000 & 001 & 000 \\ 000 & 000 & 000 & 000 & 000 & 000 & 000 & 110 \end{bmatrix}$$

where the inverses are listed here:

$1^{-1}\ mod\ x^3 + x + 1 = 001$
$x^{-1} mod\ x^3 + x + 1 = x^2 + 1 = 101$
$(x + 1)^{-1} mod\ x^3 + x + 1 = \mathrm{x}^2 + \mathrm{x} = 110$
$(x^2)^{-1} mod\ x^3 + x + 1 = x^2 + x + 1 = 111$
$(x^2 + 1)^{-1} mod\ x^3 + x + 1 = x = 010$
$(x^2 + x)^{-1} mod\ x^3 + x + 1 = x + 1 = 011$
$(x^2 + x + 1)^{-1} mod\ x^3 + x + 1 = x^2 = 100$

Then compute $H = xyz$ and then convert H to binary:

$$H = xyz = \begin{bmatrix} 001 & 010 & 110 & 100 & 100 & 010 & 001 & 110 \\ 010 & 001 & 100 & 000 & 100 & 011 & 011 & 010 \end{bmatrix}, H = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

4. Find the generator matrix for the (n, k) linear code.

First find the nullspace of H and then transpose it to get the generator matrix:

$nullspace(H) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 1 \\ 0 & 1 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $G = nullspace^T = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{bmatrix}$ and $(n, k) = (8,2)$.

5. Generate the k x k dense nonsingular matrix S and the n by n permutation matrix P:

$S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

And finally scramble the generator matrix:

$G = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{bmatrix}$ and $G' = SGP = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$

The public key is G' and the private key is S, G, P, $g(x)$, $f(x)$ and L.

7.2 McEliece Encryption Example

1. Choose a message m and compute mG':

$m = [0 \quad 1], G' = SGP = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$, and $mG' = [1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1]$

Choose an error vector and compute $y' = mG' + e$:

$e = [1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0]$ and $mG' = [1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1]$

$y' = mG' + e = [0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1]$

7.3 McEliece Decryption Example

1. Compute $y'P^{-1} = mSGP^{-1} + eP^{-1}$:

$y' = [0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1]$ and $P^{-1} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$

so $y'P^{-1} = [0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1]$

2. Use Patterson's algorithm to find eP^{-1}:

a) Compute the syndrome $s(x)$:

$y'P^{-1} = [0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1]$

$L = \{011,100,110,001,000,101,010,111\}$

$s(x) = \sum_{i=1}^{n} \frac{y_i}{x-\alpha_i} mod\ g(x) = \left(\frac{1}{x+110}\right) modg(x) + \left(\frac{1}{x}\right) modg(x) + \left(\frac{1}{x+101}\right)\ modg(x) +$

$\left(\frac{1}{010x+011}\right) modg(x) =$

$= (x+110)^{-1} modg(x) + (x)^{-1} modg(x) + (x+101)^{-1} modg(x) +$

$(010x+011)^{-1} modg(x)$

$= 110x + 1$

b) Compute $v(x) \equiv \sqrt{s(x)^{-1} - x}\ mod\ g(x)$: $110x + 111$

c)) Find $a(x)$ and $b(x)$ so that $a(x) \equiv b(x)v(x)\ modg(x)$ and $\deg(a) \leq \left\lfloor \frac{t}{2} \right\rfloor$ and $\deg(b) \leq \left\lfloor \frac{t-1}{2} \right\rfloor$: $a(x) = 110x + 111,\ b(x) = 1$

d) Set $\sigma(x) = a(x)^2 + x \cdot b(x)^2$: $10x^2 + x + 11$

e) Locate the errors by plugging in the values of L: they are at indices 1 and 3

$\sigma(011) = 001$

$\sigma(100) = 000$

$\sigma(110) = 001$

$\sigma(001) = 000$

$\sigma(000) = 011$

$\sigma(101) = 011$

$\sigma(010) = 010$

$\sigma(111) = 010$

So $eP^{-1} = [0\quad 1\quad 0\quad 1\quad 0\quad 0\quad 0\quad 0]$.

Finally Add eP^{-1} to $y'P^{-1}$ to get $mSGPP^{-1} = mSG$:

$y'P^{-1} = [0\quad 0\quad 1\quad 0\quad 1\quad 1\quad 0\quad 1],\ eP^{-1} = [0\quad 1\quad 0\quad 1\quad 0\quad 0\quad 0\quad 0]$

$mSG = [0\quad 1\quad 1\quad 1\quad 1\quad 1\quad 0\quad 1]$

3. Find mS by row reducing $[G^T|(mSG)^T]$:

$$[G^T|(mSG)^T] = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \text{ and } [G^T|(mSG)^T]\ reduced = \begin{bmatrix} 1 & 0 & \mathbf{0} \\ 0 & 1 & \mathbf{1} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

And so $mS = [0\ 1]$ as in the top right hand corner of the matrix.

4. Finally, find m by computing mSS^{-1}:

$mS = [0\quad 1]$ and $S^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, so $m = mSS^{-1} = [0\quad 1]$ as desired.

7.4 Regev Key Generation Example

1. Choose the security parameter $n \in Z$:

$n = 3$

2. Choose two more parameters p, m such that $p \geq 2$ is a prime number in the range $[n^2, 2n^2]$ and $m = (1 + \epsilon)(n + 1)\log_2 p$ for some constant $\epsilon > 0$:

$p = 10, m = 20$

4. Choose a vector in Z_p^n randomly and call it $\boldsymbol{s}$. This is the private key:

$s = [2,7,3]$

5. To create the private key, first choose m vectors in Z_p^n randomly and call them $\boldsymbol{a_1}, \ldots, \boldsymbol{a_m}$. Then choose m elements in Z_p according to χ and call them $\boldsymbol{e_1}, \ldots, \boldsymbol{e_m}$:

$a_1 = [1,0,2]$
$a_2 = [8,2,3]$
$a_3 = [1,6,3]$
$a_4 = [5,5,7]$ and
$a_5 = [2,0,4]$
$a_6 = [2,3,7]$
$a_7 = [8,5,2]$
...

$e_1 = 9$
$e_2 = 1$
$e_3 = 0$
$e_4 = 0$
$e_5 = 2$
$e_6 = 0$
$e_7 = 0$
...

Then compute $b_1, \ldots, b_n$ by setting $b_i = \langle \boldsymbol{a_i}, \boldsymbol{s} \rangle + e_i$ where $\langle \boldsymbol{a_i}, \boldsymbol{s} \rangle$ is the dot product of $\boldsymbol{a_i}$ and s reduced modulo p:

$b_1 = (1)(2) + (0)(7) + (2)(3)$ + 9 = 2+6+9=17 mod 10 = 7

...

$b_5 = (2)(2) + (0)(7) + (4)(3)$ + 2 = 4+12+2 = 18 mod 10 = 8

...

$b_7 = (8)(2) + (5)(7) + (2)(3) + 0 =$ 16+35+6 = 57 mod 10 = 7

The public key is m tuples $(\boldsymbol{a_i}, b_i)$ for $0 \leq i < m$:

$([1,0,2], 7)$

...

$([2,0,4], 8)$

...

$([8,5,2], 7)$

...

7.5 Regev Encryption Example

1. Randomly choose a set S from all subsets of the set $\{1, \ldots, m\}$:

$([1,0,2], 7)$

$([2,0,4], 8)$

$([8,5,2],\ 7)$

Then encrypt a 0 as $(\sum_{i\in S} \boldsymbol{a_i}, \sum_{i\in S} b_i)$ and a 1 as $(\sum_{i\in S} \boldsymbol{a_i}, \left\lfloor \frac{p}{2} \right\rfloor + \sum_{i\in S} b_i)$ where all numbers are reduced modulo p:

Encrypt a 0 as $(\sum_{i\in S} \boldsymbol{a_i}, \sum_{i\in S} b_i) = ([1,5,8], 2)$

Encrypt a 1 as $(\sum_{i\in S} \boldsymbol{a_i}, \left\lfloor \frac{p}{2} \right\rfloor + \sum_{i\in S} b_i) = ([1,5,8], 2+5) = ([1,5,8], 7)$

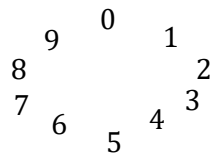
7.6 Regev Decryption Example

1. To decrypt a pair $(\boldsymbol{a}, b)$ compute $b - \langle a, b\rangle$:

If the ciphertext is $([1,5,8], 2)$, $\vec{b} - \langle \vec{a}, \vec{s}\rangle = 2 - (1)(2) + (5)(3) + (8)(7) = 2 - 61 = 2 - 1 = 1$

If the ciphertext is $([1,5,8], 7)$, $\vec{b} - \langle \vec{a}, \vec{s}\rangle = 7 - (1)(2) + (5)(3) + (8)(7) = 7 - 61 = 7 - 1 = 6$

Check if $b - \langle a, b\rangle$ is closer to 0 than to $\left\lfloor \frac{p}{2} \right\rfloor$ modulo p. This is the same as computing the circular distance of $b - \langle a, b\rangle$ to 0 and $\left\lfloor \frac{p}{2} \right\rfloor$, and noting which is the smaller quantity.



Since 1 is closer to 0 on the circle, the first bit is decrypted as a 0 as desired.

Since 6 is closer to 5 on the circle, the second bit is decrypted as a 1 as desired.

8. TESTS

It is important to test each system for correctness. Our goal is to have every decrypted message be the same as the message we encrypted. In this chapter we go over how we verified this.

8.1 McEliece: Finding Parameter Limits

Since we wanted to test the system with as many parameters as possible, we first wanted to determine how high we can turn the parameters of the system. Ideally, we would want to meet McEliece's original suggestion of a system with $m = 10$ and $t = 50$. Since the first step in creating a McEliece system is the generation of two polynomials, it makes sense to test the limits of those methods first. We previously mentioned that the generation of an irreducible polynomial of degree n should take n tries on average [1]. We first tested this hypothesis for polynomials over GF(2) by generating 100 irreducible polynomials each for degrees one through ten and averaging the number of tries it took to find each one. The results are shown in Figure 16.

```
average tries for polynomial of degree 1: 0
average tries for polynomial of degree 2: 3
average tries for polynomial of degree 3: 3
average tries for polynomial of degree 4: 6
average tries for polynomial of degree 5: 4
average tries for polynomial of degree 6: 7
average tries for polynomial of degree 7: 7
average tries for polynomial of degree 8: 8
average tries for polynomial of degree 9: 9
average tries for polynomial of degree 10: 8
$
```

Figure 16. Number of tries to find an irreducible polynomial over GF(2)

As the image shows, McEleice's claim pans out almost exactly for polynomials over GF(2). We continued the test for polynomials over GF(2) of up to degree 50 and find that it holds. Next we chose the coefficient field GF(2^3) and generated 100 polynomials over GF(2^3) each for degrees two through five and averaged the number of tries it took to find each one. The results were less impressive, with averages of 10, 20, 32, and 216 tries respectively. Since the generation of the polynomial $g(x)$ thus seems to be a bottleneck, we wanted to find out the out the largest we can generate over GF(2^m) in a reasonable amount of time. To accomplish this, we first held m constant at values of 1,2,3,4,5, and 6 and increased t starting from 1 until the generation of a polynomial took more than thirty seconds. Then we held t constant at values of 2, 3, and 4 and increased m starting from 1 until the generation of a polynomial took more than thirty seconds. The results are shown in Table 1, where the dashes represent where more than thirty seconds went by without finding a polynomial.

Table 1. Maximum parameters for irreducible polynomials over GF(2^m)

m	Maximum t	t	Maximum m
1	2	2	14
2	3	3	14
3	9	4	--

4	9		
5	7		
6	--		

Finally, as a standalone experiment, we set $t = 5$ and $m = 10$ and checked 500,000 polynomials but never found an irreducible one. We reason that a cause could be that our random polynomial generator is producing some or many repeated polynomials, but unfortunately we are unable to get to the parameters that McEliece originally suggested of $m = 10$ and $t = 50$.

