Analysis of McEliece Cryptosystem on Raspberry Pi 3

Gokay Saldamli, Levent Ertaul** and Krishnaraj M. A. Menon***

*San Jose State University, San Jose, CA, USA, **California State University, East Bay, Hayward, CA, USA

gokay.saldamli@sjsu.edu, levent.ertaul@csueastbay.edu, kmavungalayyappamenon@horizon.csueastbay.edu

***Abstract* - As the dawn of quantum computing is approaching, we find ourselves in need of cryptographic algorithms that will withstand these new machines. We know there are methods by which quantum computers will be able to cryptanalyze the most commonly used public key-based key agreement and digital signature algorithms as quantum computers only pose a serious threat to integer factorization and discrete logarithm problems. Meanwhile, there are cryptosystems including McEliece Cryptosystem that rely on general linear code decoding problem remain secure to quantum attacks. In this study, we attempt to find out the performance analysis of McEliece cryptosystem on Raspberry Pi 3; an embedded system where benchmark parameters such as execution time, energy and memory consumption are analyzed. All these metrics are measured for encryption and decryption separately using different public key and plaintext sizes with various parameters of Goppa codes. Moreover, a comparative analysis of these results is outlined, and suitable parameters of Goppa Codes are identified.**

Keywords: Post Quantum Cryptography, IoT Security, McEliece Crypto Algorithm, Raspberry Pi 3, Goppa Codes

1. INTRODUCTION

Cryptography is the study of techniques to realize confidentiality and integrity. Either a small image file stored in a host machine or a huge bank transactions involving millions of dollars could be protected using encryption techniques. Encryption plays the major role in almost all the security services; which uses a key or/and a cipher for modifying the data to be transmitted. Decryption on the other hand does exactly the opposite of encryption to recover a secret message. There are mainly two different types of encryption; namely *symmetric* or *asymmetric* encryption. Based on which method is utilized, in decrypting a message, an exact same encryption key or a key related to encryption key is used.

Sometimes asymmetric encryption using different but related keys is also called as Public key encryption; while encryption and decryption keys are named as public and private keys respectively. The requirement for asymmetric key systems is that public keys that are used for encryption should be shared in a common medium and can be accessed by anyone. However, the private key used for decryption should be kept secret by the receiver at all times. Therefore, the public and private keys form a pair and should be used dependently. In most cases, the security of cryptosystems relies on the toughness in breaking this relation between public and private keys in other words, calculating a private key from a given public key without any other information. For most cryptosystems, this problem is easy to tackle if key sizes are small. However, it could become infeasible if key sizes get bigger. Therefore, algorithms are classified with a logarithmic measure of the fastest known attack (in general, relative to the key length) solving their underlying problem. Moreover, these complexity models are used in finding the appropriate key lengths of a cryptosystem to meet the minimum desired security requirements.

Nevertheless, all complexity and appropriate key length calculations are based on the classical computation model where computations are carried more like the way we do by hand. However, in a quantum computing model, a quantum computer would enjoy the quantum-mechanical phenomena, such as superposition and entanglement and solve the hard problems of classical model in linear time. This brings up the possibility of breaking the current crypto algorithms and our need of cryptographic algorithms that will withstand these new machines. We know there are methods, such as Shor's algorithm [3] and Grover's algorithm [4], by which quantum computers will be able to cryptanalyze the most commonly used public key-based key agreement and digital signature algorithms. Algorithms such as Diffie-Hellman, RSA, DSA and ECDSA, which are ubiquitous in today's communication and networking technology, will be quickly broken if sufficiently large quantum computers are built. However, it currently seems that quantum computers only pose a serious threat to the presumed difficulty of integer factorization and the difficulty of solving the discrete logarithm problem. Meanwhile, there are cryptosystems that rely on "other" problems remain secure to quantum attacks. For instance, McEliece cryptosystem [1] which depends on "general linear code decoding problem" is considered as quantum-resistant (secure even against theoretical quantum level attacks [6]) and studied as a strong candidate for post quantum cryptography (PQC).

In this work, we investigate the efficiency of McEliece cryptosystem in embedded systems particularly on Raspberry Pi 3 [8]. We simply implement the algorithm and evaluate the performance parameters. In the next two sections, we will overview and go through implementation details of McEliece Cryptosystem respectively. In Section 3 we will give the performance analysis of our implementation

and its dependencies on plaintext and cipher text sizes with the variations of Goppa parameters. Section 4 will give a conclusion based on the performance parameters and the its future scope.

