IoT Security Risk Analysis

IoT Security Risk Analysis

A THESIS SUBMITTED TO BRAC UNIVERSITY IN

PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

OF BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND

ENGINEERING

Supervised by

Dr. Amitabha Chakrabarty

Submitted by

Nazmus Sakib, Student ID: 13101241

Ismot Jerin, Student ID: 13301054

Nuzhat Khan, Student ID: 13301070

Shaela Quader, Student ID: 13101178

Department of Computer Science and Engineering
BRAC University
Dhaka, Bangladesh

TABLE OF CONTENTS

List of Tables

List of Figures

DECLARATION

We hereby solemnly declare that the research work embodying the results reported in this thesis entitled "IoT Security Risk Analysis" submitted by Nazmus Sakib, Ismot Jerin, Nuzhat Khan and Shaela Quader has been carried out under supervision of Dr. Amitabha Chakrabarty, Assistant professor of Computer Science and Engineering department at BRAC University, Dhaka. It is further declared that the research work presented here is original and has not been submitted to any other institution for any degree or diploma.

Certified by:

Dr. Amitabha Chakrabarty

Assistant Professor,

Department of Computer Science and Engineering,

BRAC University, Dhaka

Signature of Authors:

____

Nazmus Sakib

____

Ismot Jerin

____

Nuzhat Khan

____

Shaela Quader

ABSTRACT

Internet of things (IoT) has become a buzzword in today’s world to describe billions of devices, interconnected via the web. It includes a diverse range of devices, starting from wearable ultra low-powered gadgets like fitness bands to medical instruments and to home appliances to automobiles. There may be so many devices harnessing the power of IoT, however, security is still an issue for these devices as these are constrained with limited power supply, processing cycles and memory usage. The IoT sector is not impeccable, security is still a threat for the IoT devices as these devices are meant for low power usage for small scale setups. Security algorithms are not abrupt, but many of them don’t fit the IoT systems as their compatibility can only rely on products with larger form factor (Which usually means better performance, storage etc.). In such context, working with security of IoT devices has become an interesting area in computer science. Though, researchers and security professionals have developed advanced algorithms for ensuring digital security, but many of them are not suitable for the IoT world because of the restrictions we have. In our work, we tried to contribute to the matter of security in IoT devices. In this work, the main concern has been to investigate the performance of different security algorithms and compare them in terms of processing cycle and execution time in Raspberry Pi. We have worked with FLECC_IN_C and Crypto++, two different libraries with number of algorithms where we can find ecdh, ecdsa, ciphers, message authentication codes, one-way hash functions, public-key cryptosystems, key agreement schemes, and deflate compression. and measured their performance in a constrained environment. It is the first of its kind, to this work’s knowledge, to use raspberry pi which is established as black box device and implemented security algorithms on it. We implemented these libraries in different IoT platforms, showing comparisons of how these algorithms may affect a system in terms of resource utilization. The work in the end shows a summarised view of several key algorithms and decides which is better in the terms of IoT constraints.

ACKNOWLEDGEMENT

Firstly, our earnest appreciation Almighty Allah, because of whom this research works is successful and who enabled us to pursue our degree in computer science and engineering.

Secondly, we would like to express our heartiest and sincere gratitude to our supervisor Dr. Amitabha Chakrabarty, Assistant professor of Computer Science and Engineering department at BRAC University, for his patience, motivation, enthusiasm and wise collaboration. His guidance helped us in every way possible of our research without which it was impossible to accomplish this research work.

We are thankful to our companions, Erfan and Hirok for providing testing equipments and helpful cooperation for which we can successfully complete this work.

Last but not the least, We would like to thank our family members for their blessing, inspiration, encouragement and moral support during this time period. Their patience motivation and inspiration has helped us a lot to achieve our goal.

CHAPTER 01

INTRODUCTION

This chapter talks about the objectives and motivation of the project and then explains the outline for the rest of the work

1.1 OBJECTIVES

The objectives of this work are:

- Discuss performance of security algorithms in Iot
- Discuss about similar work
- Show performance in multiple devices
- Testing on device
- First use of library

1.2 MOTIVATION

In today's world, with the increasing number of interconnected devices, protecting the privacy has become a challenging task. It can automatically be expected that billions of humans will be connected through trillions of devices. Now this is really alarming especially in the field of Internet of Things as this field is not yet fully developed in the security field although it has started to gain wide popularity. The Internet of Things (IoT) is one of the most hyped topics in this era. It has binded almost every electrical devices in a chain and has brought everything together with an improvised version. It is a probabilistic estimation that, if the use of IoT increases at the current rate, the user number will increase greatly in the coming years. In this current age is of Internet of Things (IoT), digitally connected devices are involved in every aspect of our lives, including our homes, offices, cars and even our bodies. With the help of IPv6 and the wide deployment of Wi-Fi networks, IoT is growing at a dangerously fast pace.

In near future, it is expected that billions of humans will be connected through trillions of smaller devices which might be regulated without direct human interaction and only using the IOT (Internet of Things). The main challenges with the number of increasing devices are i. The infrastructure and ii. The security issues.

This is really alarming in the field of Internet of Things (IoT) as the ecosystem is yet to be developed, especially in the security field. The fundamental security challenge of IoT is that it increases the number of devices behind a network firewall. This work concentrates on current cryptographic security algorithms, implementing them on a low powered machine and analyzes the overall impact it creates on the device. It also shows the use of cryptographic algorithms and algorithms which use anonymous key agreement protocols that allows two parties, each having an elliptic curve public – private key pair, to establish a shared secret over an insecure channel. It covers widely used cryptographic routines and the use of these under various conditions to test their performance and to get an overall insight of how they can map the future of IoT security. The implementation methodologies were not easy due to Raspberry Pi not originally being supported by the libraries itself. The build files had to be customized to suit the needs of the low powered device and achieve our goal.

There are not enough algorithms to suit the needs of security in this vast sector of interconnected devices. There are some work going on to get the best algorithms and use them for IoT sector but the main problem is the overwhelming complexity of security architecture that needs to be implemented for interconnected devices. Our work focuses on some of the key findings of security algorithms to this date and explore them in terms of IoT constrains and put them to test in the black box device mentioned before. The task has its own complexity as mostly all the algorithms are not supported by the raspberry pi and significant tweaking had to be done to get the optimal result. This was quite troublesome but from the moment our research took on the thought of helping the next world of interconnecting wearable devices struck in our mind and it was the spark of growing some 20 billion IoT device market was our primary motivation to keep on going with the research.

1.3 THESIS OUTLINE

The rest of the thesis is organized as follows:

- ➢ Chapter 02 shows the previous works and reviews based on implementation of algorithms on constrained devices
- ➢ Chapter 03 presents the various criteria based on which the tests took place. It also covers the parameters used and test model development.
- ➢ Chapter 04 demonstrates the experimental setup, results and accuracy analysis.
- ➢ Chapter 05 discusses in details about the results and its analysis
- ➢ Chapter 06 concludes the thesis and states the future research directions.

CHAPTER 02

LITERATURE REVIEW

This chapter discusses in short some previous work, discusses about the concepts used in thi work and related work

2.1 PREVIOUS RELATED WORK

[1]In the early 2000s some companies and world leaders in IoT and data science started working on the idea of IoT systems for general people. But it was 1999 when the term internet of things came into being by Kevin Ashton executive director of the Auto-ID Center. Auto-ID opens in 1999 which then helped develop Electronic Product Code (EPC), this is an identification system based on RFID. In 2000 LG was the one who started the idea of internet refrigerator, one true IoT device and also one of the first IoT devices at work. In 2002 Ambient Orb, a data-centric IoT platform was released which gathered personalised data and changed its color based on the dynamic parameter. In 2003-2004 the term IoT started to become public and has mentioned several times by The Guardian, Scientific American and the Boston Globe. In 2005 Nabaztag introduced the IoT enabled robot (Rabbit). It can notify users about stock markets, news headlines etc. In 2008 EU recognized IoT and the first conference held. In the period of 2008-2009 the Internet of Things was born. A whole range of IoT platforms (Pachube, Thingspeak), standards (6LoWPAN, Dash7, etc) hardware and software (Contiki, TinyOS, etc) have developed in the time of 199x to 200x. In 2011 ipv6 was launched and started growing.

Form the early 1990s security experts has been warning us about the potential risk involving Interconnected devices in a global scale. If an outbreak of massive data leak occurs there will be no way to mitigate the problem and devices could be unsecured throughout the entire infectious process. A researcher at Proofpoint which happens to be an security firm for enterprise solution, in December of 2013, discovered the first botnet specially made for IoT enabled devices. According to Proofpoint, more than 25 percent of the botnet was made up of devices other than pcs, which includes smart TVs, baby monitors etc, pretty much all of the smart devices can be used to compromise security. [2]A report made public in 2014 estimated financial losses as a result of cybercrime and cyber espionage at $445 billion to the world economy. That is eerily huge and uncomfortable for most of the giants who wants to invest in the IoT world.

1Information details can be found at: https://goo.gl/RMNRLq

2Source: https://goo.gl/PQ74Fb

2.2 LITERATURE REVIEW

From past few decades, working with and working for Internet of things (IoT) has become an interesting area in computer science. Though the research on this topic is still in progress and security and privacy are some enormous concerns along with the infrastructure issue. In IoT world the drawback is that the processing cycles are smaller and power supplies are lower. In such cases,it is known that for faster computation elliptic curves are used. There are numerous books and research articles describing how elliptic curves are effective and use smaller keys to encrypt and decrypt data[12],they are useful in wireless environment compared to other algorithms[9] but there exists very little work on IoT security enhancement using elliptic curves. For example, work has been done with elliptic curves to implement security for IoT devices.[7] [15]Again, there are works with cipher algorithms where they introduce an open framework of lightweight block ciphers on a multitude of embedded platforms.[4]

Now, people are going for hybrid methodology to solve the low computational issues. In some works, they have addressed some lightweight ciphers, compared them and came up with a new algorithm.[14] Again there are some who proposes a hybrid semantic service matchmaking method which merges their previous work on probabilistic service matchmaking using latent semantic analysis with a weighted - link analysis based on logical signature matching.[2] However, to the best of this work's knowledge based on literature review done, there are very few articles which propose a new kind of algorithm to balance these kind of problems with IoT devices. For example, some proposes an encryption method based on XOR manipulation, instead of complex encryption.[10] They have proposed it for RFID system which requires particular hardware design.

There are quite a few reliable articles telling which methodologies should be embraced to secure the IoT system yet lots of people are still working on it as the field is not yet completely stable in infrastructure or complete secure.[17] [18] [8] A in depth study of current security methods is needed to develop this field.

2.3 ALGORITHMS

A variety of security algorithms have analyzed here to make an efficient comparison among those. There are two main concerns about security algorithms are, how secure the data is and how fast it can be encrypted and decrypted. In IoT device when dealing with low power and computing cycle ensuring these two is really a challenge. So to guarantee security and get maximum computing speed choosing algorithms makes the work easier.

Authenticated Encryption Schemes: In authenticated encryption schemes the modes are:

- **GCM (Galios/Counter Mode):** This mode uses universal hash function to operate encryption and decryptions. The mode is defined in NIST's SP 800-38D, and P1619. GCM is a high performance mode which offers both pipelining and parallelization. The mode accepts initialization vectors of arbitrary length, which simplifies the requirement that all IVs should be distinct. GCM uses a block cipher with a block size of 128 bits and produces tags of 128, 120, 112, 104, or 96 bits (64 and 32 bits are available but not recommended under most circumstances).
- **CCM (Counter with CBC-MAC):** This is a mode of operation for cryptographic block ciphers. The mode is defined in NIST's SP 800-38C (2004), P1363, and RFC 3610. CCM provides both confidentiality and authentication. The underlying block cipher must have 128-bit blocks and is operated in CTR mode to generate a stream. The formatting function of CCM ensures there is sufficient interleaving of encryption and authentication components before XORing the plaintext and key stream.
- **EAX:** This mode of operation is an AEAD mode of operation. It provides both confidentiality and authenticity, and authentication assurances on additional data. EAX is the work of Bellare, Rogaway, and Wagner. The mode was presented in ‘A Conventional Authenticated-Encryption Mode’. EAX was a candidate during NIST's Authenticated Encryption Mode selection process. EAX mode is an n-bit mode of operation. This allow the mode to operate on AES with 128 bits or SHACAL-2 with its 256 bit block size. EAX is online, meaning the data does not need to be known in advance - it can be streamed into the object (there are some practical implementation constraints). Unlike both CCM and GCM, EAX handles messages of arbitrary length.
- **OCB:** This mode was designed to provide both message authentication and privacy. It is essentially a scheme for integrating a Message Authentication Code (MAC) into the

operation of a block cipher. In this way, OCB mode avoids the need to use two systems, a MAC for authentication and encryption for privacy. This results in lower computational cost compared to using separate encryption and authentication functions.

High Speed Stream Ciphers: A stream cipher is a symmetric key cipher where plaintext digits are combined with a pseudorandom cipher digit stream (keystream). In a stream cipher, each plaintext digit is encrypted one at a time with the corresponding digit of the keystream, to give a digit of the ciphertext stream. Since encryption of each digit is dependent on the current state of the cipher, it is also known as state cipher. In practice, a digit is typically a bit and the combining operation an exclusive-or (XOR). Stream cipher makes use of a much smaller and more convenient key such as 128 bits. Based on this key, it generates a pseudorandom keystream which can be combined with the plaintext digits in a similar fashion to the one-time pad. However, this comes at a cost. The keystream is now pseudorandom and so is not truly random. The proof of security associated with the one-time pad no longer holds. It is quite possible for a stream cipher to be completely insecure.

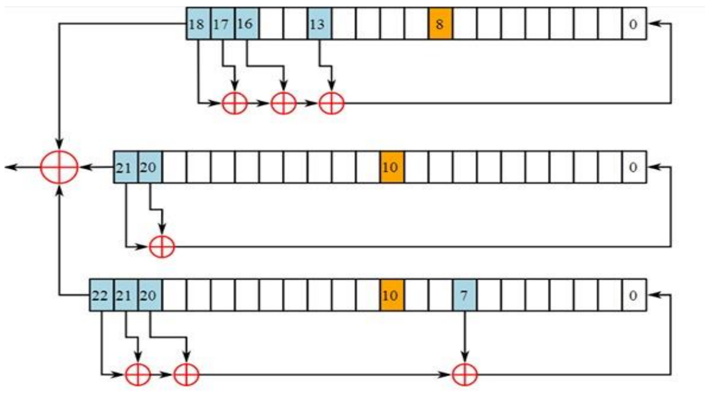


Fig 1: Operation of the keystream generator

In the figure , The operation of the keystream generator in A5/1, an LFSR-based stream cipher used to encrypt mobile phone conversations. In High Speed Stream ciphers ChaCha (ChaCha8/12/20), Panama, Sosemanuk, Salsa20, XSalsa20 have been used in this thesis.