8.2 McEliece: Unit Testing Parameter Limits

Nevertheless, calling on our results from above we create a table of as many sets of parameters as our irreducible polynomial generator will allow. Then, we shade the cells in grey which do not contain a runnable setting due to the parameter constraints of $2^m > mt$, $t > 1$, and $m > 1$. The results are in Table 2.

Table 2. All runnable parameter settings for McEliece

set #	params	set #	params	set #	params	set #	params	set #	params	set #	params
1	(2,1)	11	(8,6)	21	(16,7)	31	(2,2)	41	(2048,2)	51	(128,3)
2	(2,2)	12	(8,7)	22	(16,8)	32	(4,2)	42	(4096,2)	52	(256,3)
3	(4,1)	13	(8,8)	23	(16,9)	33	(8,2)	43	(8192,2)	53	(512,3)
4	(4,2)	14	(8,9)	24	(32,1)	34	(16,2)	44	(16384,2)	54	(1024,3)
5	(4,3)	15	(16,1)	25	(32,2)	35	(32,2)	45	(2,3)	55	(2048,3)
6	(8,1)	16	(16,2)	26	(32,3)	36	(64,2)	46	(4,3)	56	(4096,3)
7	(8,2)	17	(16,3)	27	(32,4)	37	(128,2)	47	(8,3)	57	(8192,3)
8	(8,3)	18	(16,4)	28	(32,5)	38	(256,2)	48	(16,3)	58	(16384,3)
9	(8,4)	19	(16,5)	29	(32,6)	39	(512,2)	49	(32,3)	59	
10	(8,5)	20	(16,6)	30	(32,7)	40	(1024,2)	50	(64,3)	60	

Continuing on, we modified the table to show only the runnable cases and added columns for the number of messages tried per system, the number of error vectors tried per message, and the total number of messages decrypted correctly. Recall that for each message to be encrypted, an error vector is added. For each message we tested multiple error vectors, and the number is listed in the # errs column. For smaller parameter values we tested 100 random messages with 10 error vectors each to create 1000 messages tested, for medium parameter values we tested 10 messages each with 1 error vector, and for large parameter values we tested 1 message with 1 error vector. For extremely large parameters, we did not wait. But in total we tested 15, 024 messages across 22 different

parameter sets. From the Table 3 it is clear that all 15, 024 messages were decrypted correctly. The number of messages decrypted correctly is highlighted in yellow.

Table 3. Number of messages decrypted correctly for all parameter settings

set #	(n,t)	# msgs	# errs	total correct	set #	(n,t)	# msgs	# errs	total correct	set #	(n,t)	# msgs	# errs	total correct
7	(8,2)	100	10	1000	35	(32,2)	100	10	1000	48	(16,3)	100	10	1000
16	(16,2)	100	10	1000	36	(64,2)	100	10	1000	49	(32,3)	100	10	1000
17	(16,3)	100	10	1000	37	(128,2)	10	1	10	50	(64,3)	100	10	1000
25	(32,2)	100	10	1000	38	(256,2)	10	1	10	51	(128,3)	1	1	1
26	(32,3)	100	10	1000	39	(512,2)	1	1	1	52	(256,3)	1	1	1
27	(32,4)	100	10	1000	40	(1024,2)	1	1	1	53	(512,3)	1	1	1
28	(32,5)	100	10	1000	41	(2048,2)	dnw			54	(1024,3)	dnw		
29	(32,6)	100	10	1000	42	(4096,2)	dnw			55	(2048,3)	dnw		
33	(8,2)	100	10	1000	43	(8192,2)	dnw			56	(4096,3)	dnw		
34	(16,2)	100	10	1000	44	(16384,2)	dnw			57	(8192,3)	dnw		

We show some example output of the tests in Figure 17. The top picture is for n = 16, t = 3. The bottom picture is for n = 32, t = 3. It shows how many tries it took to find the irreducible polynomials, it shows the polynomials it found, the length and dimension of the code, and the number of messages decrypted correctly and incorrectly. In both images, at bottom it shows that all 1000 messages were decrypted correctly.

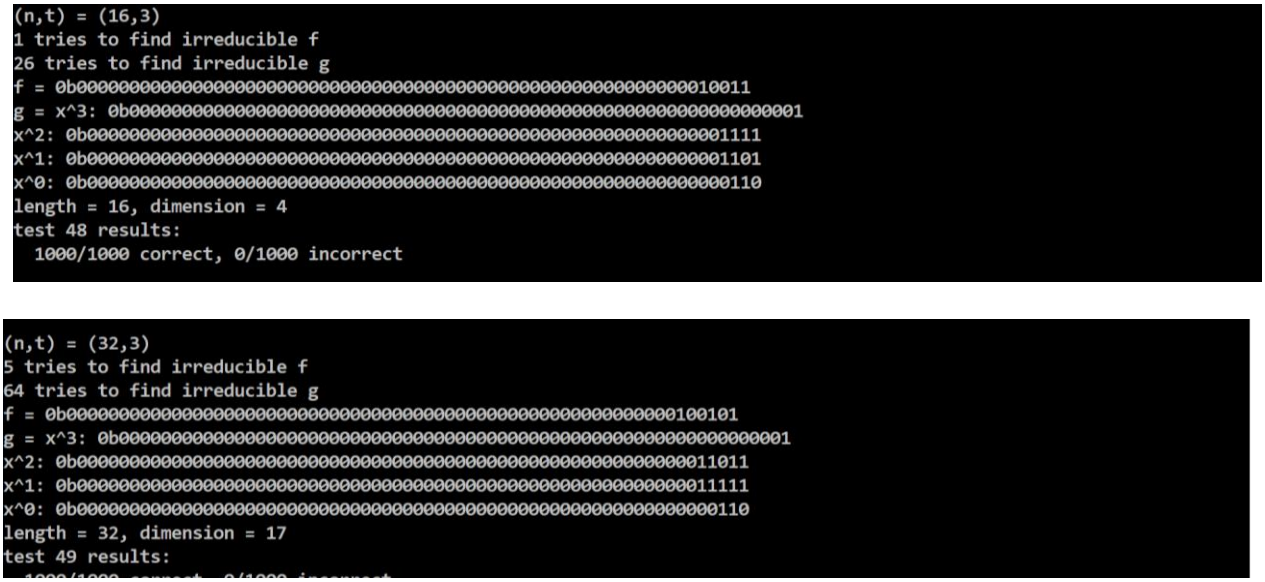


Figure 17. Example output from McEliece tests with small parameters

We also show in Figure 18 two images from larger parameter settings, n = 64, t = 3, and n = 512, t = 3. The top one was tested 1000 times and the results come out with 1000 correct decryptions, and the bottom one was tested once and that comes correctly also.

8.3 McEliece: Testing on Text

Once we were satisfied that the unit tests passed, we wanted to test our implementation of cipher block chaining. Arbitrarily we chose some of the faster parameter settings of $(8,2), (16,2), (16,3), (32,2)$, and $(32,3)$ and tested

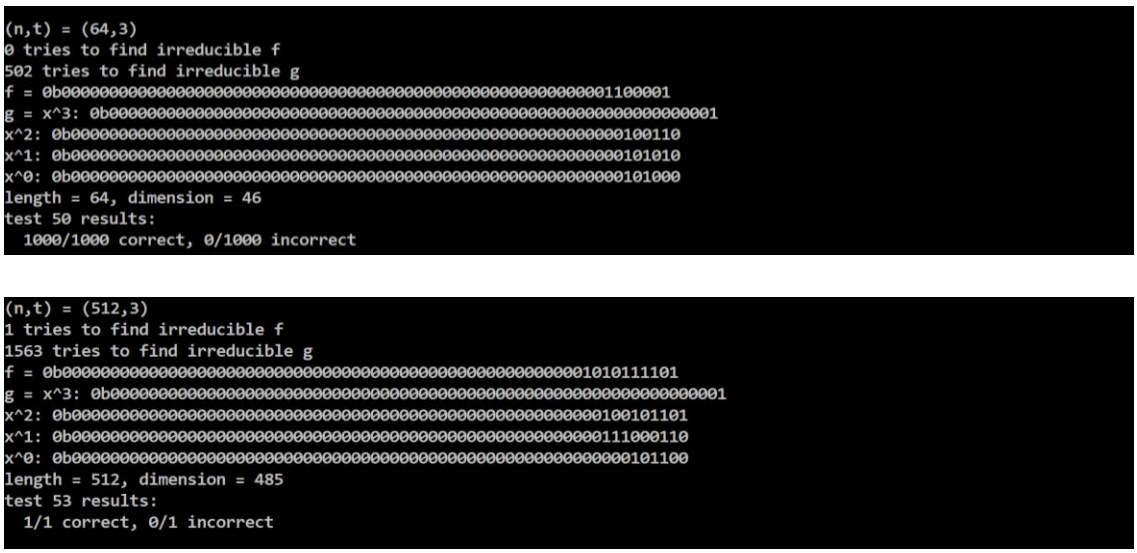


Figure 18. Example output from McEliece tests with large parameters

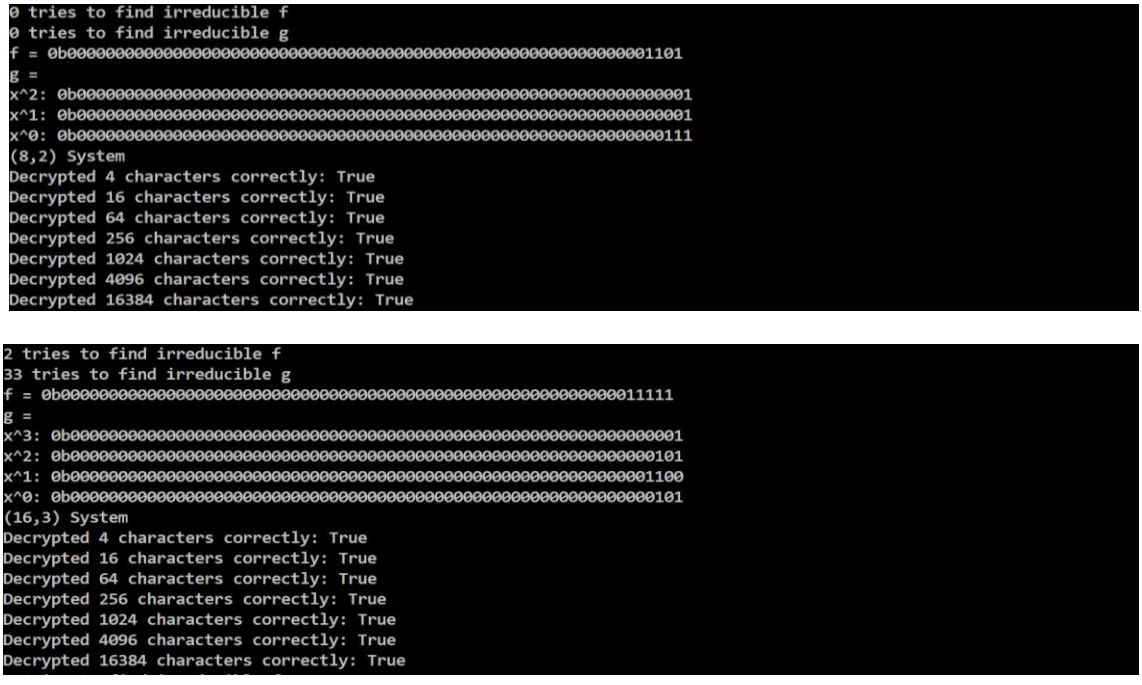


Figure 19. Example output from McEliece cipher block chaining on text

each system on 2^2, 2^4, 2^6, 2^8, 2^{10}, 2^{12}, and 2^{14} characters of Wuthering Heights. The results all come out correctly, and some screenshots are shown in Figure 19. For each of the character lengths, it prints True if the decrypted ciphertext is equal to the original message.

8.4 Regev: Initial Tests

Initial tests of the Regev system proved promising. We created three systems: one with $n = 5$, $p = 4$, and $m = 6$; one with $n = 5$, $p = 5$, and $m = 3$; and one with $n = 5$, $p = 6$, and $m = 4$. We tested all messages of length ten 100 times on each system and averaged the results, getting averages of 1022, 1020, and 1023 correct decryptions out of 1024. Screenshots of the second system are shown in Figure 20. Since we know that the system only decrypts correctly with a certain probability, it seemed like the system was going to work.