2. MCELIECE CRYPTOSYSTEM

It is a public-key cryptosystem goes back to 70s invented by R.J McEliece [1]. It is based on a hard problem in coding theory namely, general linear code decoding problem. The problem is known to have an NP-hard complexity; hence the system is considered as a candidate for PQC.

McEliece describes one of the first probabilistic encryption algorithm. To generate a key pair, one first selects a binary (n, k)-linear code C that can correct up to t errors. One requirement for the code C selection is that there has to be an efficient decoding algorithm so that the code generator matrix G could be calculated. McEliece [1] proposes to use the binary Goppa codes [5] in his original algorithm. These codes belong to the class of algebraic geometric codes constructed by using a genus-0 curve over binary extension fields. They have a simple decoding algorithm that converts a syndrome to an error vector.

After selecting a suited (n, k)-linear code C, one selects a random $k \times k$ binary non-singular matrix S and a random $n \times n$ permutation matrix P. The public key is announced as (F, t) where $F = SGP$ is an $k \times n$ matrix. The public key (F, t) is also called Disguised Generator Matrix composed of The Generator Matrix G, the Scramble Matrix S and a Permutation Matrix P. The triplet (S, G, P) is kept secret as the private key [5] [14].

Encrypting a message m using (F, t) is very simple. After encoding the message m as a binary string of length k, one computes the code vector $c' = mF$. One gets encrypted message by adding a binary random error vector z to the code vector c'. Decryption simply detects and corrects the error at the receiver end. Therefore, the ciphertext c is as follows:

$$c = mF + z = c' + z$$

where F is Disguised Generator Matrix *and* z is an n-bit random binary error vector having weight exactly t.

Decrypting can simply be achieved by using the code's decoding ability. First of all, one has to clear the effect of the random permutation matrix from the ciphertext, hence computes $c'' = cP^{-1}$ where P^{-1} is the inverse of the permutation matrix P. Notice that c'' is now a code vector with some error. Since error has only t components, (n, k)-linear code C can correct these errors. Patterson's Algorithm [9] using the row reduction method in [6] can be used to decode the code vector m'' from c''. Finally, once the effect of S is clear out from m'' by

$$m = m'' S^{-1}$$

one would get back the original message m.

Being mainly simple matrix operations, encryption and decryption in McEliece algorithm can be performed very fast [15]. However, the public and private keys are quite large and this is the main reason why McEliece algorithm never flourished to the same extend as RSA or ECC do. There are recent studies targeting to improve McEliece cryptosystem. Misoczki et. al. [16] introduced *MDPC-McEliece* scheme addressing the long key size problem. Later, Heyse et. al. [17] proposed fast hardware architectures for MDPC-McEliece. Various other more efficient code-based signature schemes [18][19][20] are proposed but still more research is required.

3. MCELIECE PERFORMANCE ON RASPBERRY PI 3

In this section, the performance analysis of the Raspberry Pi 3 for McEliece Cryptosystem has been evaluated. The performance parameters include Execution time, Energy consumption, memory utilization. These are evaluated separately for encryption and decryption with their dependencies with plaintext and ciphertext sizes based on the Goppa parameters: degree of defining polynomial and the power of prime that generates the Galois Filed [5].

Figure 1: Raspberry Pi 3

Raspberry Pi 3 shown in Figure 1, is a mini computer with the given specifications in Table 1. Raspberry Pi 3 has inbuilt Raspbian Operating system and is particularly used as budget desktop, media center, Robots, phones, tablets, laptops.

McEliece encryption and the decryption functions are implemented on Raspberry Pi 3 using a python interpreter software called SAGEMATH [10]. The inbuilt Python modules integrated with SAGE, helps in calculations involving large matrices. The referenced source code [11] for implementing the algorithm were optimized in a way to find out the parameters like energy consumption, memory utilization, CPU utilization and execution time.