AES and AES candidates: This is a form of encryption which simultaneously provides confidentiality, integrity, and authenticity assurances on the data. These attributes are provided under a single, easy to use programming interface. The need for AE emerged from the observation that securely combining a confidentiality mode with an authentication mode could be error prone and difficult. This was confirmed by a number of practical attacks introduced into production protocols and applications by incorrect implementation, or lack, of authentication (including SSL/TLS). The need for AE emerged from the observation that securely combining a confidentiality mode with an authentication mode could be error prone and difficult. This was confirmed by a number of practical attacks introduced into production protocols and applications by incorrect implementation, or lack, of authentication (including SSL/TLS).

Encryption

- Input: plaintext, key, and optionally a header in plaintext that will not be encrypted, but will be covered by authenticity protection.
- Output: ciphertext and authentication tag (Message Authentication Code).

Decryption

- Input: ciphertext, key, authentication tag, and optionally a header.
- Output: plaintext, or an error if the authentication tag does not match the supplied ciphertext or header.

AES (Rijndael), RC6, MARS, Twofish, Serpent, CAST-256 have been used in this thesis.

Block Ciphers: In cryptography, a block cipher is a deterministic algorithm operating on fixed-length groups of bits, called a block, with an unvarying transformation that is specified by a symmetric key. Block ciphers operate as important elementary components in the design of many cryptographic protocols, and are widely used to implement encryption of bulk data.

A sketch of a Substitution-Permutation Network with 3 rounds, encrypting a plaintext block of 16 bits into a ciphertext block of 16 bits. The S-boxes are the S_i's, the P-boxes are the same P, and the round keys are the K_i's.
In block cipher we have used ARIA, IDEA, Triple-DES (DES-EDE2 and DES-EDE3), Camellia, SEED, Kalyna, RC5, Blowfish, TEA, Threefish, Skipjack, SHACAL-2, XTEA - these algorithms.

Block Cipher Modes Of Operation: In cryptography, a block cipher mode of operation is an algorithm that uses a block cipher to provide an information service such as confidentiality or authenticity. A block cipher by itself is only suitable for the secure cryptographic transformation (encryption or decryption) of one fixed-length group of bits called a block. A mode of operation describes how repeatedly to apply a cipher's single-block operation securely to transform amounts of data larger than a block. ECB, CBC, CBC ciphertext stealing (CTS), CFB, OFB, counter mode (CTR) are the modes have been used in this thesis.

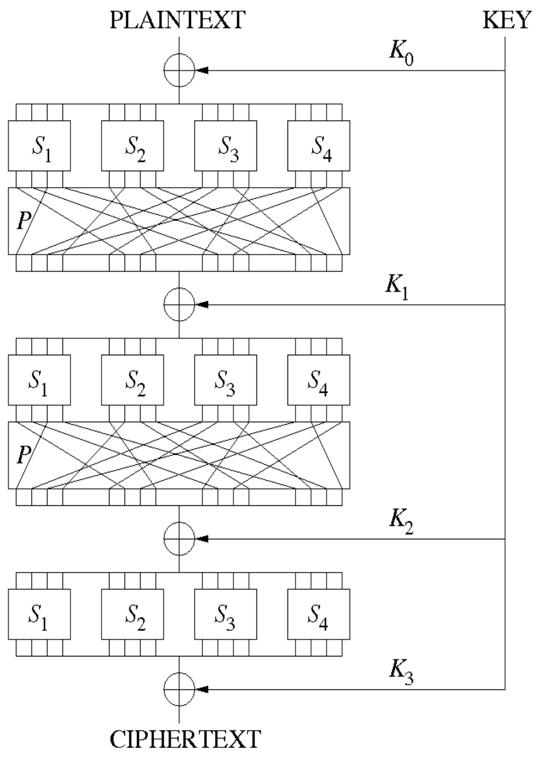


Fig 2: Cipher Text

Message Authentication Code: A message authentication code (MAC), sometimes known as a tag, is a short piece of information used to authenticate a message- in other words, to confirm that the message came from the stated sender (its authenticity) and has not been changed. The MAC value protects both a message's data integrity as well as its authenticity, by allowing verifiers (who also possess the secret key) to detect any changes to the message content. VMAC, HMAC, GMAC (GCM), CMAC, CBC-MAC, DMAC, Two-Track-MAC, BLAKE2 (BLAKE2b, BLAKE2s), Poly1305, SipHash- are the modes have been used in thesis.

Hash Functions: A cryptographic hash function is a hash function which takes an input (or 'message') and returns a fixed-size alphanumeric string. The string is called the 'hash value', 'message digest', 'digital fingerprint', 'digest' or 'checksum'). BLAKE2 (BLAKE2b, BLAKE2s), Keccack (F1600), SHA-1, SHA-2, SHA-3, Tiger, WHIRLPOOL, RIPEMD-128, RIPEMD-256, RIPEMD-160, RIPEMD-320 are the hash functions we have used.

Public Key Cryptography: In a public key encryption system, any person can encrypt a message using the public key of the receiver, but such a message can be decrypted only with the receiver's private key. For this to work it must be computationally easy for a user to generate a public and private key-pair to be used for encryption and decryption. The strength of a public key cryptography system relies on the degree of difficulty (computational impracticality) for a properly generated private key to be determined from its corresponding public key. Security then depends only on keeping the private key private, and the public key may be published without compromising security. RSA, DSA, Determinsitic DSA, ElGamal, Nyberg-Rueppel (NR), Rabin-Williams (RW), EC-based German Digital Signature (ECGDSA), LUC, LUCELG, DLIES (variants of DHAES), ESIGN are the algorims which have been used here.

Elliptic curve cryptography (ECC): is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. ECC requires smaller keys compared to non-ECC cryptography (based on plain Galois fields) to provide equivalent security. Elliptic curves are applicable for key agreement, digital signatures, pseudo-random generators and other tasks. Indirectly, they can be used for encryption by combining the key agreement with a symmetric encryption scheme. They are also used in several integer factorization algorithms based on elliptic curves that have applications in cryptography, such as Lenstra elliptic curve factorization.ECDSA, Determinsitic ECDSA, ECGDSA, ECNR, ECIES, ECDH, ECMQV are the algorithms which havebeen used here.

Key-Agreement Protocol: This is a protocol where two or more parties can agree on a key in such a way that both influence the outcome. If properly done, this precludes undesired third parties from forcing a key choice on the agreeing parties. Protocols that are useful in practice also do not reveal to any eavesdropping party what key has been agreed upon.

Many key exchange systems have one party generate the key, and simply send that key to the other party -the other party has no influence on the key. Using a key-agreement protocol avoids some of the key distribution problems associated with such systems. Diffie-Hellman (DH), Unified Diffie-Hellman (DH2), Menezes-Qu-Vanstone (MQV), Hashed MQV (HMQV), Fully Hashed MQV (FHMQV), LUCDIF, XTR-DH

CHAPTER 03

ASSESSMENT CRITERION AND PARAMETERS

This chapter talks about the assesment criteria that we have used, in details as well as parameters used. Then we discuss about the build environment model.

3.1 ASSESSMENT CRITERIA

In order to evaluate the system different tests has been conducted with two different libraries.

i. *Crypto++® 6.0.0.*

First library used is Crypto++® 6.0.0.

The reason to use the library is that it supports 32-bit and 64-bit architectures for many major operating systems and platforms and compilation using C++03, C++11, etc. runtime libraries; and a variety of compilers and IDEs, In speed tests seven open source security libraries with 15 block ciphers, it was the top performing library under two block ciphers, and did not rank below the average library performance under the remaining block ciphers.

ii. *FLECC in C*

The second library used is FLECC in C.

The reason to use FLECC is that it supports multiple elliptic curves at runtime and is designed to be executed on lightweight embedded processors. It is written in C, to make sure that it is supported by the compiler that comes with an embedded microprocessor.

The libraries are heavily researched for their credibility and their performance and usage statistics. The black box device that we used was not compatible with any of the libraries but these are best support for security reasoning in terms of performance and efficiency analysis.

We used different parameters with different input curves for the libraries and tried to generate batched results for the proposed algorithms taken into account by our tests.

We choose these two libraries to test in Raspberry Pi and Mac enabled systems. We have also taken into account of the usage of other libraries like Bouncy Castle, Botan, Wolfcrypt, Openssl etc. but *Crypto++*® and FLECC_IN_C were the clear winners for our algorithm performance and efficiency tests.

3.2 PARAMETERS USED

We used different input values and parameters for the testing purpose. The input datasets for different algorithms are different which varies algorithm to algorithm.

For testing purpose, in elliptic curve cryptography (FLECC) we have used:

- ➔ 5 input curves
    - ◆ Ranging from Secp192rl to Secp521rl.
    - ◆ With a Bit Size of 192 to 512.
    - ◆ Six-tuple is T =(p;a;b;G;n;h)

Two tables are given below that will explain the types of curvatures and their details.

Verifiably Random Elliptic Curve Domain Parameters Over Fp		
Curve	**Bit size**	**Six-tuple for specification**
Secp192r1	192	T =(p;a;b;G;n;h)
Secp224r1	224	T =(p;a;b;G;n;h)

Secp384r1	384	T =(p;a;b;G;n;h)
Secp256r1	256	T =(p;a;b;G;n;h)
Secp521r1	512	T =(p;a;b;G;n;h)

Table 1: Details Of Curves Used

Minimum bits	Curve	MDA	ECDSA Key size
80	Secp192r1	Sha 1, 224, 256, 384, 512	160 - 223
112	Secp224r1	Sha 224, 256, 384, 512	224 - 255
192	Secp384r1	Sha 384, 512	384 - 511
128	ecp256r1	Sha 256, 384, 512	256 - 383
26	secp521r1	Sha 512	512 +

Table 2: Details of curves used

For public-key cryptosystems: PKCS1v15 was used for key generation.

3.3 BUILDING ENVIRONMENT MODEL

Building Procedure:

The build procedure followed the below steps:

1. The generated input data was run through these algorithms.
2. The necessary findings for specific algorithm were printed.
3. Analysis of constrained-loop performance were taken into account.
4. A list was generated for the algorithms' performance in different platforms.

5. Computation process took place after every testing cycle ended.
6. Testing phase ended after the completion of cycles.
7. Calculation of which algorithm is better suited among comparative algorithms was done.
8. Results were generated based on the work done.

Environmental Model:

The environmental model is shown in the next page.

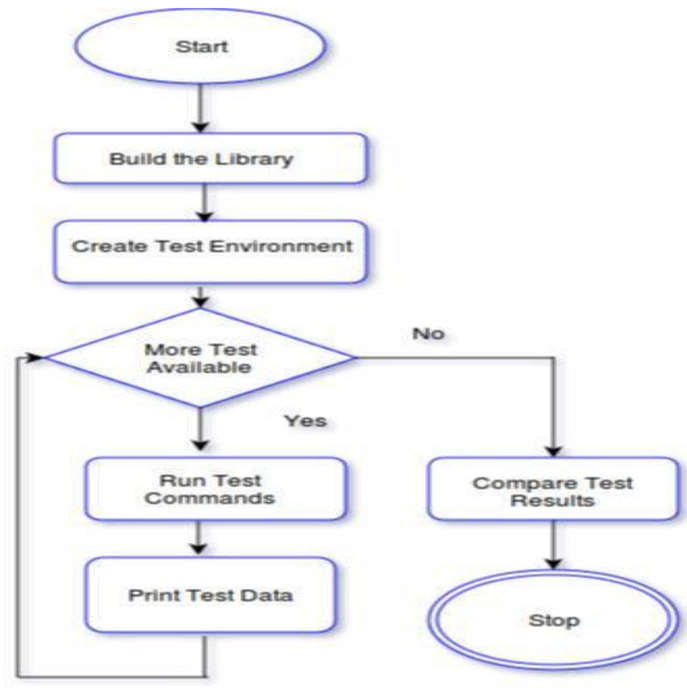


Fig 3: Testing procedure

Test Model:

The test model is comprised of:

- 8 states
    - Starting state
    - Building library
    - Creating Test environment
    - Checking for more tests
    - Running test commands
    - Printing test data
    - Comparing results
    - Stopping/ result state
- One decision state (More Test Available)
    - Yes: Go to -> Run Test Commands
    - No: Go to -> Compare Test Results
- One loopback state (Print Test Data)
    - Loopback -> More Test Available
- One Starting State
- One Stopping State

This model is then used for all test data input, calculation, output and analyzation. The test is comprised of groups of several algorithms and then local computation of the testing group. After all the tests were finished a global test occurred and the analyzation was done.

CHAPTER 04

EXPERIMENTAL SETUP AND RESULT ANALYSIS

This chapter goes through the whole process of setting up necessary hardware, generating required library files, preparing neede test environment and then finally conducting tests and test environment.

4.1 SETTING UP HARDWARE

Getting the results of library implementation in specific hardware is a challenge because the device was not natively supported by the libraries. The estimated time is based on how much time is needed for a) building the library for specific platform and b) creating test environment for the specific platform and c) carrying out result data and calculations.

This system model makes use of test environment's CPU. The test system makes use of CPU cores, which means performance would be better for higher number of core count. For conducting tests on a low powered device, the platform used was Raspberry Pi 3 model B. It has a 1.2 GHz ARM V8 Quad Core CPU.

The system is also been implemented in OS X System to know how these algorithms perform in artificially constrained environment. The CPU used in this system is a 3rd generation intel® Core™ i5 processor with family model number 3470. It's a quad core processor with 3.2 Ghz base clock speed.

4.2 GENERATING LIBRARY FILES

To achieve the results, primarily, the use of the Crypto++® Library (version 5.6.5) has been made. The testing datasets were generated, optimized and put to the test environment. The used platform is Raspberry Pi 3 Version B. The system was readied to compile the library and run the test codes afterwards.