However, Regev gives parameter constraints that guarantee safety and correctness. When we adjusted the parameters to satisfy the constraints, the results were not as good. We again created three systems: one with $n = 10$, $p = 164$, and $m = 121$; one with $n = 12$, $p = 248$, and $m = 155$; and one with $n = 15, p = 345$, and $m = 202$. For each system we encrypted and decrypted all messages of length ten ten times and averaged the number of correct decryptions, getting 637, 186, and 358 average correct decryptions out of 1024.

Most messages were decrypted incorrectly in these tests. However, after checking the code to make sure all the steps were implemented correctly, we reasoned that the cause of the incorrectness might be something to do with the randomness of the subset of the public key chosen during encryption, or something to do with the choosing of the errors from the probability distribution during public key generation. So we tested the algorithm with all the errors set to zero; as expected, the results all came out correctly. In the next section, we show how we took a closer

```
iteration 92: correct = 1018/1024
iteration 92: incorrect = 6/1024

iteration 93: correct = 1018/1024
iteration 93: incorrect = 6/1024

iteration 94: correct = 1019/1024
iteration 94: incorrect = 5/1024

iteration 95: correct = 1022/1024
iteration 95: incorrect = 2/1024

iteration 96: correct = 1020/1024
iteration 96: incorrect = 4/1024

iteration 97: correct = 1021/1024
iteration 97: incorrect = 3/1024

iteration 98: correct = 1022/1024
iteration 98: incorrect = 2/1024

iteration 99: correct = 1018/1024
iteration 99: incorrect = 6/1024

average correct over 100 tests = 1020/1024
average incorrect over 100 tests = 3/1024
$
```

Figure 20. Initial tests of Regev with small parameters

look at how encryption and decryption are supposed to work. In particular, since the algorithm encrypts a single bit at a time, we look at bit-level encryption and decryption.

8.5 Regev: Finding the Acceptable Error Range

If we suppose that $m = 10$ we can write the expanded form of the public key as follows, and we highlight in yellow the subset we choose for encryption:

$[\overrightarrow{a_1},\ b_1] = [\overrightarrow{a_1},\ \langle\overrightarrow{a_1}, \vec{s}\rangle + e_1]$

$[\overrightarrow{a_2},\ b_2] = [\overrightarrow{a_2},\ \langle\overrightarrow{a_2}, \vec{s}\rangle + e_2]$

$[\overrightarrow{a_3},\ b_3] = [\overrightarrow{a_3},\ \langle\overrightarrow{a_3}, \vec{s}\rangle + e_3]$

$[\overrightarrow{a_4},\ b_4] = [\overrightarrow{a_4},\ \langle\overrightarrow{a_4}, \vec{s}\rangle + e_4]$

$[\overrightarrow{a_5},\ b_5] = [\overrightarrow{a_5},\ \langle\overrightarrow{a_5}, \vec{s}\rangle + e_5]$

$[\overrightarrow{a_6},\ b_6] = [\overrightarrow{a_6},\ \langle\overrightarrow{a_6}, \vec{s}\rangle + e_6]$

$[\overrightarrow{a_7},\ b_7] = [\overrightarrow{a_7},\ \langle\overrightarrow{a_7}, \vec{s}\rangle + e_7]$

$[\overrightarrow{a_8},\ b_8] = [\overrightarrow{a_8},\ \langle\overrightarrow{a_8}, \vec{s}\rangle + e_8]$

$[\overrightarrow{a_9},\ b_9] = [\overrightarrow{a_9},\ \langle\overrightarrow{a_9}, \vec{s}\rangle + e_9]$

$[\overrightarrow{a_{10}},\ b_{10}] = [\overrightarrow{a_{10}},\ \langle\overrightarrow{a_{10}}, \vec{s}\rangle + e_{10}]$

Then encrypting a zero becomes:

$$\begin{array}{rl} & [\overrightarrow{a_2},\ b_2] = [\overrightarrow{a_2},\ \langle\overrightarrow{a_2}, \vec{s}\rangle + e_2] \\ & [\overrightarrow{a_5},\ b_5] = [\overrightarrow{a_5},\ \langle\overrightarrow{a_5}, \vec{s}\rangle + e_5] \\ + & [\overrightarrow{a_6},\ b_6] = [\overrightarrow{a_6},\ \langle\overrightarrow{a_6}, \vec{s}\rangle + e_6] \\ \hline \end{array}$$

$$(\overrightarrow{a}, b) = [\vec{a}_{2+5+6},\ b_{2+5+6}] = [\vec{a}_{2+5+6},\ (\langle\overrightarrow{a_2}, \vec{s}\rangle + \langle\overrightarrow{a_5}, \vec{s}\rangle + \langle\overrightarrow{a_6}, \vec{s}\rangle) + \boldsymbol{e_2 + e_5 + e_6}\ ]$$

And encrypting a one becomes:

$$\begin{array}{rl} & [\overrightarrow{a_2},\ b_2] = [\overrightarrow{a_2},\ \langle\overrightarrow{a_2}, \vec{s}\rangle + e_2] \\ & [\overrightarrow{a_5},\ b_5] = [\overrightarrow{a_5},\ \langle\overrightarrow{a_5}, \vec{s}\rangle + e_5] \\ \hline + & [\overrightarrow{a_6},\ b_6] = [\overrightarrow{a_6},\ \langle\overrightarrow{a_6}, \vec{s}\rangle + e_6] \end{array}$$

$$(\vec{a}, b) = [\vec{a}_{2+5+6},\ b_{2+5+6} + \left\lfloor\frac{p}{2}\right\rfloor] = [\vec{a}_{2+5+6},\ (\langle\overrightarrow{a_2}, \vec{s}\rangle + \langle\overrightarrow{a_5}, \vec{s}\rangle + \langle\overrightarrow{a_6}, \vec{s}\rangle) + \left\lfloor\frac{p}{2}\right\rfloor \boldsymbol{+ e_2 + e_5 + e_6}]$$

From looking at the expanded form we can see that if we use the equality

$\langle\overrightarrow{a_2}, \vec{s}\rangle + \langle\overrightarrow{a_5}, \vec{s}\rangle + \langle\overrightarrow{a_6}, \vec{s}\rangle = \langle\vec{a}_{2+5+6}, \vec{s}\rangle$

Then decrypting the zero with no error becomes

$b - \langle\vec{a}, \vec{s}\rangle = \langle\vec{a}_{2+5+6}, \vec{s}\rangle - \langle\vec{a}_{2+5+6}\vec{s}\rangle = 0$

And decrypting the one with no error becomes

$b - \langle\vec{a}, \vec{s}\rangle = \langle\vec{a}_{2+5+6}, \vec{s}\rangle - \langle\vec{a}_{2+5+6}\vec{s}\rangle + \left\lfloor\frac{p}{2}\right\rfloor = \left\lfloor\frac{p}{2}\right\rfloor$

And decrypting the one with some error becomes

$b - \langle \vec{a}, \vec{s} \rangle = \langle \vec{a}_{2+5+6}, \vec{s} \rangle - (\langle \vec{a}_{2+5+6} \vec{s} \rangle + \boldsymbol{e_2} + \boldsymbol{e_5} + \boldsymbol{e_6}) = \boldsymbol{e_2} + \boldsymbol{e_5} + \boldsymbol{e_6}$

And decrypting the zero with some error becomes

$b - \langle \vec{a}, \vec{s} \rangle = \langle \vec{a}_{2+5+6}, \vec{s} \rangle - (\langle \vec{a}_{2+5+6} \vec{s} \rangle + \left\lfloor \frac{p}{2} \right\rfloor + \boldsymbol{e_2} + \boldsymbol{e_5} + \boldsymbol{e_6}) = \left\lfloor \frac{p}{2} \right\rfloor \boldsymbol{e_2} + \boldsymbol{e_5} + \boldsymbol{e_6}$.

Then we again take a look at the decryption step which uses circular distance to check if the computed quantity is closer to 0 than to $\left\lfloor \frac{p}{2} \right\rfloor$. In particular, we imagine that $p = 10$ and we create a circle and an unwrapped circle that represent values closer to 0 in green and values closer to $\left\lfloor \frac{p}{2} \right\rfloor = 5$ in blue. These are shown below, with arrows pointing to zero and five.

We know that without adding any error the quantity $b - \langle s, a \rangle$ will be 0 for a 0 bit and the quantity $b - \langle s, a \rangle$ will be 5 for the 1 bit. So from looking at these images we can tell that in order to decrypt a bit correctly, the sum of the errors in the subset mod p must be 0, 1, or 2 so that the sum will not become closer to the other number, or it must be 8 or 9, so that it loops back around and is still closer to the right number. From these observations we can reason that there are two acceptable error ranges:

$0 \leq \sum_{i \in S} e_i \, mod \, p \leq \left\lfloor \frac{\lfloor \frac{p}{2} \rfloor}{2} \right\rfloor$ or $p - \left\lfloor \frac{\lfloor \frac{p}{2} \rfloor}{2} \right\rfloor \leq \sum_{i \in S} e_i \, mod \, p \leq p - 1$.

We can now provide an explanation for the error distribution, again shown in Figure 21.

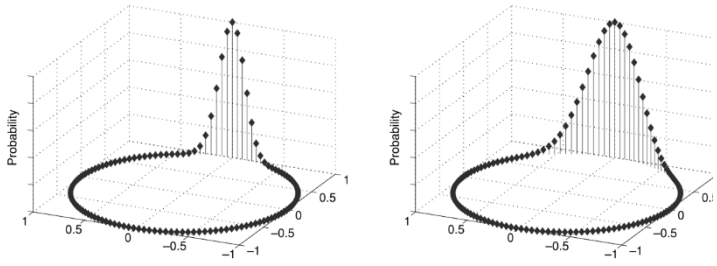


Figure 21. Error distribution in Regev

The idea is that most error numbers picked will be either closer to 0 or closer to p so that the chances that their sum modulo p is in the one of the acceptable ranges will be high.

8.6 Regev: Testing the Acceptable Error Range

We next test out the acceptable error ranges by testing one bit at a time and modifying the code to keep track of the errors of the subset chosen in the encryption step. As an informal test, we first create a system with n = 10, p = 199,

and m = 126 and print out the acceptable error ranges. Then we encrypt and decrypt 5 zero bits and 5 one bits along with the error sums corresponding to each. The results are shown in Figure 22..

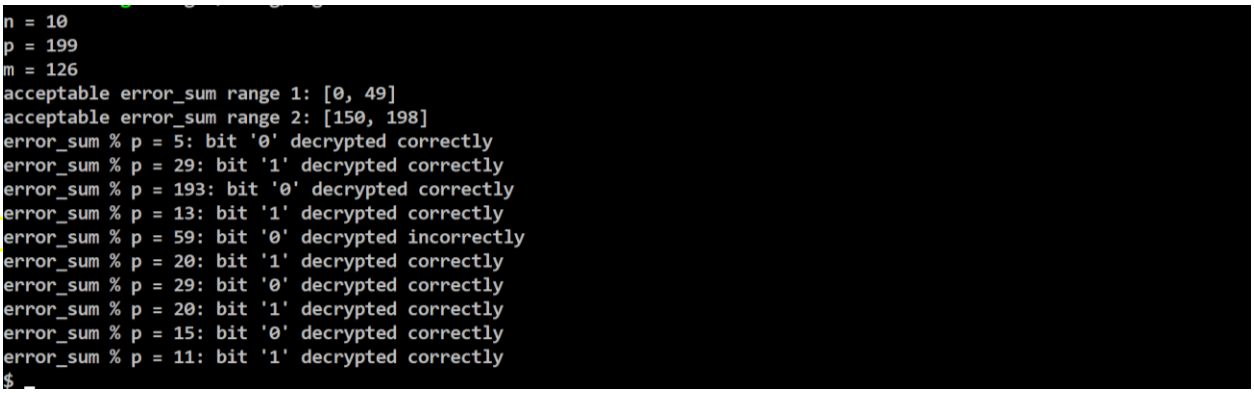
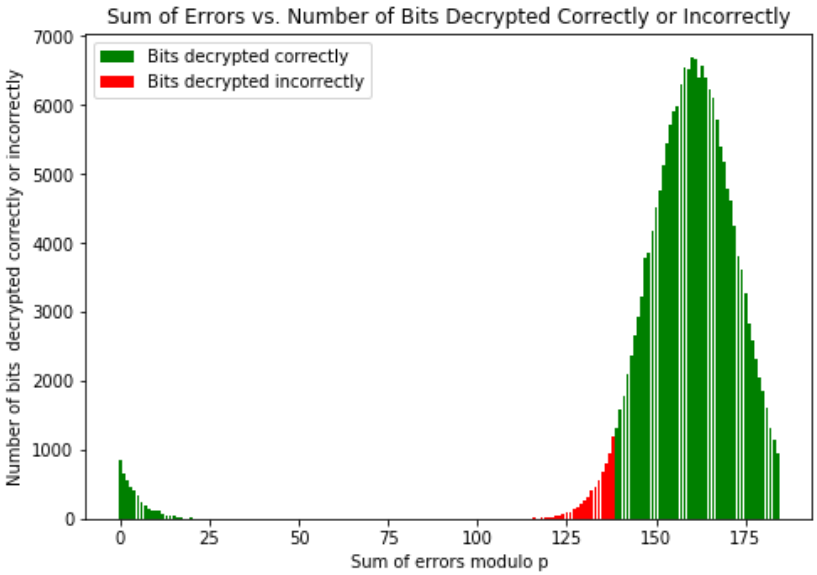