Table 1: Raspberry Pi specification

Device Feature	Specification
SoC	Broadcom BCM2837
CPU	4×ARMCortex-A53, 1.2GHz
RAM	1GB LPDDR2 (900 MHz)
Networking	10/100 Ethernet, 2.4GHz 802.11n wireless
Power supply	Micro USB power Input

With the encryption defined as: $\boldsymbol{c = mF + z = c' + z}$

$\boldsymbol{c}$ = Ciphertext
$\boldsymbol{m}$ = Plaintext message
$\boldsymbol{F}$ = Public key
$\boldsymbol{z}$ = The binary random vector

In here, by adjusting the values the plaintext size and public key size, first we compute the implementation parameters for the memory consumption, CPU utilization, execution time and power consumption of the Raspberry Pi 3. The major constrain that determines the performance of McEliece cryptosystem, $\boldsymbol{k >= n - qt}$ should be satisfied for successful encryption and decryption, where

$\boldsymbol{k}$ = Length of message
$\boldsymbol{n}$ = Length of Ciphertext
$\boldsymbol{q}$ = Power of prime
$\boldsymbol{t}$ = Degree of polynomial

Solving the following equations gives the dependencies of the performance parameters with the plaintext and public key sizes.

$$\boldsymbol{k_1 = k_2}.$$
$$\boldsymbol{n_1 - p_1 t = n_2 - p_2 t_2}$$

This also gives us the instances of Goppa parameters (q, t) which helps in finding the dependencies of performance parameters with plaintext and ciphertext sizes. By executing the script in Figure 2, we could figure out that, at certain values of (q, t) where the plaintext would be a constant with varying public key sizes, and similarly the public key would be a constant with varying plaintext sizes which directly shows the dependency.

Table 2 shows the variations of degree polynomials, power of prime with the size of plaintext bits by keeping public key size at a constant at 512Bytes. Observe the interesting fact that size of ciphertext gets smaller if plaintext sizes are increase increased once public key is fixed to constant 512bytes.

Table 2: Goppa parameters for fixed public key size

Prime power (q)	Polynomial degree (t)	Plaintext size (k-bits)	Ciphertext Size (n-bits)	Pub key size (n*k) Bytes
10	102	4	1024	512
9	56	8	512	512
8	30	16	256	512

Whereas Table 3 shows the variations of degree polynomials(t), power of prime (q) with the size of public key bytes (nk), by keeping plaintext size (k) at a constant of 16 Bits. Here one could analyze that for a constant plaintext size of 16bits, we could see different variations of public key. This can be considered as an encryption of 16bits blocks of plaintext message forms a 256bits ciphertext using a 512 Bytes key.

Table 3: Goppa parameters for fixed pub key size

Prime power (q)	Polynomial degree (t)	Plaintext size (k-bits)	Ciphertext Size (n-bits)	Pub key size (n*k) Bytes
6	8	16	1024	128
7	16	16	512	256
8	30	16	256	512

Table 2 and Table 3 show the basis of calculating the performance parameters in our analysis.

3.1. EXECUTION TIME

The execution time is one of the major performance parameter that determines the whole performance of the McEliece algorithm. Here the execution time taken for encryption and decryption has been evaluated by varying the plaintext sizes (k) and random error vector (z) by keeping the public key (nk) a constant of 512 Bytes and the graph being plotted in Figure 3. From the plot one could see that the execution time gets better with the increment of plaintext size for a constant public key size. Even though it looks contradictory with the classic concept that time should get increase with the size of message bits, the constrain that discussed above $\boldsymbol{(k >= n - qt)}$ forces the time to get decremented according to plaintext size. By analyzing the Table 2 and Table 3, we could verify the reason behind the longer execution time for encryption and decryption. The Goppa parameter, degree of the polynomial (t) is 102 even for a small plaintext size of 4 bits which leads to a high execution time. As the plaintext size gets incremented for a constant key 512 Bytes, the degree of polynomial (t) gets smaller which again underlines the fact for a better execution time.

```
def calc_k(q, t)
  return 2**q - q*t

def valid_t(q, k):
  if (pow(2, q) - k) % q == 0:
    return True
  else:
    return False

def calc_t(q, k):
  return (pow(2, q) - k) // q

def eq2(q1, q2, t1, t2):
  if pow(2, q1) - pow(2, q2) == q1*t1 - q2*t2:
    return True
  else:
    return False

k_val_dict = dict()

for q in range(5,11):
  for k in range(4, 1025):
    if valid_t(q, k):
      #print ("q: %d, k: %d" % ( q, k))
      t=  calc_t(q, k)
      if t > 0:
        if k not in k_val_dict:
          k_val_dict[k] = [(q, t)]
        else:
          k_val_dict[k].append((q, t))

for k in k_val_dict:
  if len(k_val_dict[k]) > 1:
    print (k,":", k_val_dict[k]
```

Figure 2: Script for finding the parameters

Figure 4 shows variation of execution taken for encryption and decryption separately in accordance with the variation of public key in bytes for chosen binary random error vector (e) sizes by keeping plaintext size a constant at 16Bits.