The implementation of Crypto++® Library consisted of the following procedural approach:

1. Building the library
    a. The use of github repository of Crypto++® to download the library codes to the local environment has been made.
    b. Used Clang on Raspberry Pi to build the library.

Secondarily, the use of FLECC_IN_C,[3] a library written in C for implementing cryptographic algorithm, has been used to test the data. It is again implemented in Raspberry Pi 3 Version B. The benchmark information and reviews are illustrated in the below graphs.

[3]This is a free C++ library of cryptographic schemes. More about Crypto++®can be found here: https://github.com/weidai11/cryptopp

The implementation of FLECC_IN_C required the below steps.

1. Building the library
    a. github repository of FLECC_IN_C, to download the library codes to the local environment, was used.
    b. Used CMake to compile library codes.

4.3 PREPARING TEST ENVIRONMENT

Crypto++®

1. Creating test environment
    a. For creating the test environment the utilization of *make* command over cryptest.exe file was done.
2. Getting test result data
    a. For getting test result data, it was added to the test CPU frequency and issued test command.

   b. HTML was used to view the result.

FLECC_IN_C

1. Creating test environment
   a. CMake, with *make* options to build the necessary test environment, was used.
   b. Used doxygen for documentation.
   c. Used cov & genhtml for code coverage testing.
   d. Used clang-format for automatic code base reformatting.
2. Getting test result data
   a. For getting test result data: issued the *build* flag with CMake.
   b. HTML and CSS were used to view the result.

We have listed the total number of tools used in our test environment and pasted it below. Some of the tools are interfaces and some are softwares used in conjunction with the hardware to achieve results.

Tools Used
1. CMake 3.8[4]
2. Clang 3.9 (for compiling C++ files in Raspberry Pi)
3. Raspbian Jessie (with kernel version 4.4)
4. Doxygen 2.0
5. LCOV 1.13
6. GenHTML 1.0
7. Clang-Format (Bundled with Clang 3.9)

Table 3: Tools used

[4]Details about CMake library can be found at: https://cmake.org/documentation/

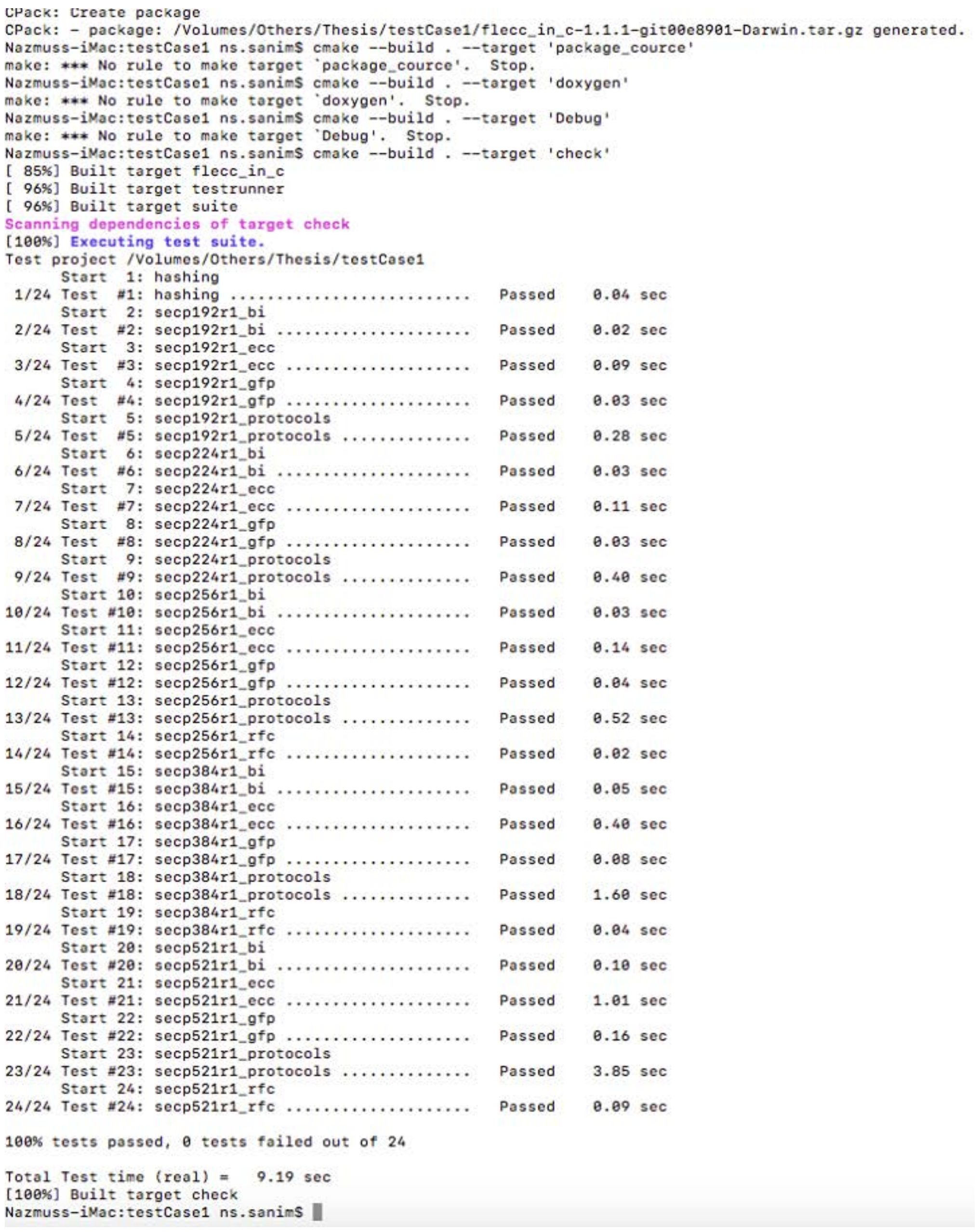
4.4 CONDUCTING TESTS

The tests were conducted on MacOS and Raspberry Pi respectively. For MacOS mac terminal was used to achieve the results. Cmake was used to generate our own specific testing system and compile the files in C. The tests were based on the performance of the algorithms at hand. It measured the time to encode and decode specific test data and given an output of how much time was actually needed to complete the whole cycle. For every algorithm this procedure went on and it printed the result on terminal window. Every algorithm took different time to generate output and that output was later being used with conjunction to other parameters to get the actual analyzation and result.

The test in MacOS was taken after thoroughly going through the algorithms and creating our own suitable algorithm list and parameters that is already incorporated in FLECC_IN_C. Then, after making some necessary tweaking, out test system was good to go with the current cpu and memory configuration. Our test did not use any kind of CUDA or openCL accelerated computing, we used just the host CPU to get constrained environment of an IoT device. The test list of algorithms was then compiled and then ran through the terminal.

All the tests took their respective time to complete. After the tests got completed we got an idea of how many tests actually ran successfully by knowing the number and percentage of code coverage report.

The following image shows the output of running FLECC_IN_C in MacOS:

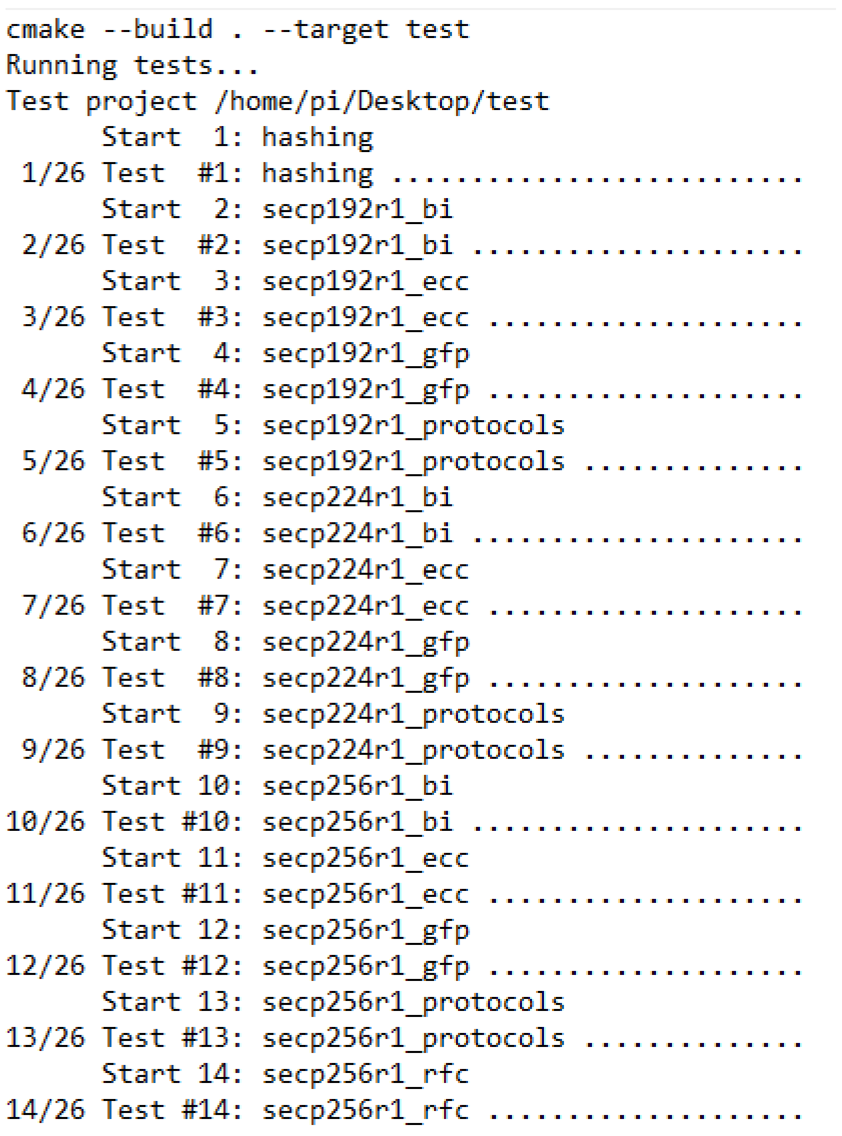


```
CPack: Create package
CPack: - package: /Volumes/Others/Thesis/testCase1/flecc_in_c-1.1.1-git00e8901-Darwin.tar.gz generated.
Nazmuss-iMac:testCase1 ns.sanim$ cmake --build . --target 'package_cource'
make: *** No rule to make target `package_cource'.  Stop.
Nazmuss-iMac:testCase1 ns.sanim$ cmake --build . --target 'doxygen'
make: *** No rule to make target `doxygen'.  Stop.
Nazmuss-iMac:testCase1 ns.sanim$ cmake --build . --target 'Debug'
make: *** No rule to make target `Debug'.  Stop.
Nazmuss-iMac:testCase1 ns.sanim$ cmake --build . --target 'check'
[ 85%] Built target flecc_in_c
[ 96%] Built target testrunner
[ 96%] Built target suite
Scanning dependencies of target check
[100%] Executing test suite.
Test project /Volumes/Others/Thesis/testCase1
      Start  1: hashing
 1/24 Test  #1: hashing ..........................   Passed    0.04 sec
      Start  2: secp192r1_bi
 2/24 Test  #2: secp192r1_bi .....................   Passed    0.02 sec
      Start  3: secp192r1_ecc
 3/24 Test  #3: secp192r1_ecc ....................   Passed    0.09 sec
      Start  4: secp192r1_gfp
 4/24 Test  #4: secp192r1_gfp ....................   Passed    0.03 sec
      Start  5: secp192r1_protocols
 5/24 Test  #5: secp192r1_protocols ..............   Passed    0.28 sec
      Start  6: secp224r1_bi
 6/24 Test  #6: secp224r1_bi .....................   Passed    0.03 sec
      Start  7: secp224r1_ecc
 7/24 Test  #7: secp224r1_ecc ....................   Passed    0.11 sec
      Start  8: secp224r1_gfp
 8/24 Test  #8: secp224r1_gfp ....................   Passed    0.03 sec
      Start  9: secp224r1_protocols
 9/24 Test  #9: secp224r1_protocols ..............   Passed    0.40 sec
      Start 10: secp256r1_bi
10/24 Test #10: secp256r1_bi .....................   Passed    0.03 sec
      Start 11: secp256r1_ecc
11/24 Test #11: secp256r1_ecc ....................   Passed    0.14 sec
      Start 12: secp256r1_gfp
12/24 Test #12: secp256r1_gfp ....................   Passed    0.04 sec
      Start 13: secp256r1_protocols
13/24 Test #13: secp256r1_protocols ..............   Passed    0.52 sec
      Start 14: secp256r1_rfc
14/24 Test #14: secp256r1_rfc ....................   Passed    0.02 sec
      Start 15: secp384r1_bi
15/24 Test #15: secp384r1_bi .....................   Passed    0.05 sec
      Start 16: secp384r1_ecc
16/24 Test #16: secp384r1_ecc ....................   Passed    0.40 sec
      Start 17: secp384r1_gfp
17/24 Test #17: secp384r1_gfp ....................   Passed    0.08 sec
      Start 18: secp384r1_protocols
18/24 Test #18: secp384r1_protocols ..............   Passed    1.60 sec
      Start 19: secp384r1_rfc
19/24 Test #19: secp384r1_rfc ....................   Passed    0.04 sec
      Start 20: secp521r1_bi
20/24 Test #20: secp521r1_bi .....................   Passed    0.10 sec
      Start 21: secp521r1_ecc
21/24 Test #21: secp521r1_ecc ....................   Passed    1.01 sec
      Start 22: secp521r1_gfp
22/24 Test #22: secp521r1_gfp ....................   Passed    0.16 sec
      Start 23: secp521r1_protocols
23/24 Test #23: secp521r1_protocols ..............   Passed    3.85 sec
      Start 24: secp521r1_rfc
24/24 Test #24: secp521r1_rfc ....................   Passed    0.09 sec

100% tests passed, 0 tests failed out of 24

Total Test time (real) =   9.19 sec
[100%] Built target check
Nazmuss-iMac:testCase1 ns.sanim$ 
```

Fig 4: Running on MacOS

After the successful run on a Mac machine we started to take into account our black box device namely Raspberry Pi. The Pi was loaded up with the default operating system, the raspberry pi Jessy (4.8). We installed the necessary scripts and then started setting those up for raspberry pi. Then, we selected our own tests for raspberry pi and started running the tests on this constrained environment.