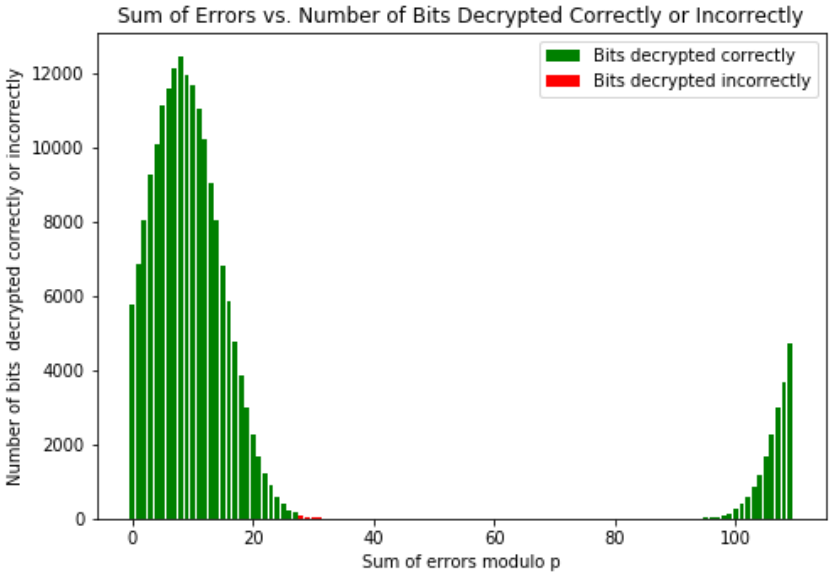
Figure 22. Decrypted bits and their error sums

From the image it is clear that the single bit decrypted incorrectly had an error sum of 59 which was not in either of the acceptable error ranges, and that all of the bits decrypted correctly had an error sum in one of the acceptable error ranges.

To conduct a more thorough test, we created two systems, one with n = 10, p = 185, and m = 124, and one with n = 10, p = 110, and m = 112, and encrypted and decrypted 100,000 zeros and 100,000 ones. For each system we used two hash tables, one for correct decryptions and one for incorrect decryptions, with values 0 through p-1 as the keys and the number of bits decrypted correctly or incorrectly as the values. For each correctly decrypted bit we incremented the count of that error sum in the correct table and for each incorrectly decrypted bit we incremented the count of the error sum in the incorrect table. The results are shown in Graph 1 and Graph 2.



Graph 1. Decryptions for n = 10, p = 185, m = 124 and acceptable error_sum range 1: [0, 46], acceptable error_sum range 2: [139, 184]



Graph 2. Decryptions for n = 10, p = 110, m = 112 and acceptable error_sum range 1: [0, 27], acceptable error_sum range 2: [83, 109]

The graphs, for an x-axis in the range of the modulus, show both the number of correct decryptions and incorrect decryptions per error sum. The correct are shown in green and the incorrect in red. It is clear that there are no incorrect decryptions when the modulus is in the range that we guessed it should be in.

Finally, since we established that the system does actually work the way it is supposed to, we decided to retest some systems on only a single bit at a time instead of a whole message. We tested five different systems on

10,000 zero bits and 10,000 one bits and averaged the bits decrypted correctly and the bits decrypted incorrectly. A screenshot of the results are shown in Figure 23.

Figure 23. Average number of bits decrypted correctly

As is shown in the image, the results are actually over 99 percent correct, specifically with 99.14% of the bits being decrypted correctly and 0.86% being decrypted incorrectly. We are satisfied with the success of these results and we realized that in our initial tests, if a single bit in a message is decrypted incorrectly, the entire message would be decrypted incorrectly, so this is why our initial results seemed so poor. A better test might be, for example, to count the number of correct bits per message instead of the number correct messages overall.

9. EXPERIMENTS

We have established that quantum-resistant encryption algorithms are important to develop given the threat of quantum computers in the near future. We have given two examples and discussed one way each might be implemented. But other than the difficulty in developing them, is there a cost associated with using them? In this chapter we examine this question by comparing features of quantum-resistant algorithms with non-quantum-resistant algorithms.

9.1 Choosing a Yardstick

So far we have discussed a code-based post-quantum algorithm and a lattice-based post-quantum algorithm. We have established that since they are based on problems that are supposed to be hard, the only way they should be able to be broken by a quantum or non-quantum computer is by some form of a brute-force attack. It seems that a reasonable way to compare the cost of using these systems would be to set them at parameters that provide an equal work factor of security, for example, we could set each system's parameters to the values that would require 2^{100} guesses in a brute-force search. This would allow us to compare the performance of the systems at equal levels of security. However, we also need to compare the post-quantum systems with non-post-quantum systems. Since we are measuring the cost of the *replacement* of these non-post-quantum systems with post-quantum systems on classical computers, we can ignore the fact that they can be broken by quantum computers and simply use the same strategy, setting them at parameters that would require some work factor in a brute-force search. In our experiments, we choose RSA as the non-post-quantum system for comparison.

9.2 Setting the X-axis

McEliece proposes two basic attacks in his original paper; the first, to try to recover the unscrambled generator matrix G; the second, to try to recover the message from the codeword without G. In the first case, since G' = SGP, the attacker would have to brute-force guess G, and then brute force guess S and P to scramble G. Consider that a binary m by n matrix has mn elements, each of which can be 0 or 1. This creates 2^{mn} possibilities for a binary m by n matrix. Here, since G is n elements in length and at least $n - mt$ elements in height, the best case for the attacker would be that the matrix G has $2^{n(n-mt)}$ possibilities = $2^{O(n^2)}$ possibilities. Similarly, the work factor for guessing S would be $2^{(n-mt)^2}$ and the work factor for guessing P, a permutation matrix, would be $n!$, even worse [51]. So the total time to brute-force guess the private key would be $2^{O(n^2)} * 2^{O(n^2)} * n! = O(n!)$. Similarly, as McEliece describes, a brute force attack based on decoding the linear code would be equivalent to an exhaustive search over all possible messages and would require $2^{dimension} = 2^k$ guesses. We will use these work factors in our experiments.

Similarly, recall that Regev states the LWE problem can be turned into a Maximum Likelihood Problem that can be solved in $2^{O(nlogn)}$ time, and that an algorithm was developed in 2003 that can solve the same problem in $2^{O(n)}$ time [2]. When setting parameters, we assume that these runtimes or "work factors" can be equated with our analysis from McEliece; in other words, we assume that $2^{O(n)}$ time implies the same number of brute force guesses.

Finally for RSA, we assume that the private key is the same length as the public key and that a brute-force search on the private key takes 2^n guesses where n is the length of the public key. Our experiments are thus run on the parameters shown below. In Table 4, the security parameters listed in the columns result in the work factor to the left for the attack in the corresponding column. The shaded cells represent non-runnable parameters.

Table 4. Parameters for equivalent work factors

	Work factor	McEliece $O(n!)$ Attack	McEliece $O(2^n)$ Attack	Regev $O(2^n)$ Attack	Regev $O(2^{nlogn})$ Attack	RSA $O(2^n)$Attack
security parameters:		(n,k)	(n,k)	(n,E)	(n,E)	n
	2^1	$(2,2) = (2,2)$	(1,2)	(1,0.5)	(2,0.5)	1
	2^2	$(2,2) = (2,2)$	(2,2)	(2,0.5)	(2,0.5)	2
	2^4	$(3,2) = (4,2)$	(4,2)	(4,0.5)	(3,0.5)	4
	2^8	$(5,2) = (8,2)$	(8,2)	(8,0.5)	(4,0.5)	8
	2^{16}	(8,2)	(16,2)	(16,0.5)	(7,0.5)	16
	2^{32}	$(13,2) = (16,2)$	(32,2)	(32,0.5)	(10,0.5)	32
	2^{64}	$(20,2) = (32,2)$	(64,2)	(64,0.5)	(16,0.5)	64
	2^{128}	$(34,2) = (64,2)$	(128,2)	(128,0.5)	(27,0.5)	128
	2^{256}	$(57,2) = (64,2)$	(256,2)	(256,0.5)	(47,0.5)	256
	2^{512}	$(98,2) = (128,2)$	(512,2)	(512,0.5)	(81,0.5)	512
	2^{1024}	$(171,2) = (256,2)$	(1024,2)	(1024,0.5)	(160,0.5)	1024
	2^{2048}	$(301,2) = (512,2)$	(2048,2)	(2048,0.5)	(318,0.5)	2048

For example, this table shows that the McEliece system set at (64,2) would result in the work factor of 2^{128} for a brute-force search on the permutation matrix which has $n!$ possibilities. And the Regev system set at (128, 0.5) would result in the same work factor for the fastest known attack, the $O(2^n)$ attack. By using the work factor as the x-axis, we can compare the costs of the three systems at the same level of security. Finally, recall the 256-bit AES key that supposedly will be secure forever. Using the same logic we can assume that the 2^{256} row in the graph represents a solid work factor for a secure system today. Larger work factors are shown for theoretical purposes.

9.3 Time and Space Complexity

We use the breakdowns in Tables 5 through 9 to hypothesize the shape of the graphs for each experiment. For the Regev system, note that m and p are functions of n: $m = (1+\epsilon)(1+n)\log p = O(n \log n^2)$ and $n^2 \leq p \leq 2n^2$ so expressions including them can be written in terms of n. The following notation is used:

t - degree of irreducible polynomial $g(x)$ over $GF(2^m)$

m - degree of irreducible polynomial $f(x)$ over $GF(2)$

d - degree of largest degree input polynomial

i - exponent

n - the length of the linear code, equal to 2^m

k - dimension of generator matrix, equal to number of columns in the nullspace of the parity check matrix, constraint: $n - tm \leq k \leq n$

Table 5. Time complexity of field operations in McEliece

Operation	GF(2^{mt}) Complexity	GF(2^m) Complexity
`gcd()`	$O(d^2m) = O(d^2)$	$O(d)$
`inverse()`	$O(t^2m) = O(t^2)$	$O(m)$
`reduce()`	$O(d^2m) = O(d^2)$	$O(d)$
`add()`	$O(t)$	$O(1)$
`is_irreducible()`	$O(2^{mt})$	$O(m^2)$
`get_ax_bx()`	$O(d^2m) = O(d^2)$	--
`square()`	$O(d^2 + dm) = O(d^2)$	--
`sqrt()`	$O(t^3m + t^2m^2) = O(t^3)$	--
`compute()`	$O(mt^2) = O(t^2)$	--
`multiply()`	--	$O(m)$
`pow()`	--	$O(mi)$
`create_L()`		$O(mt^2n)$

Table 6. Time complexity of matrix operations in McEliece

Operation	Complexity
`nullspace()`	$O(mtn)$
`rref()`	$O(mtn)$
`x()`	$O(t^2)$
`y()`	$O(mt^2n)$
`z()`	$O(mt^2n)$