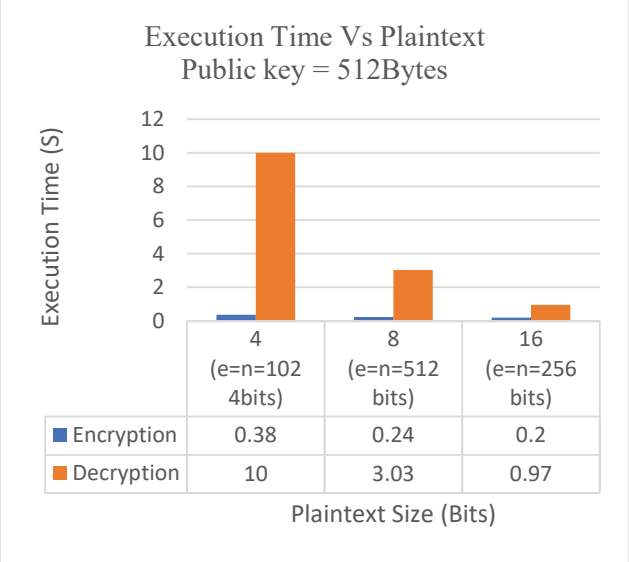

Figure 3: Execution Time vs Plaintext

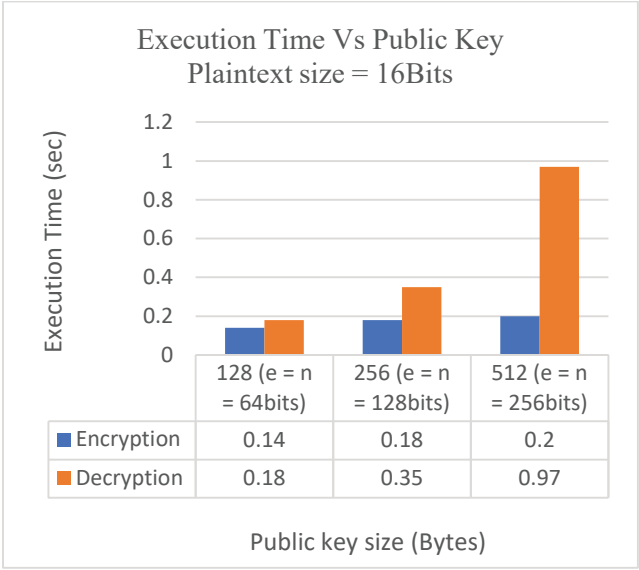

Figure 4: Execution Time vs Public key size (Plaintext = 16bits)

McEliece execution time gets better if plaintext size is increased for a fixed public key size. Figure 3 itself shows that one could have a faster encryption and decryption for 16bits plaintext when compared with 4bits plaintext with a fixed 512 Byte public key. This reveals a new way for encrypting large files since fixed block size with a fixed public key yields a faster encryption and decryption. More options of the McEliece parameters (n, k, q, t) can be obtained from Niebuhr et. al. [13].

3.2. ENERGY CONSUMPTION

One of the major criteria that determines the performance evaluation of McEliece cryptosystem on a device like Raspberry Pi is energy consumption. The energy consumption can be calculated using the relation

$$\boldsymbol{E = I * N * \tau * Vcc} \qquad [12]$$

where $\boldsymbol{E}$ = Energy in joule
$\boldsymbol{Vcc}$ = Supply Voltage
$\boldsymbol{I}$ = Current in Ampere
$\boldsymbol{N}$ = Number of Clock cycles
$\boldsymbol{\tau}$ = Clock period
$\boldsymbol{N * \tau}$ = Execution time

Note that Raspberry Pi 3 needs $\boldsymbol{Vcc}$ = 5.1 volts and $\boldsymbol{I}$ = 0.7 A. The overall energy consumption for encryption and decryption has been evaluated in accordance with the variations of plaintext sizes for a constant public key size and by varying public key sizes for a constant plaintext size.