The tests looked like the below image in raspberry pi.



```
cmake --build . --target test
Running tests...
Test project /home/pi/Desktop/test
      Start  1: hashing
 1/26 Test  #1: hashing ..........................
      Start  2: secp192r1_bi
 2/26 Test  #2: secp192r1_bi .....................
      Start  3: secp192r1_ecc
 3/26 Test  #3: secp192r1_ecc ....................
      Start  4: secp192r1_gfp
 4/26 Test  #4: secp192r1_gfp ....................
      Start  5: secp192r1_protocols
 5/26 Test  #5: secp192r1_protocols ..............
      Start  6: secp224r1_bi
 6/26 Test  #6: secp224r1_bi .....................
      Start  7: secp224r1_ecc
 7/26 Test  #7: secp224r1_ecc ....................
      Start  8: secp224r1_gfp
 8/26 Test  #8: secp224r1_gfp ....................
      Start  9: secp224r1_protocols
 9/26 Test  #9: secp224r1_protocols ..............
      Start 10: secp256r1_bi
10/26 Test #10: secp256r1_bi .....................
      Start 11: secp256r1_ecc
11/26 Test #11: secp256r1_ecc ....................
      Start 12: secp256r1_gfp
12/26 Test #12: secp256r1_gfp ....................
      Start 13: secp256r1_protocols
13/26 Test #13: secp256r1_protocols ..............
      Start 14: secp256r1_rfc
14/26 Test #14: secp256r1_rfc ....................
```

Fig 5: Running on Raspberry Pi

Again, after conducting the tests we took into account the test results and the coverage report. This result was analysed and were computed for, to get the idea of which algorithm took less time in terms of encryption and decryption.

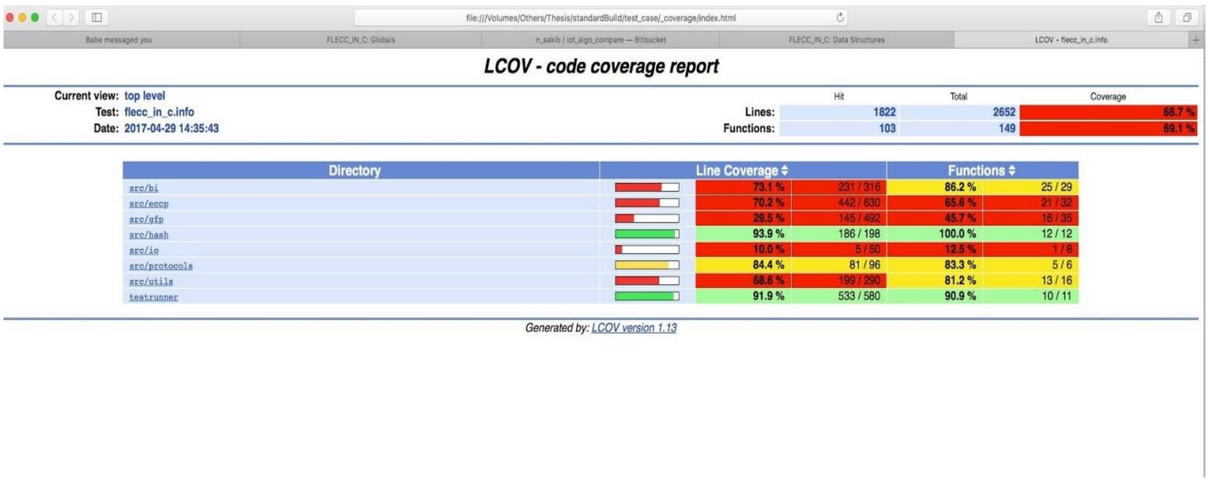
The same pattern was followed while testing with Crypto++®. Crypto++® consists of several different algorithms and was used for the same purpose. Then, the results were being generated and were taken into account after thoroughly running the tests in the black box environment.

Crypto++® was run again in both MacOS environment and also in the raspberry pi system. After the test was finished we got the results. The result and the accuracy details are discussed in the next parts.

4.5 ACCURACY MEASUREMENT

Accuracy measurement was one of the biggest part of our testing. We used LCOV to get the code coverage report and from there we get a rough idea of how much code was actually been used to run the total test. From LCOV we get:

- A coverage of 68.7% of lines (1822 out of 2852 lines)
- A coverage of 68.1% functions (103 out of 148 functions)


file:///Volumes/Others/Thesis/standardBuild/test_case/_coverage/index.html

Babe messaged you | FLECC_IN_C: Globals | n_sakib / iot_algo_compare — Bitbucket | FLECC_IN_C: Data Structures | LCOV - flecc_in_c.info

LCOV - code coverage report

Current view: top level
Test: flecc_in_c.info
Date: 2017-04-29 14:35:43

	Hit	Total	Coverage
Lines:	1822	2652	68.7 %
Functions:	103	149	69.1 %

Directory	Line Coverage			Functions	
src/bi		73.1 %	231 / 316	86.2 %	25 / 29
src/eccp		70.2 %	442 / 630	65.6 %	21 / 32
src/gfp		29.5 %	145 / 492	45.7 %	16 / 35
src/hash		93.9 %	186 / 198	100.0 %	12 / 12
src/io		10.0 %	5 / 50	12.5 %	1 / 8
src/protocols		84.4 %	81 / 96	83.3 %	5 / 6
src/utils		68.6 %	199 / 290	81.2 %	13 / 16
testrunner		91.9 %	533 / 580	90.9 %	10 / 11

Generated by: LCOV version 1.13

Fig 6: Code coverage

LCOV and genHTML were to be set up in our test environments. LCOV scans the directory and looks for the functions and line numbers and stores the information. Then after running the codebase, LCOV again checks the codebase real time, and tries to identify which of the lines and functions were covered in the runtime.

CHAPTER 05

GENERATING AND ANALYSING RESULT

This chapter talks in necessary details about in depth result analysis

5.1 RESULT ANALYSIS

A. *RESULTS USING* CRYPTO++®

First library used is Crypto++® 6.0.0.

First Host: Raspbian Jessie

Host OS is RASPBIAN JESSIE WITH PIXEL (ARM 64/32-bit). The Host CPU is an ARMv8-A, frequency is 1.22 GHz.

For all algorithms discussed in this section the input keys are generated based on the standard mentioned in section C.

As shown in Figure 7, for the first part of the experiment, we considered a script with 16 arguments to compare the performance on the two factors as shown in the graph below. It analyses the performance of popular cryptographic primitive types such as specific stream and block ciphers as well as AES models and MAC algorithms. Time taken for setting up key and initialization vector is lowest for ChaCha20 and highest for DES-XEX3/CTR. However, bytes processed are fastest for HMAC and lowest for Threefish/CTR. Overall, on average, from the algorithms tested in the specific constraint environment considered here, the optimal performance is considered as ChaCha and HMAC among these algorithms that here is considered of the primitive type.

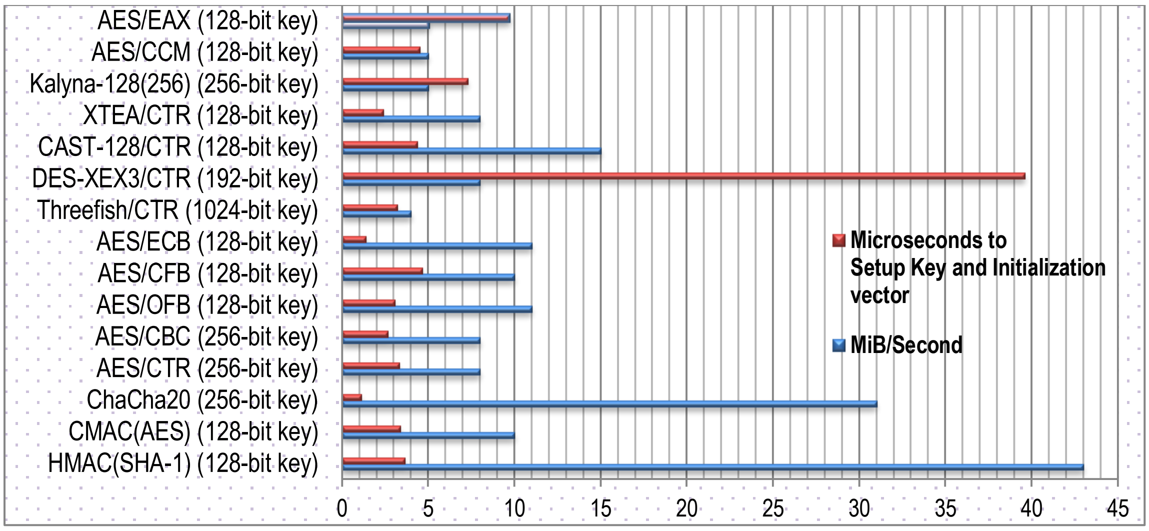


Fig 7: Bytes per second and time to set up key and IV

For the second part, the experiment was conducted in a batch of 4 test scripts; each contained 17 test arguments, as shown in Figure 8. The first batch has ESIGN, DSA, NR, RW, LUC and RSA. Other than LUC and RSA the key-sign and verification process had to be used and for those encryption and decryption methodologies were used. In this test ESIGN verification and signature took a total of 1.94 (0.51 + 1.43) milliseconds which is the lowest time taken by an algorithm to execute. Here the worst performance came from LUC 1024 which took 0.63 milliseconds for encryption and 21.96 milliseconds for decryption (A total of 22.59 milliseconds).

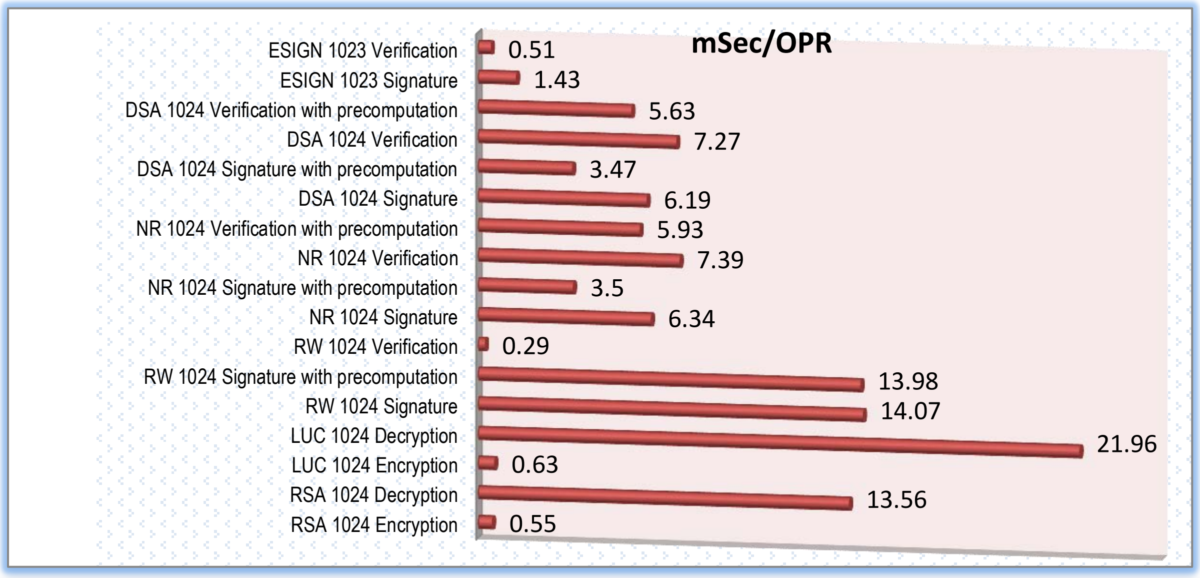


Fig 8: Milliseconds per operation

The second batch contained 16 other test vectors, as shown in Figure 9. This time it has ECIES, MQV, LUCDIF, DH and two variants of XTR-DH. Here, MQV 1024 managed to operate in lowest time resulting 15.35 (11.92 + 3.43) milliseconds with precipitation. The worst performance was from a variant of XTR-DH which is XTR-DH 342. It took 57.65 (19.21 + 38.44) milliseconds to conduct the operation.

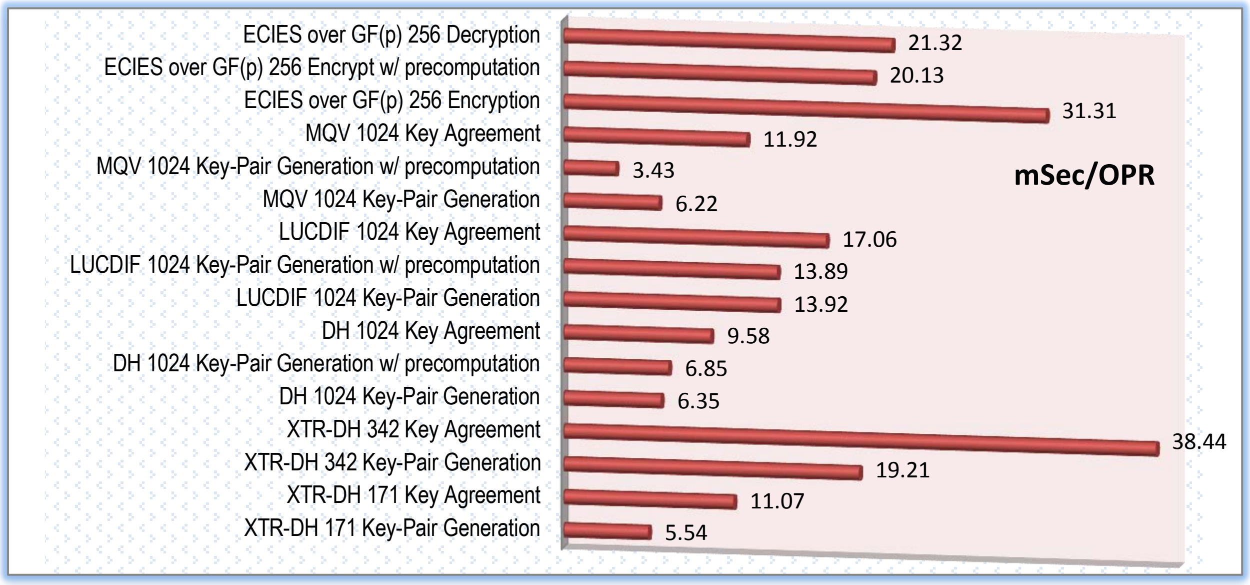


Fig 9: Milliseconds per operation

The third batch contained ECMQVC, ECDHC, ECGDSA, ECDSA, ECDSA-RFC6979 and ECDSA, as shown in Figure 8. For this test the vectors are over GF(p) 256. Here the best performing algorithm was ECDHC over GF(p) 256 resulting 25.83 (10.12 + 15.71) milliseconds with precomputed key-pair generation. The worst performance was from ECGDSA resulting 59.21 (41.74 + 17.47) milliseconds.