`multiply()`	$O(xrows * ycols * xcols * m)$
`binary_matrix()`	$O(tmn)$
`dense_nonsingular_matrix()`	$O(n^2)$
`permutation_matrix()`	$O(k^2)$

Table 7. Time complexity of key generation operations in McEliece

Operation	Complexity
`x`	$O(t^2)$
`y`	$O(mt^2n)$
`z`	$O(mt^2n)$
`x*y`	$O(t^2mn)$
`xy*z`	$O(tmn^2)$
`to_binary tm x length`	$O(tmn)$
`rref tm x length`	$O(tmn)$
`nullspace tm x length`	$O(tmn)$
`generate permutation matrix`	$O(n^2)$
`generate dense nonsingular matrix`	$O(k^2)$
`scramble generator matrix`	$O(k^2n + kn^2 + kn)$
`mod2 scrambled matrix`	$O(tmn)$
`total`	$O(n^2)$

Table 8. Time complexity of encryption in McEliece

Operation	Complexity
`m*G`	$O(kn)$
`mod2(mG)`	$O(n)$
`total`	$O(kn) = O(n^2)$

Table 9. Time Complexity of decryption in McEliece

Operation	Complexity
`syndrome()`	$O(t^2mn)$
`sigma()`	$O(t^3m + t^2m^2)$
`error_vector()`	$O(t^2mn) + O(t^3m + t^2m^2) + O(mt^2)$
`P.inv()`	$O(n^3)$
`y*P.inv()`	$O(n^2)$
`find moved errors`	$O(t^2mlength) + O(t^3m + t^2m^2) + O(mt^2)$

`add moved errors`	$O(n)$
`compute find_mS`	$O(kn)$
`rref find_mS`	$O(kn)$
`S.inv()`	$O(n^2)$
`mS*S.inv()`	$O(k^2)$
`total`	$O(n^3)$

Table 10. Time complexity of key generation in Regev

Operation	Complexity
`m dot products, n multiplications per dot product`	$O(mn) = O(n^2 \log n^2)$

Table 11. Time Complexity of encryption in Regev

Operation	Complexity
`2 summations of at most m vectors of length n, per bit to be encrypted`	$O(mn) = O(n^2 \log n^2) *$ `number of bits in message`

Table 12. Time complexity of decryption in Regev

Operation	Complexity
`1 dot product of length n vector, 3 subtractions`	$O(n) *$ `number of bits in message`

Table 13. Space complexity of public key in McEliece

Variable	# Bits to store
`generator`	$k * n *$ 64 bits
`length`	64 bits
`dimension`	64 bits
`t`	64 bits
`public key`	$O(n^2)$

Table 14. Space complexity of public key in Regev

Variable	# Bits to store
`pair`	$64 + n * 64$
`m`	64 bits

n	64 bits
p	64 bits
list	$m * (64 + n * 64)$ bits
public key	$O(n^2 \log n^2)$

9.4 Experiment 1: Work Factor vs. Public Key Size

In this experiment we vary our systems across the parameters listed above and measure the size of the public key. Since the public key is what is distributed, we want it to be as small as possible. In general we want the public key to be less than 1 Kilobyte [27].

Hypothesis

Below we show our general expectation of the shape of each graph.

Table 15. Hypothesis for public key sizes

	McEliece	Regev	RSA
public key	$O(n^2)$	$O(mn)$	$O(n)$

Based off of our space complexity analysis, we can predict the general shape of each graph, but we cannot predict which system will have the largest key size at a particular point. Based off of our hypothesis here, we expect Regev key size to grow the fastest, followed by McEliece. We expect RSA key size to be linear in the security parameter.

Results 1

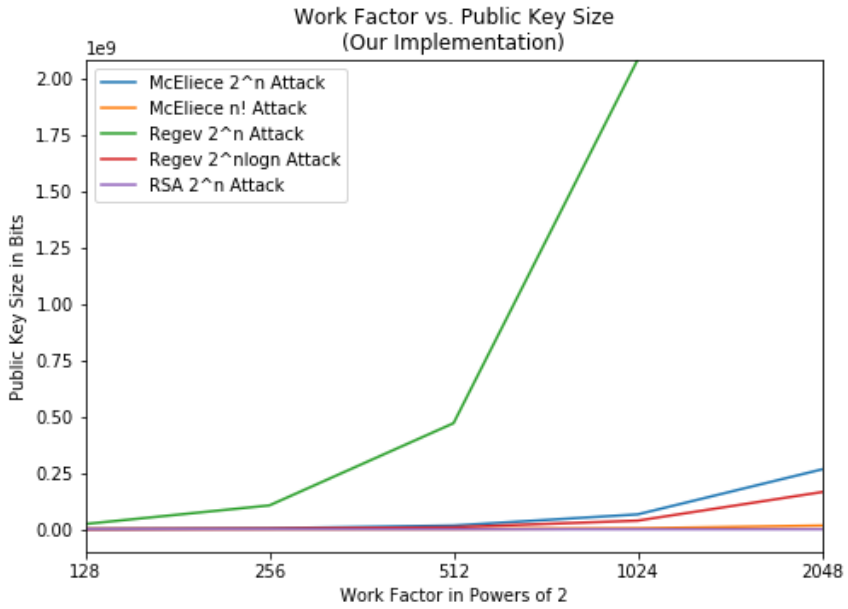
The resulting key sizes from our implementation are shown below, with the keys less than 1 Kilobyte highlighted in yellow.

Table 16. Key sizes of McEliece, Regev and RSA

	Work factor	McEliece $O(n!)$ Attack	McEliece $O(2^n)$ Attack	Regev $O(2^n)$ Attack	Regev $O(2^{nlogn})$ Attack	RSA Attack $O(2^n)$
public key size		bits	bits	bits	bits	bits
	2^1				1728	
	2^2			1472	1728	
	2^4			8128	4800	
	2^8	1216	1216	48320	7872	

	2^{16}	1216	8384	234688	35136	
	2^{32}	8384	45248	1034432	79552	
	2^{64}	45248	213184	4968640	222400	
	2^{128}	213184	934080	23019712	732864	192
	2^{256}	213184	3932352	105545920	2614144	320
	2^{512}	934080	16187584	470614208	8719680	576
	2^{1024}	3932352	65798336	2084700352	37970112	1088
	2^{2048}	16187584	265552064	8989442240	164810688	2112

And we graph the results:



Graph 3. Key sizes of McEliece, Regev, and RSA

Analysis 1

The graph shows that the Regev key does grow the fastest, as expected, and the McEliece public key grows the second fastest, as expected. This indicates that as security increases, the bandwidth required to send a post-quantum key increases much faster than a pre-quantum public key, which is undesirable. We can also tell that for all work factors, the RSA key is smaller than all other keys, which indicates that regardless of growth, a pre-quantum key is always smaller. Finally, from looking at the highlighted cells in the table, it does not appear that any of the post-quantum systems meet the requirement of a 1 Kilobyte key for any reasonable security factor. Specifically, the 2^{256} row of the table looks like this:

Table 17. Row 2^{256} Key Sizes

	Work factor	McEliece $O(n!)$	McEliece $O(2^n)$	Regev $O(2^n)$	Regev $O(2^{nlogn})$	RSA $O(2^n)$
	2^{256}	213184	3932352	105545920	2614144	320

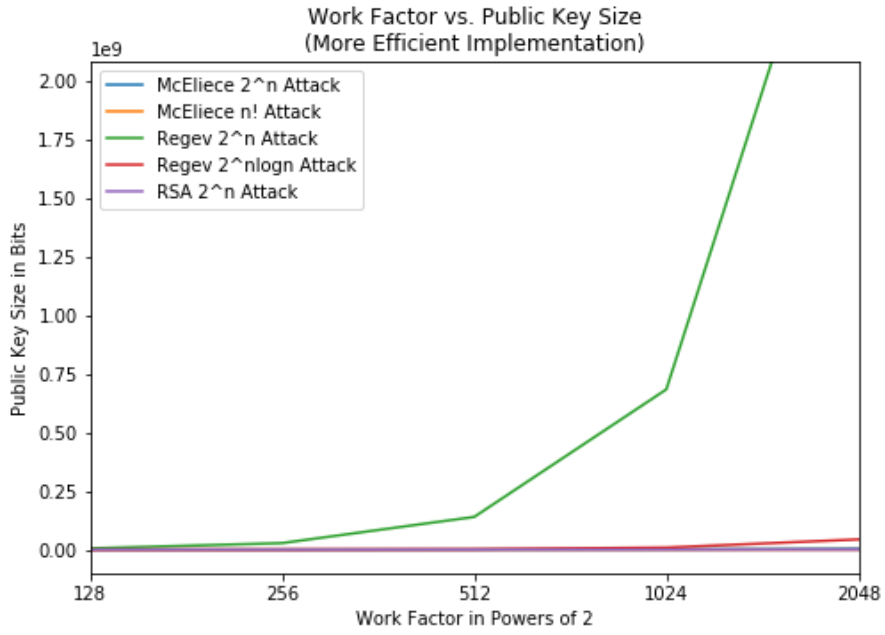
And it is clear that all keys are significantly larger than the RSA key on the right. From these results we can conclude that with regard to public key size, the cost of replacing pre-quantum systems with post-quantum systems is likely to be high.

However, in our implementation we store every single number as a `u64`, even binary numbers. So as a followup experiment, we adjusted the key size to count only bits. We calculated the public key size using the formula $3 * u64 + n * k * u64$ for the McEliece public key and the formula $3 * 64 + m * n * (\lfloor log_2(p) \rfloor + 1)$ for the Regev public key since each number is between 0 and p. The adjusted results are shown below.

Results 2

Table 18. Key sizes of McEliece, Regev, and RSA

	Work factor	McEliece $O(n!)$ Attack	McEliece $O(2^n)$ Attack	Regev $O(2^n)$ Attack	Regev $O(2^{nlogn})$ Attack	RSA Attack $O(2^n)$
public key size		bits	bits	bits	bits	bits
	2^1				264	
	2^2			252	264	
	2^4			812	480	
	2^8	208	208	5456	792	
	2^{16}	208	320	33168	3468	
	2^{32}	320	896	177952	8872	
	2^{64}	896	3520	1009408	31440	
	2^{128}	3520	14784	5395392	114672	192
	2^{256}	3520	61632	28035317	490308	320
	2^{512}	14784	253120	139713728	1771338	576
	2^{1024}	61632	1028288	684042495	8899437	1088
	2^{2048}	253120	4149440	3230580997	43778031	2112



Graph 4. Key sizes of McEliece, Regev, and RSA

Analysis 2

The graph looks a little bit better than the previous graph, but the Regev public key still grows very fast. However in the table, many more keys fall under 1 Kilobyte as desired. Specifically, the 2^{256} row looks like the following:

Table 19. Row 2^{256} Key Sizes

Work factor	McEliece $O(n!)$	McEliece $O(2^n)$	Regev $O(2^n)$	Regev $O(2^{nlogn})$	RSA $O(2^n)$
2^{256}	3520	61632	1649152 * 17 + 192=28035317	40843*12+192=490308	320

So the McEliece public key does appear to be an efficient size and therefore useable for a strong security factor today.

9.5 Experiment 2: Work Factor vs. Key Generation Speed

In this experiment we vary our systems across the parameters that result in the work factors above and measure the speed of key generation. Additionally, since both the McEliece and Regev systems include randomness, we ran each system's key generation multiple times and took the average. For Regev for all security factors, we ran each key generation 100 times. For McEliece parameters up to 2^{32} we ran 100 times. For 2^{64} through 2^{512} we ran

10 times, and for 2^{1024} and 2^{2048} we ran twice. Finally, since cryptosystems are needed so regularly we can assume key generation should take less than one second.