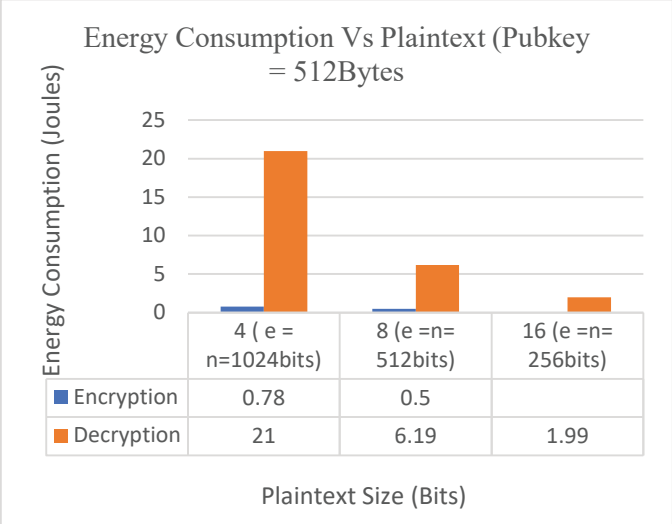

Figure 5: Energy Consumption vs Plaintext size (Pub key = 512Bytes)

Figure 5 shows the evaluation of energy consumption with this new hypothesis. Like in execution time, we could see a decrease in energy consumption for a fixed public key of 512Bytes while plaintext size is increased. Similarly, Figure 6 shows the energy consumed by a Raspberry Pi while encrypting and decrypting a message with the variation in public key size and keeping the size of plaintext a constant value of 16 Bits.

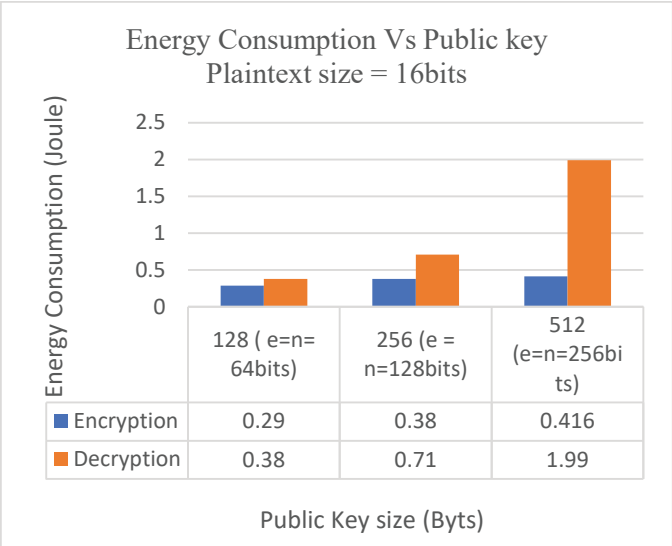

Figure 6: Energy Consumption vs Public key size

Notice that Figure 6 shows how the energy consumption keeps on increasing in accordance with the public key size for constant plaintext size of 16bits. This is similar to what is shown in Figure 3 and Figure 4; the same Goppa parameters degree of the polynomial (t) and power of prime (q) determines the energy consumption parameter. This leads to high energy consumption for 4 bits plaintext size with 512 Bytes key and conversely a less energy consumption for a 16bits plaintext size for 128Byte key.

3.3. MEMORY UTILIZATION

The memory utilization of the device while on encryption and decryption can be calculated using the below given equations respectively.

$$c = mF + z \quad \text{and} \quad c\,P^{-1} = [mF + z]\,P^{-1}$$

Where m denotes the plaintext message matrix with dimension $1 \times k$, F denotes the Disguised matrix with a dimension $n \times k$ and z denotes the random error vector with dimension $1 \times n$.

Recall that $F = G\,S\,P$ where G, the generator matrix, S the scramble matrix and P the permutation matrix. The generator matrix [G] is derived from the parity check matrix [H] composed of three matrices [X], [Y] and [Z]. Theoretically, we could calculate the total number of bits involved in encryption and decryption. Therefore, the memory utilized would be the addition of the dimensions of the matrices involved.

Matrix	Dimension
X	$t \times t$.
Y	$t \times n$.
Y	$n \times n$.
G	$k \times n$.
S	$k \times k$.
P	$n \times n$.
e	$1 \times n$.