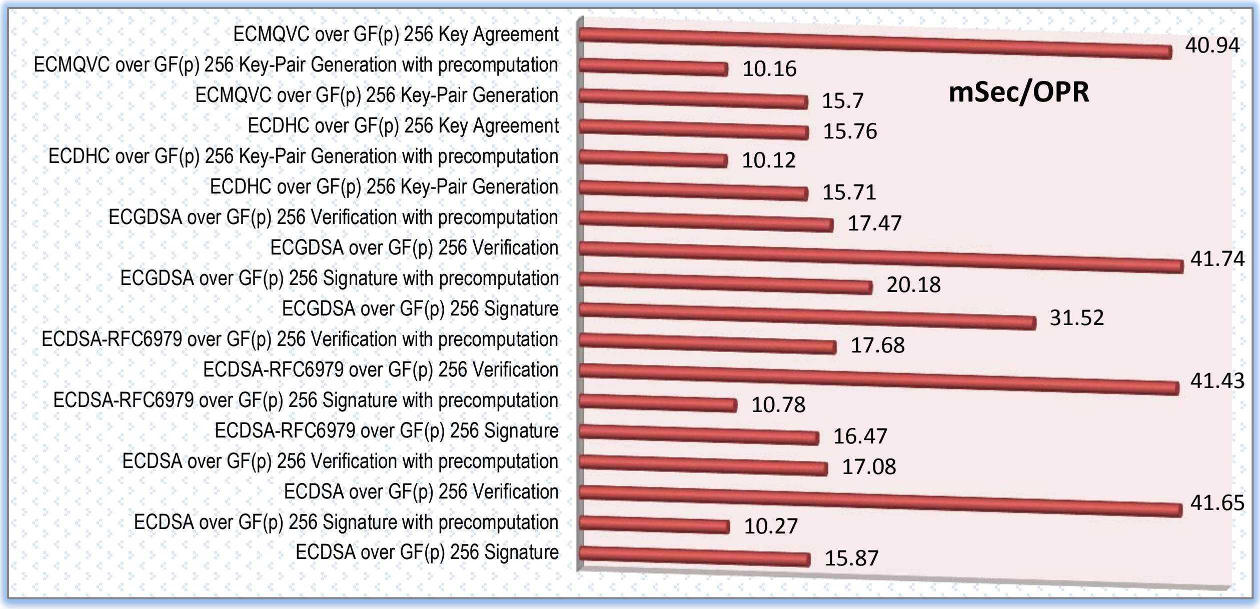


Fig 10: Milliseconds per operation

The final batch consisted of ECMQVC, ECDHC, ECGDSA, ECDSA-RFC6979 and finally ECIES, as shown in Figure 11. These are over GF(p) 233 in this test. ECMQVC over GF(p) 233 came out with the best result with 60.24 (46.64 + 13.58) milliseconds. The worst was 120.68 (27.28 + 93.4) milliseconds from ECGDSA.

Overall, on average, from the algorithms tested in our specific constraint environment under the key generation standard used, the optimal performance is considered as that of ESIGN signature and verification and least optimal is of ECGDSA among these algorithms that here is considered of not the primitive type.

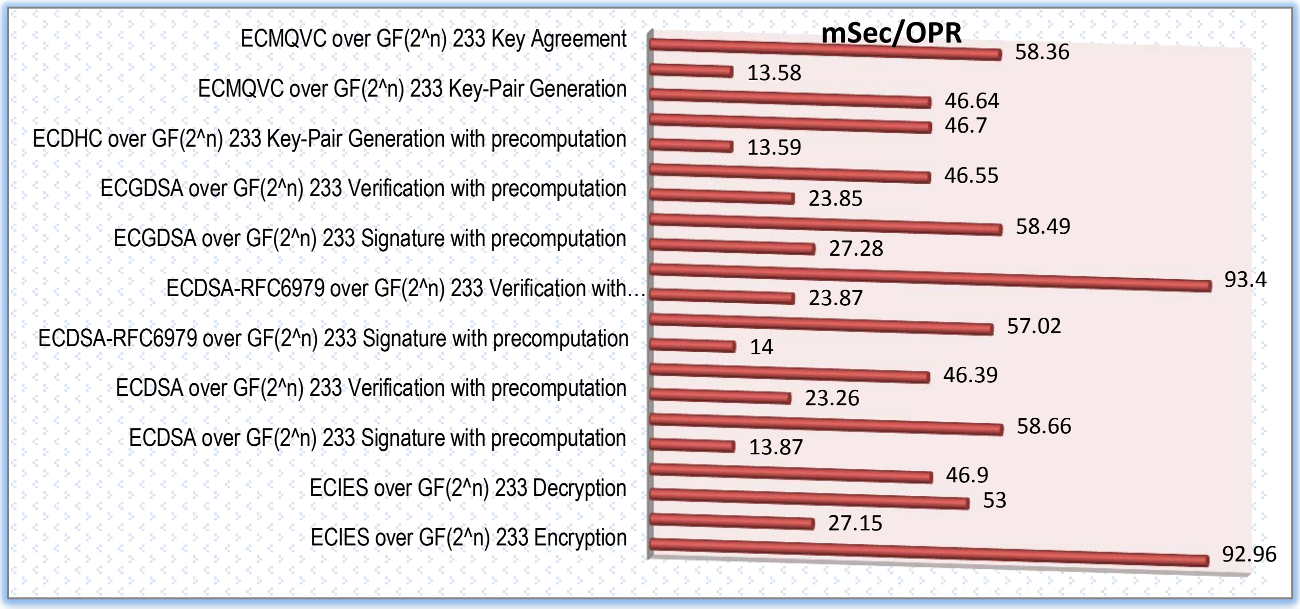


Fig 11: Milliseconds per operation

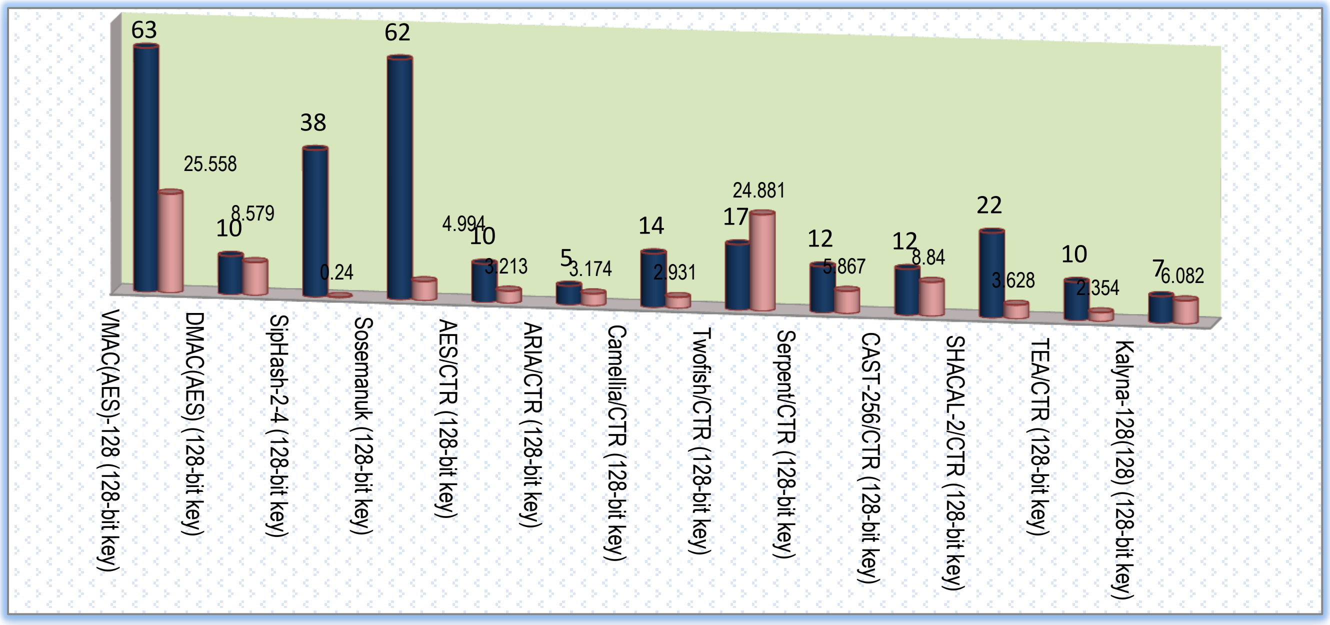


Fig 12: Time to setup key and IV

As shown in Figure 12, for the first part of the experiment, we considered a script with 13 arguments to compare the performance on the two factors as shown in the graph below. It analyses the performance of quite popular and useful cryptographic primitive types such as specific stream & block ciphers as well as AES models and MAC algorithms. Time taken for setting up key and initialization vector is lowest for ARIA/CTR and highest for SOSEMANUK. However, bytes processed are slowest for VMAC. Overall, on average, from the algorithms tested in the specific constraint environment considered here, the optimal performance is considered as Kalyna and ARIA among these algorithms that here is considered of the primitive type.

The final batch consisted of DLIES, RSA, LUC, etc. as shown in Figure 13. These are over 1024 bits mostly in this test LUCDIF and RSA came out with the best result with 7.6 and 14.1 milliseconds approximately. The worst was 44.9 milliseconds approximately from LUCEL G.

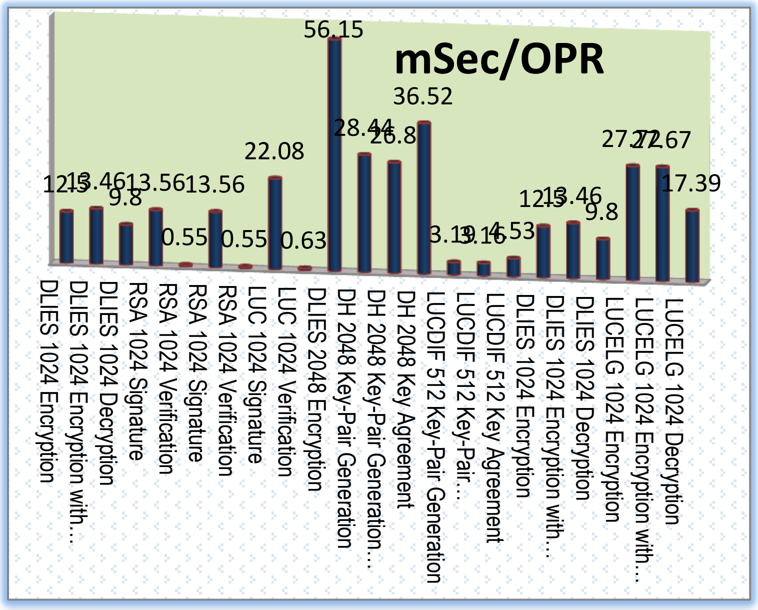


Fig 13: Milliseconds per operation (test 2)

Overall, on average, from the algorithms tested in our specific constraint environment under the key generation standard used, the optimal performance is considered as that of RSA signature and least optimal is of LUCELG among these algorithms that here is considered of not the primitive type.

Second Host: MAC OS

Host OS is MacOS Sierra (10.11.3). The Host CPU is an Intel® Core™ i5-3470 processor with a frequency of 3.20 Ghz.

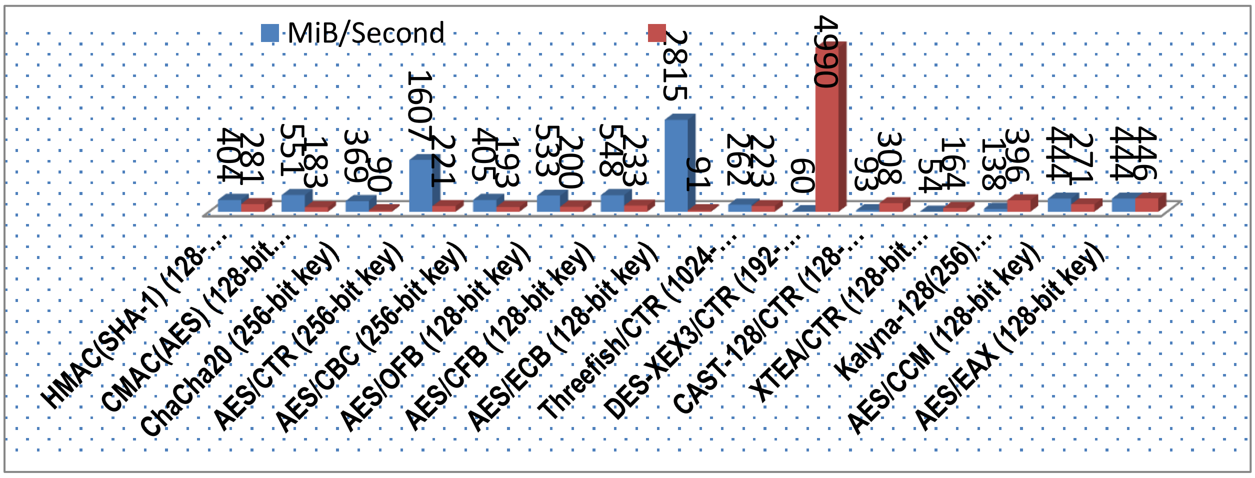


Fig 14: Bytes per second and time to set up key and IV

The first group shown in Figure 14 consists of HMAC, CMAC, ChaCha, AESCTR, AES/CBC, AES/OFB, AES/CFB, AES/ECB, Threefish, DES, CAST-128, XTEA/CTR, Kalyn, AES/CCM and finally AES/EAX. From the graph, To setup key and initialize vectors AES/CTR took the lowest time of 90 ms. CAST-128 took a big 4980 ms to complete the task.