Hypothesis

Table 20. Hypothesis for key generation speed

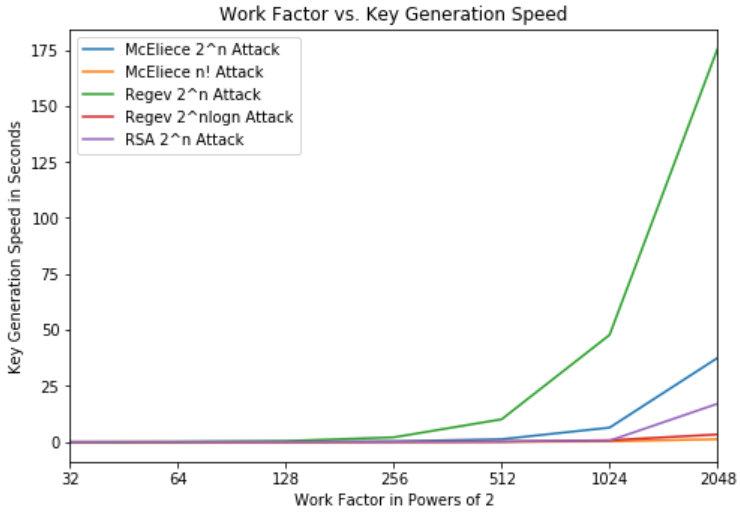
	McEliece	Regev	RSA
key generation speed	$O(n^2)$	$O(mn) = O(n^2 log n^2)$	$O(n^2)$

For our hypothesis we can again predict the general shapes of the graphs; we expect that Regev key generation speed will grow the fastest, and that McEliece and RSA will grow similarly. But until we see the results we cannot know which systems actually take longer at a particular level of security.

Results

Table 21. Key generation speeds of McEliece, Regev, and RSA

	Work factor	McEliece $O(n!)$ Attack	McEliece $O(2^n)$ Attack	Regev $O(2^n)$ Attack	Regev $O(2^{nlogn})$ Attack	RSA Attack $O(2^n)$
key generation time		seconds	seconds	seconds	seconds	seconds
	2^1				.000123	
	2^2			.000112392	.000055	
	2^4			.000212368	.000130	
	2^8	.000784	.000918	.001173041	.000220	
	2^{16}	.000512	.001768	.004583284	.000790	.010810
	2^{32}	.001943	.006540	.020853274	.001781	.003524
	2^{64}	.005974	.020723	.102617203	.004651	.003550
	2^{128}	.020653	.074750	.44960736	.015713	.004828
	2^{256}	.019606	.273176	2.0868353	.054341	.031310
	2^{512}	.075963	1.24842	10.126917	.177354	.103344
	2^{1024}	.274670	6.39750	47.790256	.780439	.602274
	2^{2048}	1.239622	37.5098	175.428124	3.354826	17.069227



Graph 5. Key generation speeds of McEliece, Regev, and RSA

Analysis

From the graph is clear that Regev key generation time grows the fastest as expected. We can also see that McEliece and RSA appear to be quadratic as expected. Most importantly, for realistic security factors, the three systems appear somewhat comparable. Specifically, the row 2^{256} appears as follows:

Table 22. Row 2^{256} Key Generation Speed

Work factor	McEliece $O(n!)$	McEliece $O(2^n)$	Regev $O(2^n)$	Regev $O(2^{nlogn})$	RSA $O(2^n)$
2^{256}	.019606	.273176	2.0868353	.054341	.031310

So for the same security factor as a 256-bit AES key, the McEliece system key generation takes about the same time as RSA key generation, which means that in terms of key generation time, replacing the pre-quantum system with a post-quantum system is reasonable and does not come at great cost. The Regev system key generation, at two seconds, could even be considered reasonable.

9.6 Experiment 3: Work Factor vs. Encryption Speed

In this experiment we vary our systems across the parameters that result in the work factors above and measure the speed of encryption. Again, since encryption involves randomness in the McEliece and Regev systems, for each security factor we ran encryption multiple times and averaged the results. For McEliece factors up to 2^{512} we averaged encryption time over 100 encryptions and for 2^{1024} and 2^{2048} we averaged encryption time over 10 encryptions. For Regev factors up to 2^{32} we averaged 100 encryptions, for the factor 2^{64} we averaged 10, for 2^{128} we averaged 2, and for 2^{256} through 2^{512} we only timed one encryption. For RSA, we implemented cipher block

chaining and for each system we set the block size to the maximum size runnable. These block sizes were 5 characters for $n = 128$, 20 characters for $n = 256$, 50 characters for $n = 512$, 110 characters for $n = 1024$, and 240 characters for $n = 2048$. Finally, since public keys are used for symmetric key exchange, we we used a 256-bit AES key as the message to be encrypted.

Hypothesis

Table 23. Hypothesis for encryption speed

	McEliece	Regev	RSA
encryption speed	$O(n^2)$	$O(mn) = O(n^2 log n^2)$	$O(n^2)$

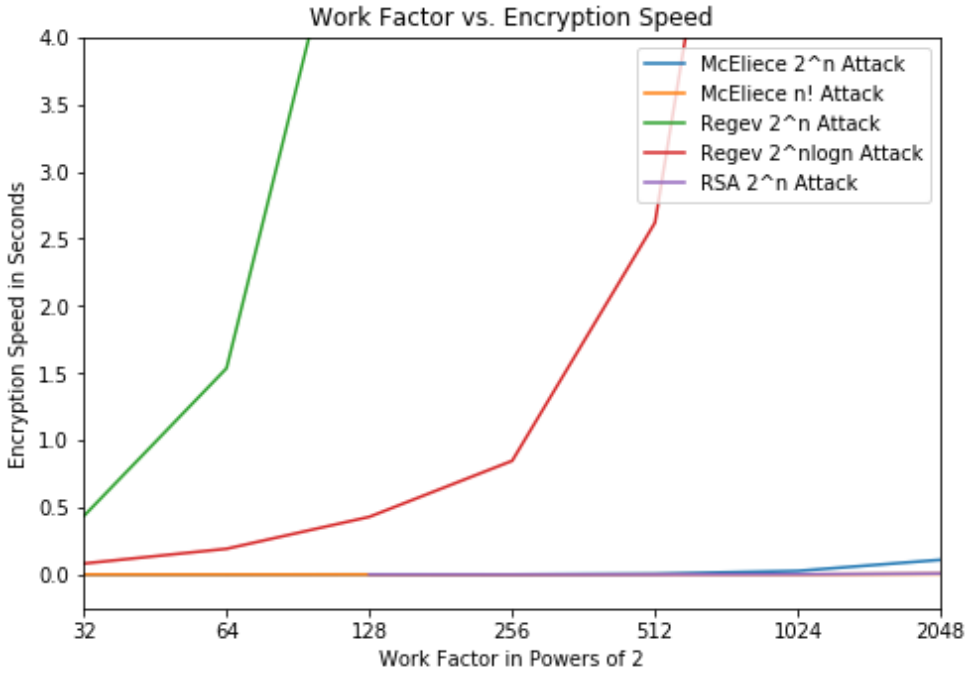
Again we expect Regev to grow the fastest and the shape of McEliece and RSA to look similar.

Results

The results are shown below where "inf" means greater than 5 minutes.

Table 24. Encryption speeds of McEliece, Regev, and RSA

	Work factor	McEliece $O(n!)$ Attack	McEliece $O(2^n)$ Attack	Regev $O(2^n)$ Attack	Regev $O(2^{nlogn})$ Attack	RSA Attack $O(2^n)$
encryption speed		seconds	seconds	seconds	seconds	seconds
	2^1				.003851	
	2^2			.004972	.003950	
	2^4			.012093	.007741	
	2^8	.002640	.002525	.041004	.014580	
	2^{16}	.002546	.001736	.133774	.030920	
	2^{32}	.001144	.001083	.435323	.082704	
	2^{64}	.000904	.001130	1.535871	.192512	
	2^{128}	.001087	.002128	5.8007805	.430259	.001512
	2^{256}	.001027	.001821	24.071204	.847360	.000834
	2^{512}	.002385	.006608	98.704114	2.620565	.000984
	2^{1024}	.001593	.027680	inf	9.082701	.002692
	2^{2048}	.006030	.110428	inf	37.743688	.008710



Graph 6. Encryption speeds of McEliece, Regev, and RSA

Analysis

From the graph we can indeed see that Regev encryption grows very quickly, perhaps even more quickly than expected, which means that if Regev were the replacement system, increasing the security in a post-quantum system would come at a much higher cost than increasing the security in a pre-quantum system. However, from the table, we can see that McEliece and RSA encryptions are similar and remain feasible even up to the highest work factors. Specifically, the 2^{256} row looks like the following:

Table 25. Row 2^{256} Encryption Speeds

Work factor	McEliece $O(n!)$	McEliece $O(2^n)$	Regev $O(2^n)$	Regev $O(2^{nlogn})$	RSA $O(2^n)$
2^{256}	.001027	.001821	24.071204	.847360	.000834

From this we can conclude that if Regev were the replacement system, the cost of replacement in terms of encryption speed would be unacceptable at 24 seconds. However, if McEliece were the replacement system, the cost of replacement in terms of encryption speed would be very low, making McEliece a viable candidate.

9.7 Experiment 4: Work Factor vs. Decryption Speed

In this experiment we vary our systems across the parameters that result in the work factors above and measure the speed of decryption. Again we use the 256-bit AES key as the message to be decrypted. And again to accommodate

randomness we averaged the decryption time over 100 messages for McEliece factors up to 2^{64}, 10 messages for 2^{128}, and one message for 2^{512}. Since the encryptions for the factors 2^{1024} and 2^{2048} took too long we could not time those decryptions and put a question mark in those cells in the graph.

Hypothesis

Table 26. Hypothesis for decryption speed

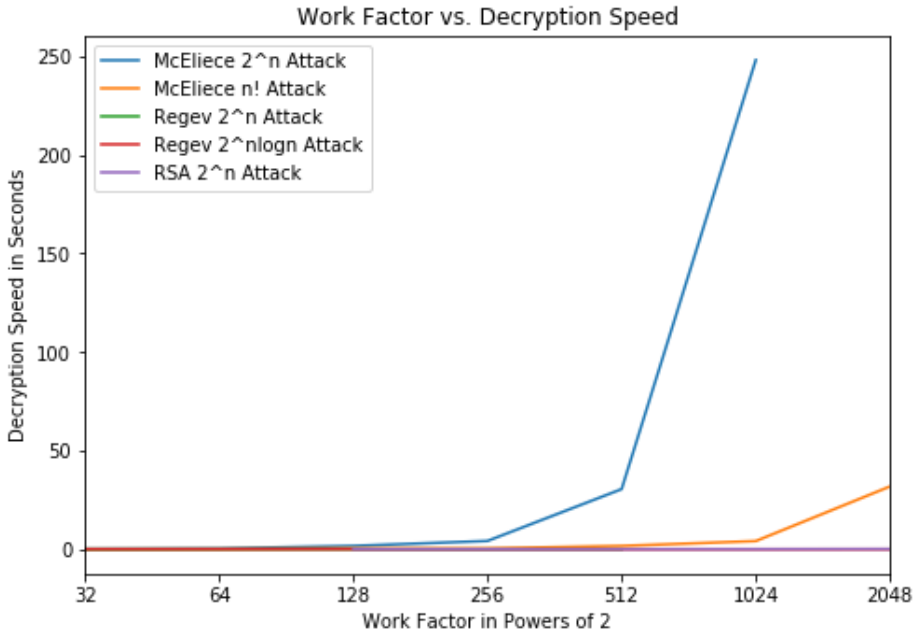
	McEliece	Regev	RSA
decryption speed	$O(n^3)$	$O(n)$ per bit	$O(n^3)$

For the decryption speed we expect the shape of McEliece and RSA to be similar. We expect Regev decryption speed to grow linearly and it should be faster than McEliece and RSA at all security levels.

Results

Table 27. Decryption Speeds of of McEliece, Regev, and RSA

	Work factor	McEliece $O(n!)$ Attack	McEliece $O(2^n)$ Attack	Regev $O(2^n)$ Attack	Regev $O(2^{nlogn})$ Attack	RSA Attack $O(2^n)$
decryption speed		seconds	seconds	seconds	seconds	seconds
	2^1				.000217	
	2^2			.000231	.000280	
	2^4			.000255	.000242	
	2^8	.057649	.058582	.000326	.000256	
	2^{16}	.062007	.072849	.000500	.000324	
	2^{32}	.068922	.125848	.000780	.000395	
	2^{64}	.141895	.387596	.001371	.000515	
	2^{128}	.388524	1.656945	.002543	.000643	.002256
	2^{256}	.388930	4.169249	.004916	.001085	.002033
	2^{512}	1.643109	30.439314	.009317	.001760	.003246
	2^{1024}	4.112714	247.936048	?	.003120	.014033
	2^{2048}	31.787255	inf	?	.005721	.083484



Graph 7. Decryption Speed of of McEliece, Regev, and RSA

Analysis

From the graph we can see that the shape of McEliece does grow the fastest. But from some investigation we also know that computing the inverse of the two $O(n)$ by $O(n)$ matrices is the most expensive operation in decryption in McEliece. In reality we can expect the private key holder to have these inverses pre-computed. Meanwhile Regev decryption is very fast and appears to grow linearly as expected. From these observations we can say that in terms of decryption speed, the cost of replacing RSA with Regev would be low and thus feasible, while the cost of replacing RSA with McEliece might be unreasonable.