Therefore, the memory utilized in bits for encryption is

$$t^2 + nt + n^2 + nk + k^2 + n^2 + n = n(1+k+t+2n) + t^2 \qquad (1)$$

In the case of decryption, the bits involved would be the sum of dimensions of above-mentioned matrices in addition with dimension of inverses of the permutation matrix P, Generator matrix G and scramble matrix S are being calculated. Therefore, number of bits used for decryption are,

$$n(1+k+t+2n) + t^2 + n*n + nk + k^2 = n(3n+2k+t+1) + t^2 + k^2. \qquad (2)$$

Equations (1) and (2) under the constrain [$k >= n - m * t$], can be used to find the dependency of memory consumption for encryption and decryption separately, with the plaintext and public key size. The graph shown in Figure 7 provides the various instances obtained regrading memory utilization for encryption and decryption separately, by varying plaintext size and by keeping the public key constant at 512 Bytes.

To give an example: let the memory utilized for Encryption and Decryption based on Goppa parameters be,

q (Power of prime) = 10
t (Degree of Polynomial) = 102
plaintext size (k) = 4bits
Ciphertext size (n) = 1024bits
Pub key size = 512 Bytes

For encryption, substitute the values in Equation (1) gives

$$\text{Memory cost} = n(1+k+t+2n) + t^2 = 277.1405\text{Kbytes}$$

while for decryption, substitute the values in Equation (2)

$$\text{Memory cost} = n(3n+2k+t+1) + t^2 + k^2 = 408.7265\text{KBytes}$$

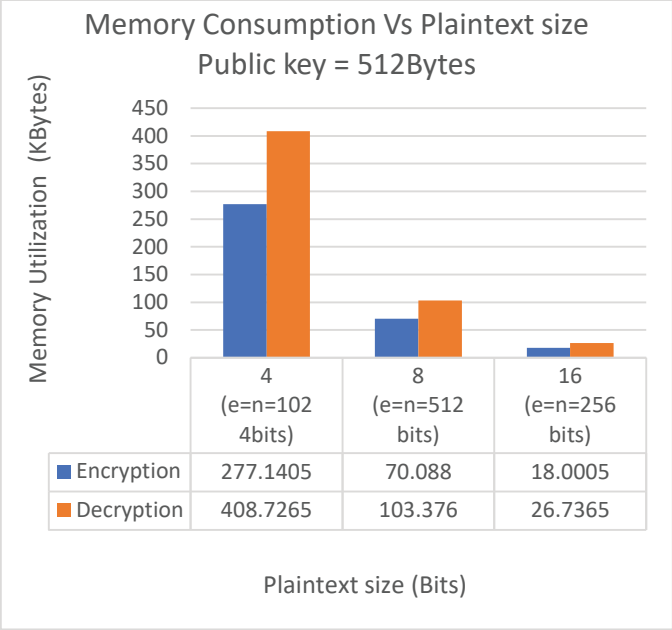

Figure 7: Memory Consumption vs Plaintext sizes

From Figure 7, we could easily figure out that the memory utilization gets decreased as the plaintext size increases. Even for the low value of plaintext size (k), the degree of the Goppa polynomial (t) is quite high for maintaining a constant public key size, which in leads in the high memory utilization. The graph shown in Figure 8, provides the various instances obtained regrading memory utilization on encryption and decryption by varying plaintext size with keeping the public key constant at 512 Bytes

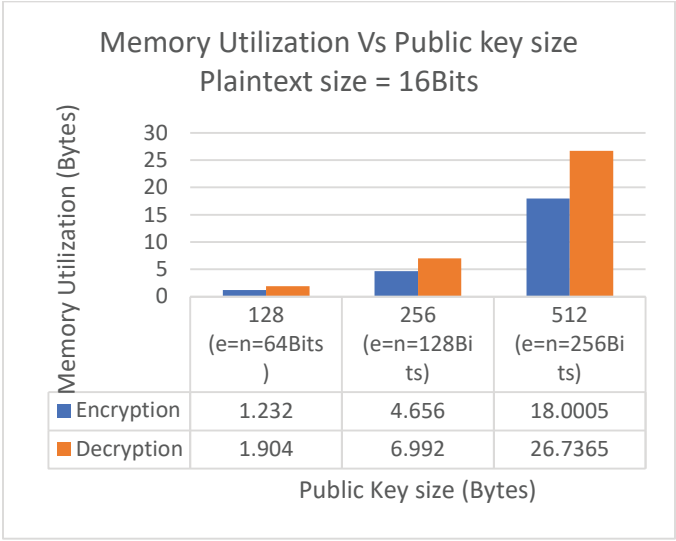

Figure 8: Memory Consumption vs Public key sizes

In Figure 8, one could see that the memory utilization is directly proportional with the size of the public key size. By running McEliece algorithm in Raspberry Pi 3, we verify that the memory consumed by the device matches with what is computed theoretically.