After the setup the encryption and decryption test took place. XTEA/CTR took the least time of 54 ms to run the encrypt and decrypt routine. Threefish took the most amount of time in this routine which is 2815 ms.

The best overall was achieved with XTEA/CTR with a setup time of 164 ms and a run time of 54 ms which is 218 in total.

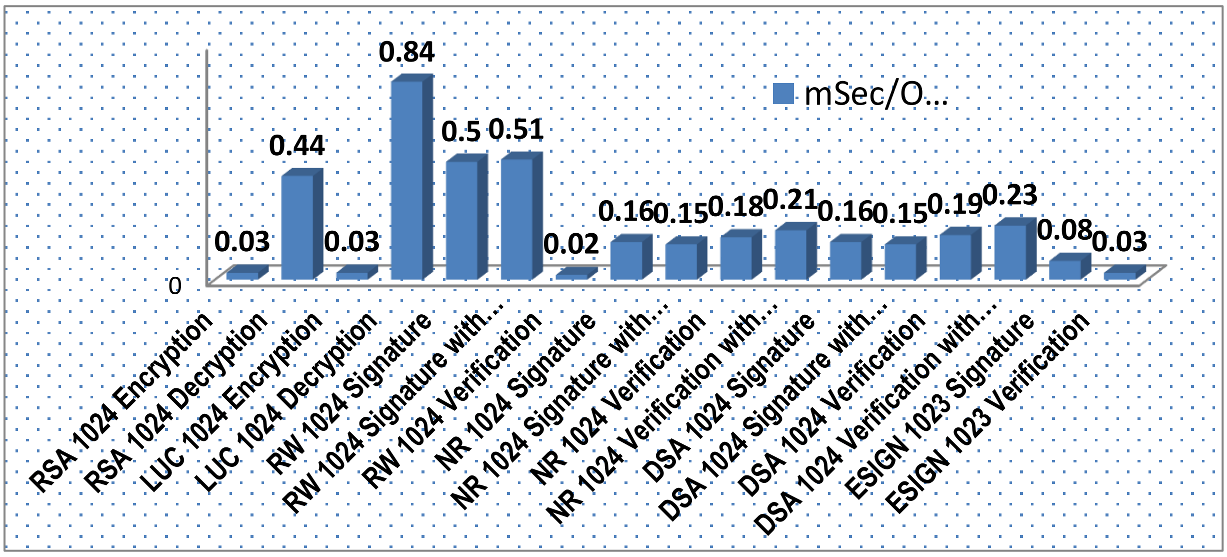


Fig 15: Milliseconds per operation

Figure 15 shows the operation per second characteristics of the algorithms. This group consists of RSA, LUC, RW, NR, DSA and ESIGN (1024). Both their encryption and decryption performance were taken into account.

Here, The lowest time was taken by the verification process of NR 1024 signatures. The highest time taken for decryption is by RW 1024 which is 0.84 mSec/OPR.

The lowest rate in encryption process was produced by 3 algorithms. They are RSA, ESIGN and LUC. All of them took 0.03 mSec per operation.

The best overall encryption and decryption process was performed by ESIGN algorithm with a signature generation time 0.08 mSec/OPR and verification time of 0.03 mSec/OPR in total 0.11 mSec/OPR.

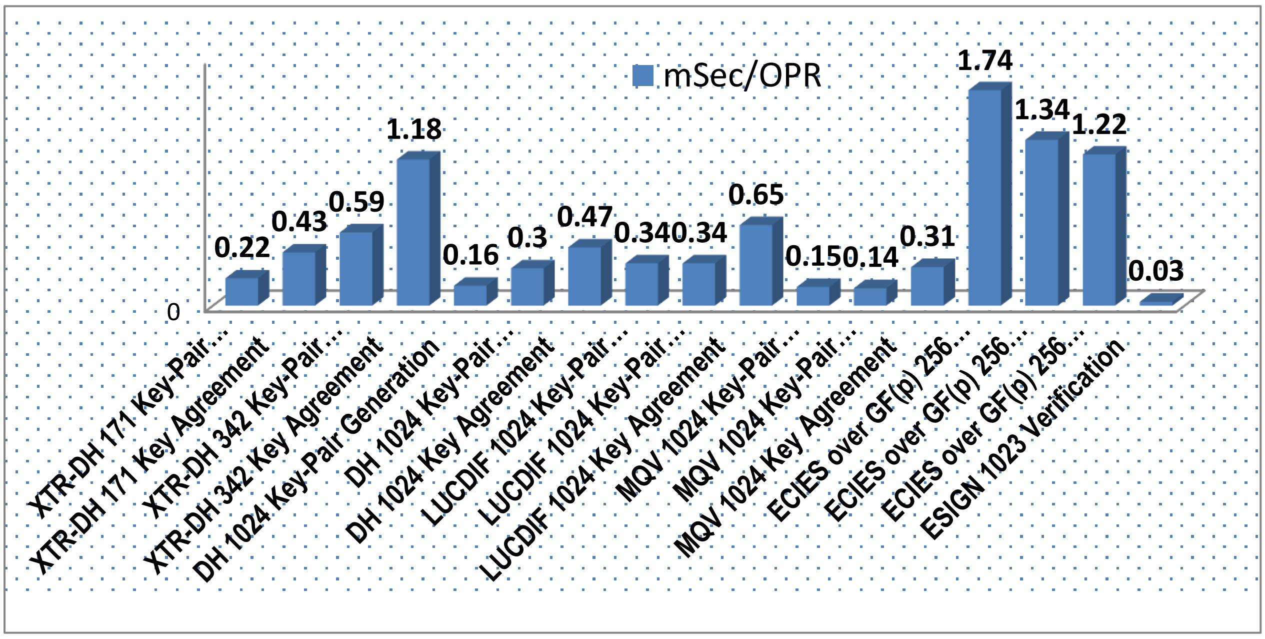


Fig 16: Milliseconds per operation

Figure 16 follows the same categorization as figure 11. It consists of XTR-DH, DH, LUCDIF, MQV ECIES over GF(p) and ESIGN 1023 (Other algorithms are in 1024 bit flavors).

Here, Again ESIGN 1023 took the best place with lowest time of 0.03 mSec/OPR.

ECIES took the most amount of time which is 1.74 mSec/OPR was the highest in this range.

The best overall in this category is taken by MQV with a key agreement time of 0.14 mSec and key-pair generation of 0.15 mSec.

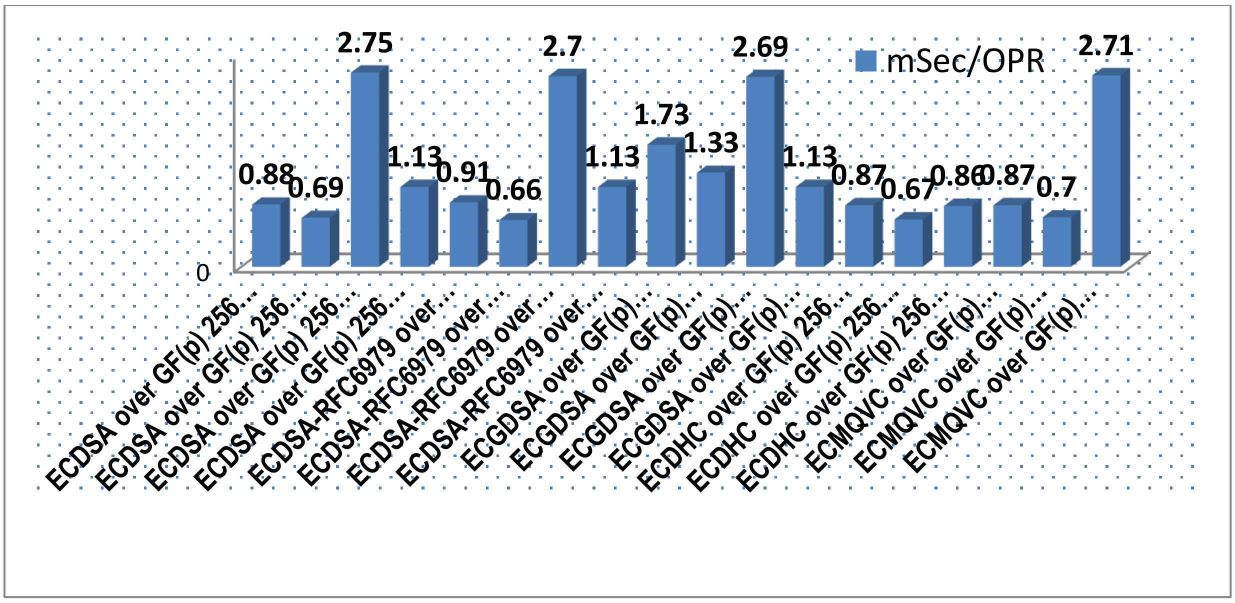


Fig 17: Milliseconds per operation

Figure 17 represents algorithms namely ECDSA, ECDSA-RFC, ECGDSA, ECDHC, ECMQVC. Here, The lowest time was taken by ECDSA-RFC signature generation which is 0.66 mSec/OPR. ECDSA verification took the most amount of time which is 2.75 mSec/OPR. The best overall was taken by ECMQVC with 1.57 mSec/OPR.

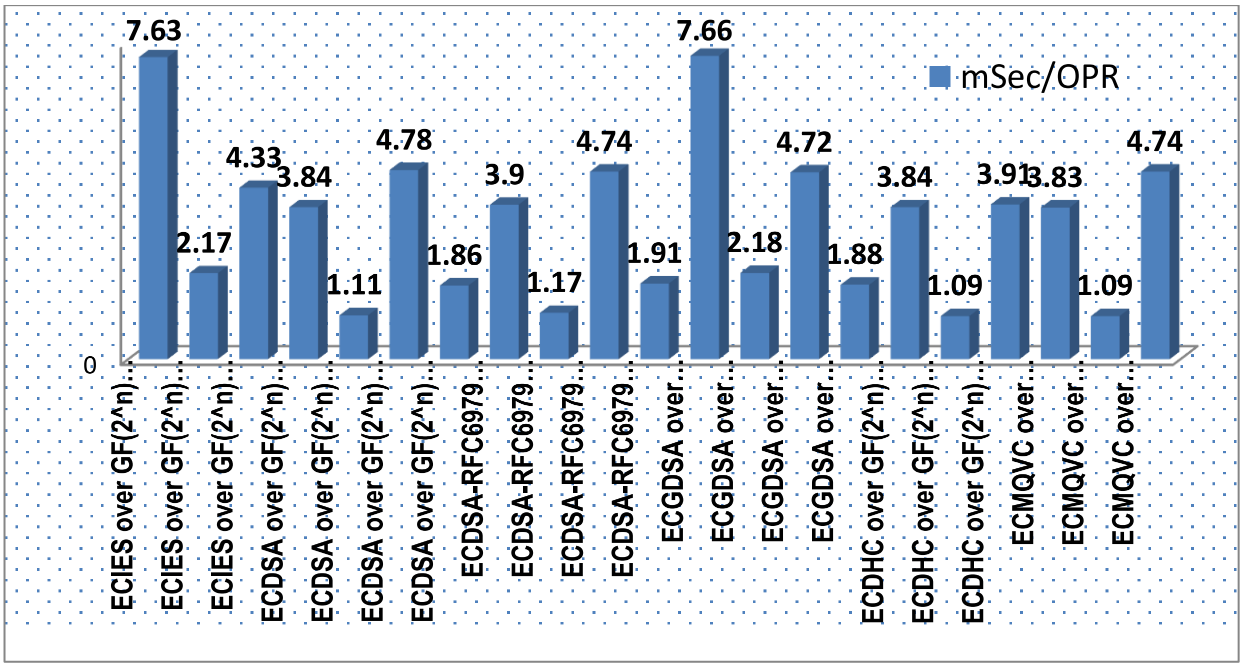


Fig 18: Milliseconds per operation

The next batch is shown in Figure 18. It consists of ECIES, ECSDA, ECDSA-RFC, ECGDSA, ECDHC and ECMQVC.

Here, The lowest time is taken by the key-pair generation phase of ECDHC and ECMQVC. Both of them have taken 1.09 mSec/OPR.

The highest time however, is taken by ECGDSA which is 7.66 mSec/OPR.

The best overall score was from ECDSA. It took around 3.84 mSec/OPR for signature verification. It also took 1.11 mSec/OPR to generate the signature with pre-computation and 4.78 mSec/OPR without pre-computation.

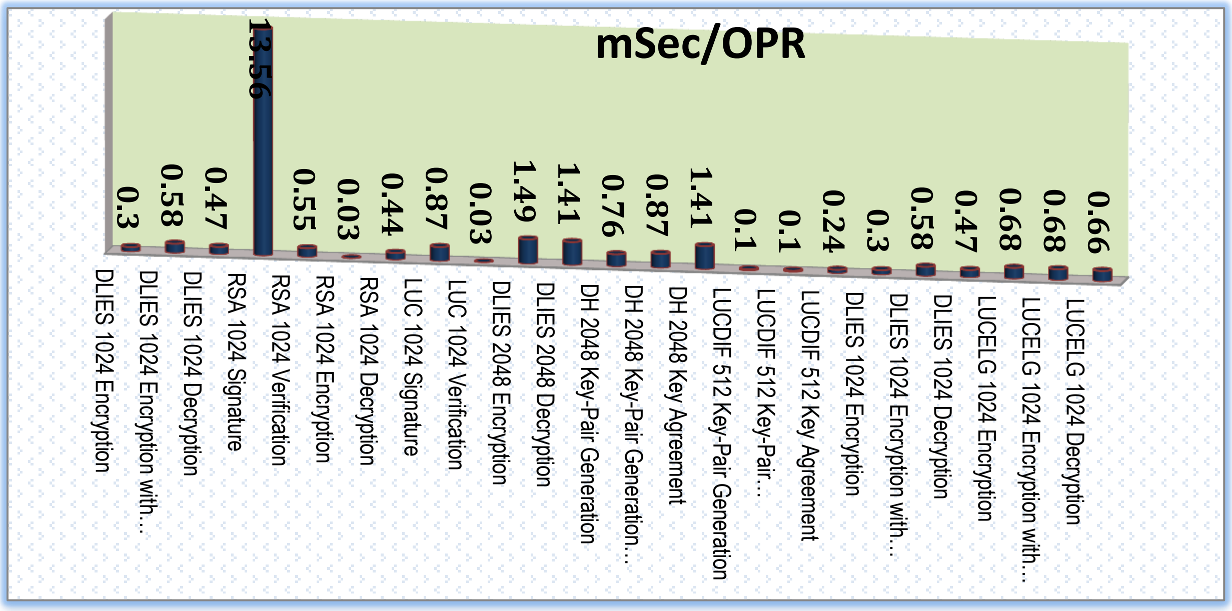


Fig 19: Milliseconds per operation

Next batch (Figure 19) consists of LUCEL G(1024), DLIES(1024), LUCDIF(512), DH(20148), DLIES(2048), LUC(1024) and RSA(1024). Here, RSA took huge amount of time resulting 13.56 mSec/OPR where other algorithms were mostly under 2 mSec/OPR.