More specifically, the 2^{256} row of the table looks like the following:

Table 28. Row 2^{256} Decryption Speeds

Work factor	McEliece $O(n!)$	McEliece $O(2^n)$	Regev $O(2^n)$	Regev $O(2^{nlogn})$	RSA $O(2^n)$
2^{256}	.388930	4.169249	.004916	.001085	.002033

This table confirms that Regev would be a good replacement for RSA in the decryption category. McEliece decryption at four seconds takes slightly too long.

9.8 Experiment 5: Work Factor vs. Encrypted Message Size

Finally, in this experiment we vary our systems across the parameters that result in the work factors above and measure the size of the encrypted message. Again we use the 256-bit AES key, but since no randomness is involved in the size of the encrypted message, we only test each parameter set one time.

Hypothesis

Table 29. Hypothesis for encrypted message size

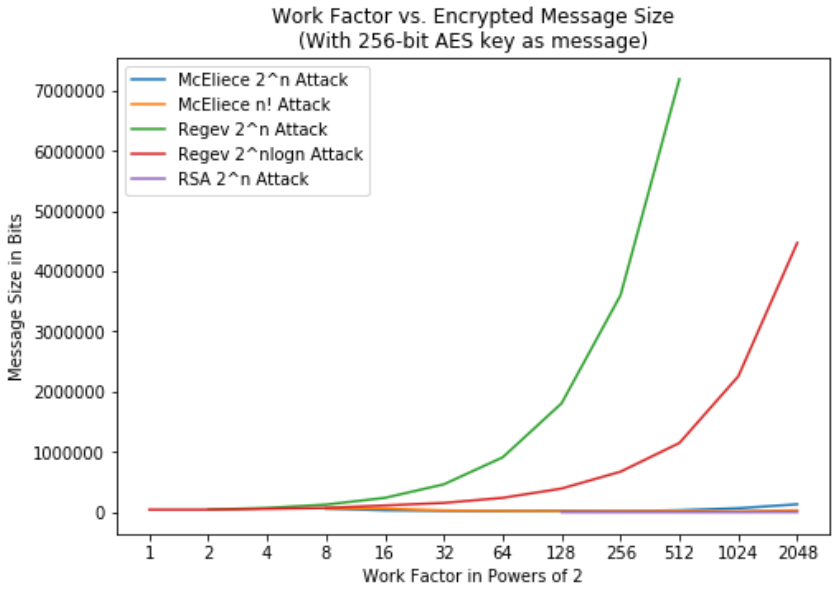
	McEliece	Regev	RSA
encrypted message length	$(n-k)/k\ +1$ bits per bit $=\ O(n)$	$n+1$ bits per bit $=$ $O(n)$	$O(n)$ bits total

From the space complexities calculated we expect Regev message size to grow the fastest followed by McEliece followed by RSA.

Table 30. Encrypted Message Sizes of McEliece, Regev, and RSA

Results 1

	Work factor	McEliece $O(n!)$ Attack	McEliece $O(2^n)$ Attack	Regev $O(2^n)$ Attack	Regev $O(2^{nlogn})$ Attack	RSA Attack $O(2^n)$	# of years to solve	# of years to solve with quantum
# bits		bits	bits	bits	bits	bits		
	2^1				42048			
	2^2			42048	42048			
	2^4			70080	56064			
	2^8	61440	61440	126144	70080			
	2^{16}	61440	30720	238272	112128			
	2^{32}	30720	22528	462528	154176			
	2^{64}	22528	20480	911040	238272			
	2^{128}	20480	24576	1808064	392448	768		
	2^{256}	20480	16384	3602112	672768	512		
	2^{512}	24576	32768	7190208	1149312	512		
	2^{1024}	16384	65536	?	2256576	1024		
	2^{2048}	32768	131072	?	4471104	2048		



Graph 8. Encrypted Message Sizes of of McEliece, Regev, and RSA

Analysis 1

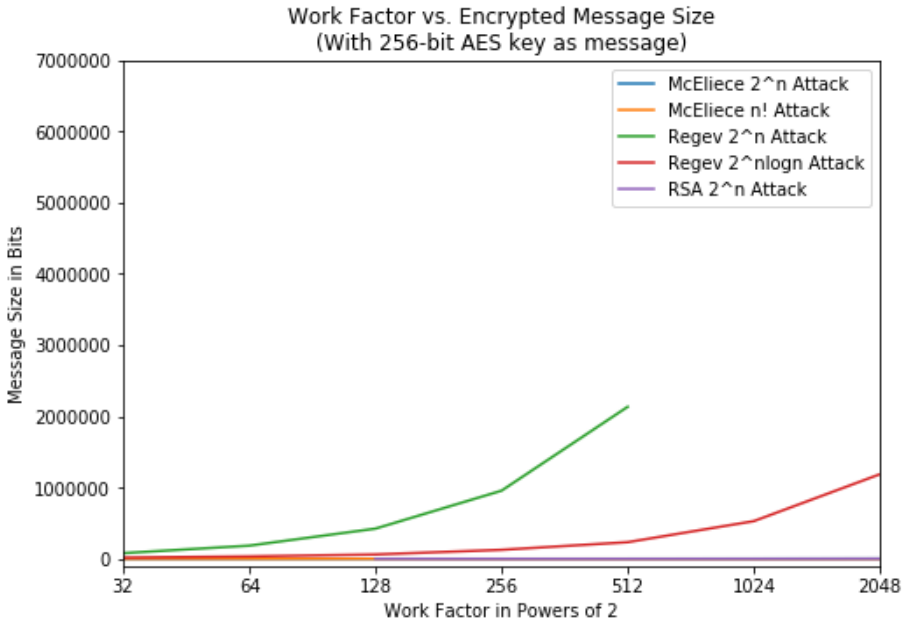
In the results we find what we expected, with Regev encrypted message size growing the fastest followed by McEliece followed by RSA. However, similarly to Experiment 1, we stored each number of the encrypted message in a `u64` even if it was a bit or a small number. So we adjusted the results to include only the bits necessary. The message length in McEliece then becomes n. The message length in Regev is computed as $(n+1) * (\lfloor log_2(p) \rfloor + 1) * 256$.

Results 2

Table 31. Encrypted Message Sizes of McEliece, Regev, and RSA

	Work factor	McEliece $O(n!)$ Attack	McEliece $O(2^n)$ Attack	Regev $O(2^n)$ Attack	Regev $O(2^{nlogn})$ Attack	RSA Attack $O(2^n)$
		# bits	# bits	# bits	# bits	# bits
	2^1				1971	
	2^2			1971	1971	
	2^4			5475	3504	
	2^8	960	960	13797	5475	
	2^{16}	960	480	33507	10512	

	2^{32}	480	352	79497	16863	
	2^{64}	352	320	185055	33507	
	2^{128}	320	384	423765	61320	768
	2^{256}	320	256	956811	126144	512
	2^{512}	384	512	2134593	233454	512
	2^{1024}	256	1024	?	528885	1024
	2^{2048}	512	2048	?	1187637	2048



Graph 9. Encrypted Message Sizes of of McEliece, Regev, and RSA

Analysis 2

While the encrypted message size of Regev is still much larger than in RSA, these results look significantly better for McEliece, with encrypted message size being smaller than in RSA for lower security parameters and the same size at the higher security levels. More specifically, the 2^{256} row of the table looks like the following:

Table 32. Row 2^{256} for encrypted message size

Work factor	McEliece $O(n!)$	McEliece $O(2^n)$	Regev $O(2^n)$	Regev $O(2^{nlogn})$	RSA $O(2^n)$

2^{256}	320	256	56283 * 17=956811	10512*12=126144	512

From this row we can conclude that in terms of encrypted message size, there would be no cost associated with replacing RSA with McEliece.

9.9 Experiments Conclusions

Again assuming that a 2^{256} work factor is sufficient for security today, for the conclusion we combine the results from the 2^{256} work factor from every experiment for the three fastest attacks. The results are shown below, with the least expensive algorithm highlighted in green, the second least expensive algorithm highlighted in yellow, and the most expensive algorithm highlighted in red.

Table 33. Row 2^{256} from all experiments

	Work factor	McEliece $O(2^n)$	Regev $O(2^n)$	RSA $O(2^n)$
public key size	2^{256}	61632	28035317	320
key generation speed	2^{256}	.273176	2.0868353	.031310
encryption speed	2^{256}	.001821	24.071204	.000834
decryption speed	2^{256}	4.169249	.004916	.002033
encrypted message size	2^{256}	256	956811	512

From these results we can see that whether we choose McEliece or Regev, there will be a cost associated with replacing RSA. However, McEliece remains a viable candidate: its public key size, key generation speed, and encryption speed are all comparable to those of RSA. While its decryption speed is costly, we can assume that pre-computing the two matrix inverses would considerably speed up the process, bringing it into a more reasonable range. And McEliece's encrypted AES key is smaller than that of RSA.

By running these experiments we hoped to get an idea of the cost of replacing pre-quantum systems with post-quantum systems. We chose two candidates, the code-based McEliece system and the lattice-based Regev system, to represent post-quantum systems, and we chose RSA to represent pre-quantum systems. Our results are both promising and foreboding; our code-based system seems to be a viable candidate with a negligible cost when used as a replacement, while our lattice-based system seems to be completely unusable, with an encryption time of

twenty-four seconds for a 256-bit AES key. In the greater scheme, we can conclude that while the replacement of pre-quantum systems with post-quantum systems is feasible, more work must be done to gather additional candidate systems.

10. CHALLENGES

We faced many challenges while working on this project. Few examples of the McEliece cryptosystem in its entirety appear on the internet, making it difficult to pin down exactly how to do each step. Additionally, efficient methods for finite field arithmetic were not obvious and were found only later in the project. Implementing them and modifying them for the extension field also required a thorough understanding of finite fields which we did not have at the beginning of the project. It also took a while to find an irreducibility test for polynomials over finite fields, and we experienced confusion while trying to make the method efficient enough for testing many large polynomials. The overall implementation of the McEliece system was so layered and interdependent that debugging the system was extremely difficult. We experienced some difficulties with the Rust programming language. Documentation for different versions of Rust libraries would appear on Google searches, making it difficult to find the most useful version and stick with it. Also, the Rust matrix library, Peroxide, had functionality limited to matrix multiplication and matrix inverse and often crashed during other operations. We had to spend ample time implementing basic matrix methods that Peroxide did not have. In the Regev system, the steps in the decryption algorithm were ambiguous, leading to a delayed understanding of circular distance. Finally, preparing the experiments first required a lot of thought and research in order to create an x-axis on which to compare the systems. And running the experiments required a lot of extra code to be written and parameters to be changed by hand. Even using the library version of RSA took time as we had to choose an appropriate version and subsequently determine runnable parameters, find maximum block sizes, and implement cipher block chaining.