3.4. CPU UTILIZATION

One major parameter which determines the performance of Raspberry Pi is CPU utilization. It has been evaluated for encryption and decryption together. As we did with other performance analysis, CPU utilization also been done by varying the plaintext sizes with constant public key size and by varying the public key for constant plaintext size.

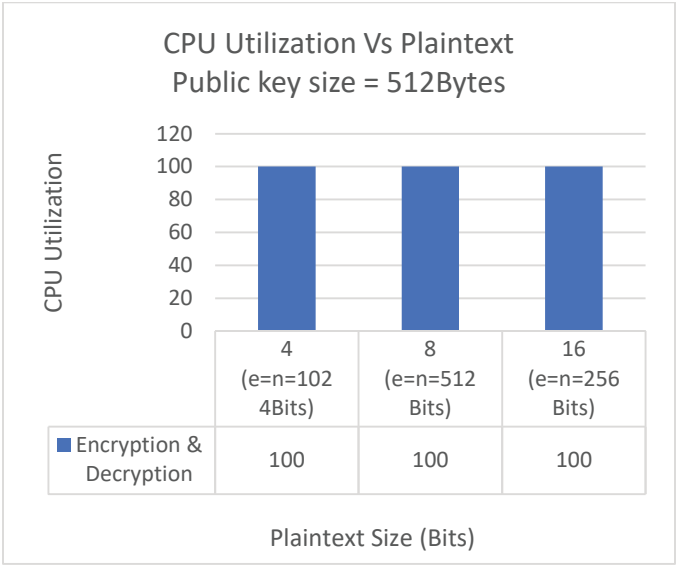

Figure 9: CPU Utilization vs Plaintext

In Figure 9, the CPU utilization being analyzed for encryption and for decryption together. We are getting a utilization even for various values of plaintext sizes with constant public key size of 512bytes. As the parameters degree of Goppa polynomial (*t*), and prime power (*q*) determines the utilization by maintaining a stable value of public key size(512Bytes), which enforces the utilization to be stable. In other words, even for a low range value for degree of polynomial (t) = 30 and for a high range value of t = 100, the CPU is being utilized at its maximum range of 100%. Graph shown in Figure 10, evaluates the variation of CPU utilization for encryption and decryption together, with variation of public key size and by keeping the plaintext size a constant equal to 16Bits. Here the case is bit different as that of the previous one, were the CPU utilization varies a bit and it ranges from 82% to 100%

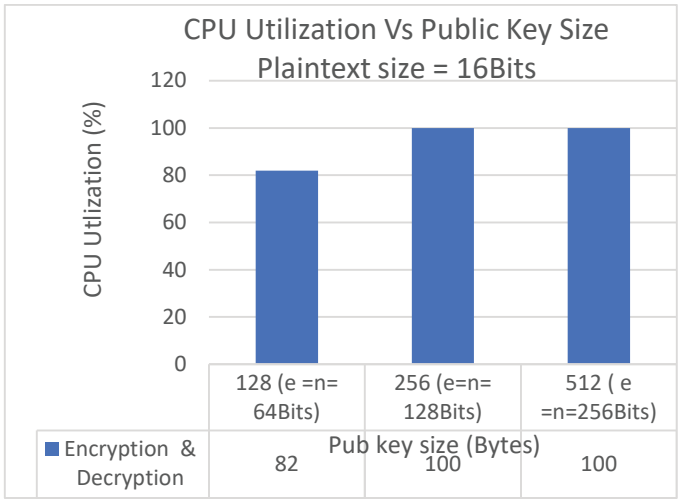

Figure 10: CPU Utilization vs public key size

4. CONCLUSION

In this study, we evaluated the performance analysis of McEliece Cryptosystem in Raspberry Pi 3. The performance parameters include execution time, energy consumption, CPU and memory utilization. The parameters are evaluated for and encryption and decryption separately and with their corresponding dependencies with plaintext and public key sizes. The Goppa parameters like degree of the polynomial (*t*), power of prime (*q*), set the major constrain in the variations of the performance of this algorithm. These performance parameters get decremented according to the increase in plaintext size for a constant public key, at the same time these parameters get incremented with increase in public key size for a constant plaintext size. The results obtained regarding the variations of performance parameters with plaintext and public key sizes were really exciting. On this basis, we can say that it could be a good candidate for big block size message encryptions for a fixed public key. But the major thing that stand against the proper functioning of this algorithm on a device like Raspberry Pi was its CPU power, which took very high execution time even for low ranges of plaintext messages. Taking these facts into account we could conclude that it can open a new platform for more secure way of data exchange in the era of quantum computing.