The lowest time taken was by LUCDIF which is an impressive 0.1 mSec/OPR. LUCDIF takes 0.1 mSec for key-pair generation and also the same 0.1 mSec for the agreement.

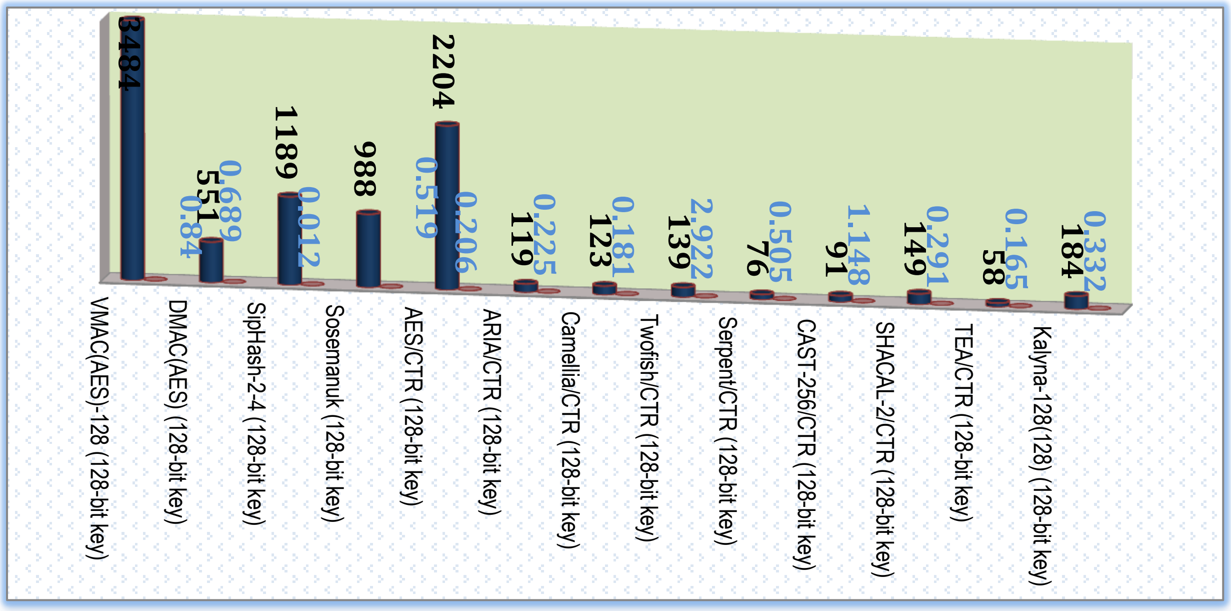


Fig 20: Milliseconds per operation

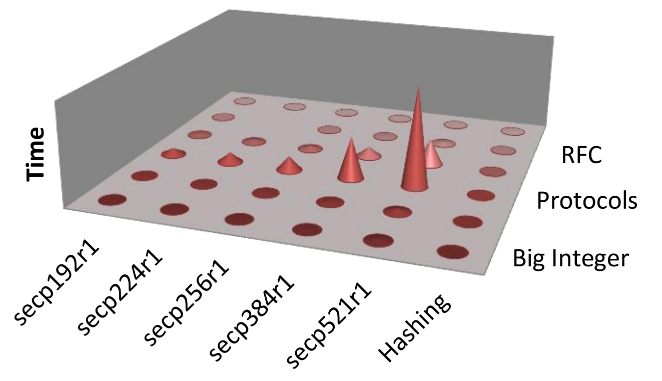
Lastly, in Figure 20, we got Kalyna-128, TEA/CTR, the SHACAL, CAST-256, Twofish, Camellia, ARIA/CTR, AES/CTR, Sosemanuk, SipHash-2, DMAC(AES), and VMAC(AES).

Here, The least amount of time was taken by Camellia/CTR which is 0.181 mSec/OPR. It has taken a total of 123.181 mSec for both the key generation and key verification purpose.

B. *Results using FLECC_IN_C.*

FLECC_IN_C or **F**lexible **E**lliptic **C**urve **C**ryptography shows detailed information of the elliptic curve algorithm when implemented and tested. The library used is FLECC_IN_C to determine the overall performance of an elliptic curve security algorithm. Here, ECDSA signatures is used (FIPS 186-3 standard) along with EC-DH key exchange.

First Host: Raspbian Jessie



	secp192r1	secp224r1	secp256r1	secp384r1	secp521r1	Hashing
■ Big Integer	0.2	0.26	0.25	0.54	1.26	-0.01
■ Prime Galois Fields	0.24	0.32	0.39	1.05	2.48	-0.01
■ Protocols	4.33	6.39	8.98	26.72	63.7	-0.01
□ ECC	1.12	1.65	2.1	6.4	16.51	-0.01
□ RFC	-0.01		0.22	0.57	1.39	-0.01
□ Hashing	-0.01	-0.01	-0.01	-0.01	-0.01	0.24

Fig 21: Performance in Pi

Now, for secp192r1 it took a total of 5.86 milliseconds to generate the key and create the signature. For secp224r1, secp256r1, secp384r1, secp521 it took 8.62, 11.94, 35.28, 85.34 respectively shown in Figure 21.

Hashing functions are in negative values; which means it did not run on any of the mentioned curves. From the Figure 21, therefore we can conclude that Big Integer processing as well as ECC and RFC processing remained more or less stable throughout and the protocols processing step was the costliest, hence it could be the area of modification for easiest improvement.

Second Host: MAC OS

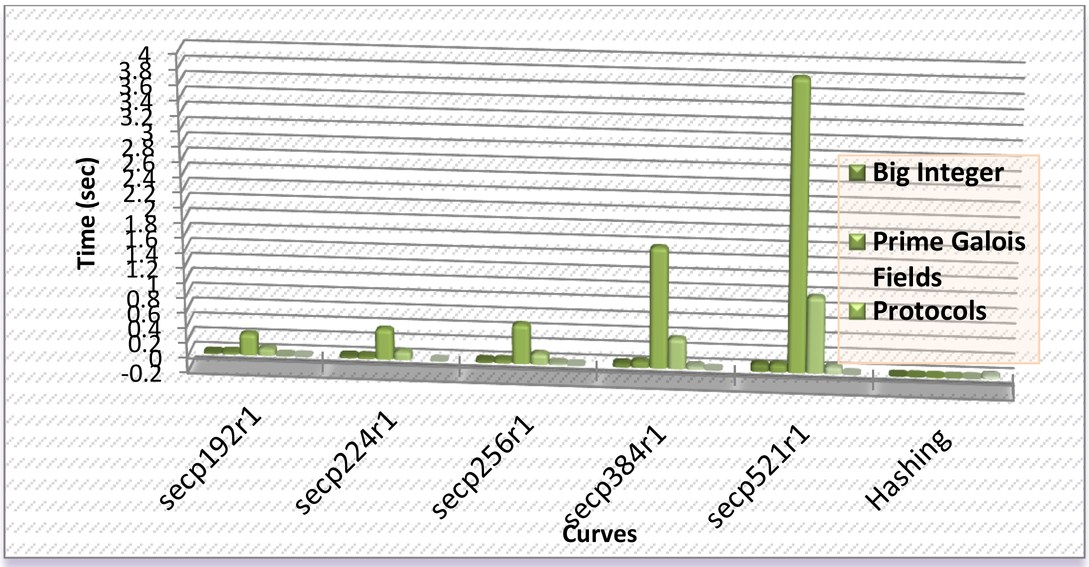


Fig 22: Performance in MAC OS

Now, for secp192r1 it took a total of 0.4 milliseconds to generate the key and create the signature. For secp224r1, secp256r1, secp384r1, secp521 it took 0.7, 0.87, 3.1 and 5.42 seconds respectively shown in Figure 22.

Data Sheet of all results found have been included in the appendix.

5.2 REMARKS AND FINAL OBSERVATIONS

From the above data we get RSA 1024 (14.1 milliseconds), LUCDIF 512 (7.6 milliseconds) are the algorithms that takes least time to perform the security routine in different tests. Based on our tests, algorithms like ECMQVC. Again, ECDHC over GF(p) 256 (25.83 milliseconds), MQV 1024 (15.35 milliseconds) , ECMQVC over GF(p) 233 (60.24 milliseconds) and ESIGN (1.94 milliseconds) are the algorithms that takes least time to perform the security routine in different tests. ESIGN is the fastest one here with a speed of more than 20 times than RSA but it has got weaknesses in the ephemeral key which could be a potential security threat. On the other hand, ECDHC also has good performance and smaller messages. MQV is also considerable but weaknesses like key-compromise impersonation, unknown key-share is present. Based on our tests, ECMQVC takes far more time to process than the other three algorithms.

CHAPTER 06

CONTRIBUTION OF PAPER

This chapter discusses the ways we think our work has contributed as well as dicusses options for fututre development

6.1 CONTRIBUTION

This thesis is trying to resolve the security issue by applying a number of security algorithms, for example, authenticated encryption schemes, AES, block ciphers message authentication codes, hash functions, elliptic curves etc. to compare and analyse their performances on an IoT platform, Raspberry Pi, an embedded system which is referred to as the black box and used as an IoT device in may work.[7] This thesis shows some optimal cases, presented a comparative analysis and hence helped identifying areas for improvement or constraint based selection of the algorithms worked on. This thesis suggests some important aspects covering the whole picture of algorithms that can be used in IoT sector, their drawbacks in respect to their performance and details analysis of the algorithms used.

LCOV and Genhtml is also used to find out the code coverage and accuracy remarks are based on the tests as well. This test data and test environment setup can be used in conjunction to IoT device development. It shortlists the algorithms which can be used for security purposes and can help making better product line by removing overheads and creating more secured and pleasant ioT experience. The detailed work can be understood in the section below.

6.2 FUTURE WORKS

There are lots of scopes for further research in IoT world where this thesis can provide crucial information on security issues for those devices. There are some limitations of this work.

Firstly, we were not able to test on various constraint environments. So in future, first work needs to be done in more number of constrained devices. This will help us develop more test cases or situations if necessary Secondly, node testing for targeted environments needs to be conducted in real time environment. Thirdly, more algorithms, current algorithm key size variations and input sets needs to be tested for all devices to make the researched field extensive and to cover all points.

If these works are done then the research can be a very useful work for review on the current security condition for IoT. As it stands, it has some scopes for further expansion.

CHAPTER 07

CONCLUSION

This chapter concludes the whole work.

7.1 CONCLUSION

Our work gives an overview of several algorithms run in different environments and more than one device and we hope it will help in various works such as easily choosing an algorithm or easily understanding performance by having a look at the graphical comparison. There is scope for further expansion by using more devices which could not always be accessed due to several restrictions.

The work gives an overview of several algorithms run in constrained environment. Once again, it is the first of its kind, to this work's knowledge, to use raspberry pi which is established as black box device and implemented security algorithms on it. This work focuses on different key factors in future of IoT world such as easily choosing an algorithm for a specific device with certain key size or easily understanding performance of a IoT device just by having a look at the graphical comparison given in this work. So, this work can be used as a guideline to design security structure for any specific IoT device. Overall it gives an easy overview for the field of further research or tools or algorithms to use for IoT security purpose.

REFERENCES

[1] Authenticated encryption. (2017, August 18). Retrieved August 20, 2017, from https://goo.gl/9z2S9F

[2] Cassar, G., Barnaghi, P., Wang, W., & Moessner, K. (2012, November). A hybrid semantic matchmaker for iot services. In Green Computing and Communications (GreenCom), 2012 IEEE International Conference on (pp. 210-216). IEEE.

[3] CCM Mode. (n.d.). Retrieved August 20, 2017, from https://goo.gl/xGmVZg

[4] Dinu, D., Corre, Y. L., Khovratovich, D., Perrin, L., Großschädl, J., & Biryukov, A. (2015). Triathlon of lightweight block ciphers for the internet of things. IACR ePrint archive.

[5] EAX Mode. (n.d.). Retrieved August 20, 2017, from https://goo.gl/mSWMQc

[6] GCM Mode. (n.d.). Retrieved August 20, 2017, from https://goo.gl/E7PNqB

[7] He, D., & Zeadally, S. (2015). An analysis of RFID authentication schemes for internet of things in healthcare environment using elliptic curve cryptography. IEEE Internet of Things Journal, 2(1), 72-83.

[8] Keoh, S. L., Kumar, S. S., & Tschofenig, H. (2014). Securing the internet of things: A standardization perspective. IEEE Internet of Things Journal, 1(3), 265-275.

[9] Lauter, K. (2004). The advantages of elliptic curve cryptography for wireless security. IEEE Wireless communications, 11(1), 62-67.

[10] Lee, J. Y., Lin, W. C., & Huang, Y. H. (2014, May). A lightweight authentication protocol for internet of things. In Next-Generation Electronics (ISNE), 2014 International Symposium on (pp. 1-2). IEEE.

[11] Maksimović, M., Vujović, V., Davidović, N., Milošević, V., & Perišić, B. (2014). Raspberry Pi as Internet of things hardware: performances and constraints. design issues, 3, 8.

[12] Miller, V. S. (1985, August). Use of elliptic curves in cryptography. In Conference on the Theory and Application of Cryptographic Techniques (pp. 417-426). Springer Berlin Heidelberg.

[13] OCB mode. (2017, August 17). Retrieved August 20, 2017, from https://goo.gl/c6mKia

[14] Shemaili, M. B., Yeun, C. Y., Mubarak, K., & Zemerly, M. J. (2012, December). A new lightweight hybrid cryptographic algorithm for the internet of things. In Internet Technology And Secured Transactions, 2012 International Conference for (pp. 87-92). IEEE.