11. Future Work

In this project, we implemented two candidates for post-quantum cryptography and attempted to measure the cost of replacing current systems with those supposedly equipped for quantum attacks. While we feel we made a solid start, much work remains to be done. Our implementations of the McEliece and Regev systems could be analyzed for code inefficiency, something we were not fully focused on here as we were more concerned with correctness and readability. Even small implementation changes might improve their various runtimes. In particular, we could re-run McEliece decryption with the two matrix inverses pre-computed, a step we determined was a major bottleneck, especially for large parameters. Also as we previously discussed, our McEliece system parameters could not be increased to the settings McEliece originally suggested in 1978 due to our trouble generating irreducible polynomials. The ability to reach those parameters seem like a basic component of any implemented McEliece system, and investigation must be done to determine the cause of such problems finding large irreducible polynomials. Perhaps there was some flaw in our irreducibility test or redundancy in our generation of random polynomials. In the Regev system, we used `u64`s to hold all values in the system. Perhaps runtime could be decreased by using the smallest necessary data type. Additional holdups in the Regev system could be searched for and could include our method for choosing a subset of the public key, the time it takes to generate random numbers, or inappropriately chosen data structures. In our experiments, we assumed the fastest attack on an RSA private key was exponential in the length of the key. In reality, this is not the case, as an attack on the private key would likely involve only trying prime numbers, not all numbers, so as to find the factors the modulus. This means that in reality, the parameters of each RSA work factor would have been higher, possibly by a lot, meaning that the apparent efficiency of RSA we found may have been misleading. A significant feature of public-key systems is their ability to be used "in reverse" to provide digital signature capabilities. We did not test these capabilities here, as the McEliece system in its original form cannot be used for digital signatures. A crucial additional experiment would be to implement the Niederrieter scheme, a variant of McEliece that has this capability, and compare it with the original Regev system used in reverse or even an implementation of the Ring-LWE Signature scheme. Furthermore, these post-quantum signature schemes should be compared with RSA as a digital signature or perhaps an implementation of the Diffie-Hellman key exchange. Lastly, in our experiments, we only used the Rust programming languages. Perhaps another programming language would reveal different strengths or further weaknesses in pre-quantum versus post-quantum systems. And finally, there are many more post-quantum systems and approaches to be explored, such as the hash-based Merkle signature scheme or another lattice-based approach such as NTRU [27].

12. Conclusion

Cryptography has come a long way since the markings first inscribed on an Egyptian king's tomb four thousand years ago. Through the sands of time, ciphers have risen and fallen: from the Spartan scytalae to the Polybius Checkerboard, from the Caesar Cipher to the One-Time Pad, from DES to Triple DES to the AES of the modern day, human ingenuity continues to drive the advancement of cryptography. This ingenuity was demonstrated once again in the 1970s when public keys were invented to solve the major weakness of symmetric key systems. However, progress happens in all fields, and the development of quantum computers and the invention of Shor's algorithm have emerged to threaten public keys in the near future. In this project we attempted to address this issue by discussing and developing two candidates for post-quantum public key systems and estimating the cost of replacing pre-quantum systems with those supposedly resistant to quantum attacks. To do this, we implemented the McEliece and Regev public key systems in Rust and ran experiments to compare their performance in public key size, key generation speed, encryption speed, decryption speed, and encrypted message size with those of RSA. Our results showed us that implementing these systems can be complex and that while code-based systems such as McEliece seem to be viable candidates, lattice-based systems such as Regev are currently infeasible for practical use and must be improved. Humans will continue to push the boundaries of technology, and the universal need for information security will remain present; whether quantum computers can crack current systems in one decade, two, or ten, we must match such progress with equally strong cryptography as we step steadily toward the post-quantum future.

REFERENCES

[1] McEliece, Robert J. "A public-key cryptosystem based on algebraic." *Coding Thv* 4244 (1978): 114-116.

[2] Regev, Oded. "On lattices, learning with errors, random linear codes, and cryptography." *Journal of the ACM (JACM)* 56, no. 6 (2009): 1-40.

[3] En.wikipedia.org. 2022. *Khnumhotep II - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Khnumhotep_II> [Accessed 30 May 2022].

[4] Madainproject.com. 2022. *Tomb BH3 - Madain Project (en)*. [online] Available at: <https://madainproject.com/tomb_bh3> [Accessed 30 May 2022].

[5] PuzzleNation.com Blog. 2022. *khnumhotep ii – PuzzleNation.com Blog*. [online] Available at: <https://blog.puzzlenation.com/tag/khnumhotep-ii/> [Accessed 30 May 2022].

[6] 2022. [online] Available at: <https://www.redhat.com/en/blog/brief-history-cryptography> [Accessed 30 May 2022].

[7] Binance Academy. 2022. *History of Cryptography | Binance Academy*. [online] Available at: <https://academy.binance.com/en/articles/history-of-cryptography> [Accessed 30 May 2022].

[8] Encyclopedia Britannica. 2022. *cryptology - History of cryptology*. [online] Available at: <https://www.britannica.com/topic/cryptology/History-of-cryptology> [Accessed 30 May 2022].

[9] Stamp, Mark. *Information security: principles and practice*. John Wiley & Sons, 2011.

[10] MIT News | Massachusetts Institute of Technology. 2022. *The beginning of the end for encryption schemes?*. [online] Available at: <https://news.mit.edu/2016/quantum-computer-end-encryption-schemes-0303> [Accessed 30 May 2022].

[11] Cs.columbia.edu. 2022. [online] Available at: <http://www.cs.columbia.edu/~rjaiswal/quantum-factoring.pdf> [Accessed 30 May 2022].

[12] Medium. 2022. *Is AES-256 Quantum Resistant?*. [online] Available at: <https://medium.com/bootdotdev/is-aes-256-quantum-resistant-4dc3447e7b82> [Accessed 30 May 2022].

[13] En.wikipedia.org. 2022. *Shor's algorithm - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Shor%27s_algorithm> [Accessed 30 May 2022].

[14] Garisto, D., 2022. *Quantum computers won't break encryption just yet*. [online] Protocol. Available at: <https://www.protocol.com/manuals/quantum-computing/quantum-computers-wont-break-encryption-yet> [Accessed 30 May 2022].

[15] Digital Guardian. 2022. *What is NIST Compliance?*. [online] Available at: <https://digitalguardian.com/blog/what-nist-compliance> [Accessed 30 May 2022].

[16] En.wikipedia.org. 2022. *Federal Information Security Management Act of 2002 - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Federal_Information_Security_Management_Act_of_2002> [Accessed 30 May 2022].

[17] Csrc.nist.gov. 2022. *Digital Signatures | CSRC*. [online] Available at: <https://csrc.nist.gov/Projects/digital-signatures> [Accessed 30 May 2022].

[18] En.wikipedia.org. 2022. *Digital Signature Algorithm - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Digital_Signature_Algorithm> [Accessed 30 May 2022].

[19] En.wikipedia.org. 2022. *Schnorr signature - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Schnorr_signature> [Accessed 30 May 2022].

[20] En.wikipedia.org. 2022. *ElGamal signature scheme - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/ElGamal_signature_scheme> [Accessed 30 May 2022].

[21] En.wikipedia.org. 2022. *Elliptic Curve Digital Signature Algorithm - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm> [Accessed 30 May 2022].

[22] Barker, E., Chen, L., Roginsky, A., Vassilev, A. and Davis, R., 2022. *Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography*.

[23] Barker, E., Chen, L., Roginsky, A., Vassilev, A., Davis, R. and Simon, S., 2022. *Recommendation for pair-wise key establishment using integer factorization cryptography*.

[24] En.wikipedia.org. 2022. *MQV - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/MQV> [Accessed 30 May 2022].

[25] Csrc.nist.gov. 2022. *Key Management | CSRC*. [online] Available at: <https://csrc.nist.gov/projects/key-management/key-establishment> [Accessed 30 May 2022].

[26] Cs.cmu.edu. 2022. [online] Available at: <https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/npcomplete.pdf> [Accessed 30 May 2022].

[27] En.wikipedia.org. 2022. *Post-quantum cryptography - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Post-quantum_cryptography> [Accessed 30 May 2022].

[28] En.wikipedia.org. 2022. *Robert McEliece - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Robert_McEliece> [Accessed 30 May 2022].

[29] Classic.mceliece.org. 2022. *Classic McEliece:Intro*. [online] Available at: <https://classic.mceliece.org/> [Accessed 30 May 2022].

[30] En.wikipedia.org. 2022. *McEliece cryptosystem - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/McEliece_cryptosystem> [Accessed 30 May 2022].

[31] Www-math.ucdenver.edu. 2022. *M5410 McEliece Cryptosystem*. [online] Available at: <http://www-math.ucdenver.edu/~wcherowi/courses/m5410/ctcmcel.html> [Accessed 30 May 2022].

[32] En.wikipedia.org. 2022. *Timeline of quantum computing and communication - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Timeline_of_quantum_computing_and_communication> [Accessed 30 May 2022].

[33] En.wikipedia.org. 2022. *Learning with errors - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Learning_with_errors> [Accessed 30 May 2022].

[34] En.wikipedia.org. 2022. *Parity learning - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Parity_learning> [Accessed 30 May 2022].

[35] Cs.virginia.edu. 2022. [online] Available at: <https://www.cs.virginia.edu/~robins/The_Limits_of_Quantum_Computers.pdf> [Accessed 30 May 2022].

[36] En.wikipedia.org. 2022. *Ring learning with errors - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Ring_learning_with_errors> [Accessed 30 May 2022].

[37] En.wikipedia.org. 2022. *NewHope - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/NewHope> [Accessed 30 May 2022].

[38] https://en.wikipedia.org/wiki/CECPQ1

[39] McEliece, Robert J. *The theory of information and coding*. No. 86. Cambridge University Press, 2004.

[40] Encyclopedia Britannica. 2022. *binary symmetric channel | communications*. [online] Available at: <https://www.britannica.com/topic/binary-symmetric-channel> [Accessed 30 May 2022].

[41] Stallings, William. "Cryptography and network security principles and practices 4th edition." (2006).

[42] A. Valentijn, “Goppa Codes and Their Use in the McEliece Cryptosystems,” B.S. thesis), Math Dept., Syracuse Univ., Syracuse, NY, 2015. PDF. Available: https://surface.syr.edu/honors_capstone/845/

[43] Magidin, A., 2022. *Integer matrices with integer inverses*. [online] Mathematics Stack Exchange. Available at: <https://math.stackexchange.com/questions/19528/integer-matrices-with-integer-inverses> [Accessed 30 May 2022.

[44] Hankerson, Darrel, Alfred J. Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[45] S. and Sarwate, D., 2022. *Square root for Galois fields $GF(2^m)$*. [online] Mathematics Stack Exchange. Available at: <https://math.stackexchange.com/questions/943417/square-root-for-galois-fields-gf2m> [Accessed 30 May 2022].

[46] Gao, Shuhong, and Daniel Panario. "Tests and constructions of irreducible polynomials over finite fields." In *Foundations of computational mathematics*, pp. 346-361. Springer, Berlin, Heidelberg, 1997.

[47] Hyperelliptic.org. 2022. [online] Available at: <https://www.hyperelliptic.org/SPEED/slides/Scott_optimal_polynomials.pdf> [Accessed 30 May 2022].

[48] En.wikipedia.org. 2022. *Row echelon form - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Row_echelon_form> [Accessed 30 May 2022].

[49] Math.vanderbilt.edu. 2022. *Proof of the first theorem about determinants*. [online] Available at: <https://math.vanderbilt.edu/sapirmv/msapir/proofdet1.html> [Accessed 30 May 2022].

[50] StackOverflow.com. 2022. *C: Calculating the distance between 2 floats modulo 12*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/6192825/c-calculating-the-distance-between-2-floats-modulo-12> [Accessed 30 May 2022].

[51] Math.drexel.edu. 2022. [online] Available at: <https://www.math.drexel.edu/~tolya/permutations.pdf> [Accessed 30 May 2022].

[52] En.wikipedia.org. 2022. *Binary Goppa code - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Binary_Goppa_code> [Accessed 30 May 2022].

[53] En.wikipedia.org. 2022. *Rust (programming language) - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Rust_(programming_language)> [Accessed 30 May 2022].

[54] Codilime.com. 2022. [online] Available at: <https://codilime.com/blog/why-is-rust-programming-language-so-popular/> [Accessed 30 May 2022].

[55] Berlekamp, Elwyn. "Goppa codes." *IEEE Transactions on Information Theory* 19, no. 5 (1973): 590-592.

[56] Patterson, Nicholas. "The algebraic decoding of Goppa codes." *IEEE Transactions on Information Theory* 21, no. 2 (1975): 203-207.

[57] Singh, Harshdeep. "Code based Cryptography: Classic McEliece." *arXiv preprint arXiv:1907.12754* (2019).