REFERENCES

[1]. R. J. McEliece. "A public-key crypeosystem based on algebraic coding theory". The Deep Space Network Progress Report, vol. 44, pp. 114-116, 1978.

[2]. "What Is Quantum Computing?" - IBM Q - US, www.research.ibm.com/ibm-q/learn/what-is-quantum-computing/.

[3]. P.W. Shor, Algorithms for quantum computation: discrete logarithms and factoring, AT&T Bell Labs., Murray Hill, NJ, USA. ISBN: 0-8186-6580-7

[4]. Dieter Fishbein, "Machine-Level Software Optimization of Cryptographic Protocols". MSc. thesis, University of Waterloo, 2014

[5]. Valentijn, Ashley, "Goppa Codes and Their Use in the McEliece Cryptosystems" (2015). Syracuse University Honors Program Capstone Projects. 845. Online: https://surface.syr.edu/honors_capstone/845

[6]. Hang Dinh, Cristopher Moore, Alexander Russell. "The McEliece Cryptosystem Resists Quantum Fourier Sampling Attacks". 2010. Online: https://arxiv.org/pdf/1008.2390.pdf

[7]. Daniel J. Bernstein. "Introduction to post-quantum cryptography". Springer-Verlag, 2009

[8]. Raspberry Pi, Online: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[9]. N. J. Patterson. "The algebraic decoding of Goppa codes". IEEE Transactions on Information Theory, vol. 21(2). pp. 203 - 207, 1975.

[10]. "Sage." *SageMath Mathematical Software System*, www.sagemath.org/.

[11]. Christopher Roering, "Coding theory-based cryptopraphy: McEliece cryptosystems in Sage". Honors theses, Saint John's University, (2013).

[12]. Levent Ertaul and Arnold Woodall. "IoT Security: Performance Evaluation of Grain, MICKEY, and Trivium - Lightweight Stream Ciphers". Proceedings of the 2017 International Conference on Security & Management (SAM'17), 2017, pp. 36–38

[13]. Robert Niebuhr, Mohammed Meziani, Stanislav Bulygin, and Johannes Buchmann, "Selecting Parameters for Secure McEliece-based Cryptosystems". International Journal of Information Security, Springer-Verlag, vol. 11, Issue 3, pp 137–147

[14]. Johannes A. Buchmann, Denis Butin, Florian Göpfert, and Albrecht Petzoldt. "Post-quantum cryptography: state of the art". In LNCS Essays on The New Codebreakers, Springer-Verlag, vol. 9100, pp. 88-108, 2015.

[15]. B. Biswas and N. Sendrier. "McEliece cryptosystem implementation: Theory and practice". In PQCrypto, volume 5299 of Lecture Notes in Computer Science, pages 47–62, 2008.

[16]. R. Misoczki, J. P. Tillich, N. Sendrier, and P. S. L. M. Barreto. "MDPC-McEliece: New McEliece variants from moderate density parity-check codes". In Proceedings of ISIT, pages 2069–2073. IEEE, 2013.

[17]. S. Heyse, I. vonMaurich, and T. Güneysu. "Smaller keys for code-based cryptography: QC-MDPC McEliece implementations on embedded devices". In CHES, volume 8086 of LNCS, pages 273–292. Springer, 2013.

[18]. E. Persichetti. "Secure and anonymous hybrid encryption from coding theory". In PQCrypto, volume 7932 of Lecture Notes in Computer Science, pages 174–187. Springer, 2013.

[19]. C. A. Melchor, P. Cayrel, P. Gaborit, and F. Laguillaumie. "A new efficient threshold ring signature scheme based on coding theory". IEEE Transactions on Information Theory, 57(7):4833–4842, 2011.

[20]. G. Landais and N. Sendrier. Implementing CFS. In INDOCRYPT, volume 7668 of Lecture Notes in Computer Science, pages 474–488. Springer, 2012.