[15] Shivraj, V. L., Rajan, M. A., Singh, M., & Balamuralidhar, P. (2015, February). One time password authentication scheme based on elliptic curves for internet of things (IoT). In Information Technology: Towards New Smart World (NSITNSW), 2015 5th National Symposium on (pp. 1-6). IEEE.

[16] Stream cipher. (2017, August 12). Retrieved August 20, 2017, from https://en.wikipedia.org/wiki/Stream_cipher

[17] Suo, H., Wan, J., Zou, C., & Liu, J. (2012, March). Security in the internet of things: a

review. In Computer Science and Electronics Engineering (ICCSEE), 2012 international conference on (Vol. 3, pp. 648-651). IEEE.

[18] Ukil, A., Sen, J., & Koilakonda, S. (2011, March). Embedded security for Internet of Things. In Emerging Trends and Applications in Computer Science (NCETACS), 2011 2nd National Conference on (pp. 1-6). IEEE\

APPENDIX

A. ***RESULTS USING*** **CRYPTO++®**

First library used is Crypto++® 6.0.0.

First Host: Raspbian Jessie

The first table has outputs of algorithms that here is considered of the primitive type.

	MiB/Second	**Microseconds to**
HMAC(SHA-1) (128-bit key)	43	3.64
CMAC(AES) (128-bit key)	10	3.391
ChaCha20 (256-bit key)	31	1.132
AES/CTR (256-bit key)	8	3.316
AES/CBC (256-bit key)	8	2.647
AES/OFB (128-bit key)	11	3.044
AES/CFB (128-bit key)	10	4.655
AES/ECB (128-bit key)	11	1.403
Threefish/CTR (1024-bit key)	4	3.2
DES-XEX3/CTR (192-bit key)	8	39.591
CAST-128/CTR (128-bit key)	15	4.344
XTEA/CTR (128-bit key)	8	2.378
Kalyna-128(256) (256-bit key)	5	7.288
AES/CCM (128-bit key)	5	4.514
AES/EAX (128-bit key)	5	9.671
VMAC(AES)-128 (128-bit key)	63	25.558

DMAC(AES) (128-bit key)	10	8.579
SipHash-2-4 (128-bit key)	38	0.24
Sosemanuk (128-bit key)	62	4.994
AES/CTR (128-bit key)	10	3.213
ARIA/CTR (128-bit key)	5	3.174
Camellia/CTR (128-bit key)	14	2.931
Twofish/CTR (128-bit key)	17	24.881
Serpent/CTR (128-bit key)	12	5.867
CAST-256/CTR (128-bit key)	12	8.84
SHACAL-2/CTR (128-bit key)	22	3.628
TEA/CTR (128-bit key)	10	2.354
Kalyna-128(128) (128-bit key)	7	6.082

Table 4: Output of Crypto++® in Raspbian Jessie

	mSec/OPR
RSA 1024 Encryption	0.55
RSA 1024 Decryption	13.56
LUC 1024 Encryption	0.63
LUC 1024 Decryption	21.96
RW 1024 Signature	14.07
RW 1024 Signature with precomputation	13.98
RW 1024 Verification	0.29
NR 1024 Signature	6.34
NR 1024 Signature with precomputation	3.5

NR 1024 Verification	7.39
NR 1024 Verification with precomputation	5.93
DSA 1024 Signature	6.19
DSA 1024 Signature with precomputation	3.47
DSA 1024 Verification	7.27
DSA 1024 Verification with precomputation	5.63
ESIGN 1023 Signature	1.43
ESIGN 1023 Verification	0.51
XTR-DH 171 Key-Pair Generation	5.54
XTR-DH 171 Key Agreement	11.07
XTR-DH 342 Key-Pair Generation	19.21
XTR-DH 342 Key Agreement	38.44
DH 1024 Key-Pair Generation	6.35
DH 1024 Key-Pair Generation w/ precomputation	6.85
DH 1024 Key Agreement	9.58
LUCDIF 1024 Key-Pair Generation	13.92
LUCDIF 1024 Key-Pair Generation w/ precomputation	13.89
LUCDIF 1024 Key Agreement	17.06
MQV 1024 Key-Pair Generation	6.22
MQV 1024 Key-Pair Generation w/ precomputation	3.43
MQV 1024 Key Agreement	11.92
ECIES over GF(p) 256 Encryption	31.31
ECIES over GF(p) 256 Encrypt w/ precomputation	20.13
ECIES over GF(p) 256 Decryption	21.32
ECDSA over GF(p) 256 Signature	15.87

ECDSA over GF(p) 256 Signature with precomputation	10.27
ECDSA over GF(p) 256 Verification	41.65
ECDSA over GF(p) 256 Verification with precomputation	17.08
ECDSA-RFC6979 over GF(p) 256 Signature	16.47
ECDSA-RFC6979 over GF(p) 256 Signature with precomputation	10.78
ECDSA-RFC6979 over GF(p) 256 Verification	41.43
ECDSA-RFC6979 over GF(p) 256 Verification with precomputation	17.68
ECGDSA over GF(p) 256 Signature	31.52
ECGDSA over GF(p) 256 Signature with precomputation	20.18
ECGDSA over GF(p) 256 Verification	41.74
ECGDSA over GF(p) 256 Verification with precomputation	17.47
ECDHC over GF(p) 256 Key-Pair Generation	15.71
ECDHC over GF(p) 256 Key-Pair Generation with precomputation	10.12
ECDHC over GF(p) 256 Key Agreement	15.76
ECMQVC over GF(p) 256 Key-Pair Generation	15.7
ECMQVC over GF(p) 256 Key-Pair Generation with precomputation	10.16
ECMQVC over GF(p) 256 Key Agreement	40.94
ECIES over GF(2^n) 233 Encryption with precomputation	27.15
ECIES over GF(2^n) 233 Decryption	53
ECDSA over GF(2^n) 233 Signature	46.9
ECDSA over GF(2^n) 233 Signature with precomputation	13.87
ECDSA over GF(2^n) 233 Verification	58.66
ECDSA over GF(2^n) 233 Verification with precomputation	23.26
ECDSA-RFC6979 over GF(2^n) 233 Signature	46.39
ECDSA-RFC6979 over GF(2^n) 233 Signature with precomputation	14

ECDSA-RFC6979 over GF(2^n) 233 Verification	57.02
ECDSA-RFC6979 over GF(2^n) 233 Verification with precomputation	23.87
ECGDSA over GF(2^n) 233 Signature	93.4
ECGDSA over GF(2^n) 233 Signature with precomputation	27.28
ECGDSA over GF(2^n) 233 Verification	58.49
ECGDSA over GF(2^n) 233 Verification with precomputation	23.85
ECDHC over GF(2^n) 233 Key-Pair Generation	46.55
ECDHC over GF(2^n) 233 Key-Pair Generation with precomputation	13.59
ECDHC over GF(2^n) 233 Key Agreement	46.7
ECMQVC over GF(2^n) 233 Key-Pair Generation	46.64
ECMQVC over GF(2^n) 233 Key-Pair Generation with precomputation	13.58
ECMQVC over GF(2^n) 233 Key Agreement	58.36
DLIES 1024 Encryption	12.5
DLIES 1024 Encryption with precomputation	13.46
DLIES 1024 Decryption	9.8
RSA 1024 Signature	13.56
RSA 1024 Verification	0.55
RSA 1024 Signature	13.56
RSA 1024 Verification	0.55
LUC 1024 Signature	22.08
LUC 1024 Verification	0.63
DLIES 2048 Encryption	56.15
DH 2048 Key-Pair Generation	28.44
DH 2048 Key-Pair Generation with precomputation	26.8
DH 2048 Key Agreement	36.52
LUCDIF 512 Key-Pair Generation	3.19

LUCDIF 512 Key-Pair Generation with precomputation	3.16
LUCDIF 512 Key Agreement	4.53
DLIES 1024 Encryption	12.5
DLIES 1024 Encryption with precomputation	13.46
DLIES 1024 Decryption	9.8
LUCELG 1024 Encryption	27.72
LUCELG 1024 Encryption with precomputation	27.67
LUCELG 1024 Decryption	17.39

Table 5: Output of Crypto++® in Raspbian Jessie

SECOND HOST: MAC OS

The first table has outputs of algorithms that here is considered of the primitive type.

	MiB/Second	**Microseconds to Setup Key and Initialization vector**
HMAC(SHA-1) (128-bit key)	404	281
CMAC(AES) (128-bit key)	551	183
ChaCha20 (256-bit key)	369	90
AES/CTR (256-bit key)	1607	221
AES/CBC (256-bit key)	405	193
AES/OFB (128-bit key)	533	200
AES/CFB (128-bit key)	548	233
AES/ECB (128-bit key)	2815	91
Threefish/CTR (1024-bit key)	262	223
DES-XEX3/CTR (192-bit key)	60	4990

CAST-128/CTR (128-bit key)	93	308
XTEA/CTR (128-bit key)	54	164
Kalyna-128(256) (256-bit key)	138	396
AES/CCM (128-bit key)	444	271
AES/EAX (128-bit key)	444	446
VMAC(AES)-128 (128-bit key)	3484	0.84
DMAC(AES) (128-bit key)	551	0.689
SipHash-2-4 (128-bit key)	1189	0.012
Sosemanuk (128-bit key)	988	0.519
AES/CTR (128-bit key)	2204	0.206
ARIA/CTR (128-bit key)	119	0.225
Camellia/CTR (128-bit key)	123	0.181
Twofish/CTR (128-bit key)	139	2.922
Serpent/CTR (128-bit key)	76	0.505
CAST-256/CTR (128-bit key)	91	1.148
SHACAL-2/CTR (128-bit key)	149	0.291
TEA/CTR (128-bit key)	58	0.165
Kalyna-128(128) (128-bit key)	184	0.332

Table 6: Output of Crypto++® in MAC OS

	mSec/OPR
RSA 1024 Encryption	0.03
RSA 1024 Decryption	0.44
LUC 1024 Encryption	0.03

LUC 1024 Decryption	0.84
RW 1024 Signature	0.5
RW 1024 Signature with precomputation	0.51
RW 1024 Verification	0.02
NR 1024 Signature	0.16
NR 1024 Signature with precomputation	0.15
NR 1024 Verification	0.18
NR 1024 Verification with precomputation	0.21
DSA 1024 Signature	0.16
DSA 1024 Signature with precomputation	0.15
DSA 1024 Verification	0.19
DSA 1024 Verification with precomputation	0.23
ESIGN 1023 Signature	0.08
ESIGN 1023 Verification	0.03
XTR-DH 171 Key-Pair Generation	0.22
XTR-DH 171 Key Agreement	0.43
XTR-DH 342 Key-Pair Generation	0.59
XTR-DH 342 Key Agreement	1.18
DH 1024 Key-Pair Generation	0.16
DH 1024 Key-Pair Generation w/ precomputation	0.3
DH 1024 Key Agreement	0.47
LUCDIF 1024 Key-Pair Generation	0.34
LUCDIF 1024 Key-Pair Generation w/ precomputation	0.34
LUCDIF 1024 Key Agreement	0.65
MQV 1024 Key-Pair Generation	0.15
MQV 1024 Key-Pair Generation w/ precomputation	0.14

MQV 1024 Key Agreement	0.31
ECIES over GF(p) 256 Encryption	1.74
ECIES over GF(p) 256 Encrypt w/ precomputation	1.34
ECIES over GF(p) 256 Decryption	1.22
ESIGN 1023 Verification	0.03
ECDSA over GF(p) 256 Signature	0.88
ECDSA over GF(p) 256 Signature with precomputation	0.69
ECDSA over GF(p) 256 Verification	2.75
ECDSA over GF(p) 256 Verification with precomputation	1.13
ECDSA-RFC6979 over GF(p) 256 Signature	0.91
ECDSA-RFC6979 over GF(p) 256 Signature with precomputation	0.66
ECDSA-RFC6979 over GF(p) 256 Verification	2.7
ECDSA-RFC6979 over GF(p) 256 Verification with precomputation	1.13
ECGDSA over GF(p) 256 Signature	1.73
ECGDSA over GF(p) 256 Signature with precomputation	1.33
ECGDSA over GF(p) 256 Verification	2.69
ECGDSA over GF(p) 256 Verification with precomputation	1.13
ECDHC over GF(p) 256 Key-Pair Generation	0.87
ECDHC over GF(p) 256 Key-Pair Generation with precomputation	0.67
ECDHC over GF(p) 256 Key Agreement	0.86
ECMQVC over GF(p) 256 Key-Pair Generation	0.87
ECMQVC over GF(p) 256 Key-Pair Generation with precomputation	0.7
ECMQVC over GF(p) 256 Key Agreement	2.71
ECIES over GF(2^n) 233 Encryption	7.63
ECIES over GF(2^n) 233 Encryption with precomputation	2.17
ECIES over GF(2^n) 233 Decryption	4.33

ECDSA over GF(2^n) 233 Signature	3.84
ECDSA over GF(2^n) 233 Signature with precomputation	1.11
ECDSA over GF(2^n) 233 Verification	4.78
ECDSA over GF(2^n) 233 Verification with precomputation	1.86
ECDSA-RFC6979 over GF(2^n) 233 Signature	3.9
ECDSA-RFC6979 over GF(2^n) 233 Signature with precomputation	1.17
ECDSA-RFC6979 over GF(2^n) 233 Verification	4.74
ECDSA-RFC6979 over GF(2^n) 233 Verification with precomputation	1.91
ECGDSA over GF(2^n) 233 Signature	7.66
ECGDSA over GF(2^n) 233 Signature with precomputation	2.18
ECGDSA over GF(2^n) 233 Verification	4.72
ECGDSA over GF(2^n) 233 Verification with precomputation	1.88
ECDHC over GF(2^n) 233 Key-Pair Generation	3.84
ECDHC over GF(2^n) 233 Key-Pair Generation with precomputation	1.09
ECDHC over GF(2^n) 233 Key Agreement	3.91
ECMQVC over GF(2^n) 233 Key-Pair Generation	3.83
ECMQVC over GF(2^n) 233 Key-Pair Generation with precomputation	1.09
ECMQVC over GF(2^n) 233 Key Agreement	4.74
DLIES 1024 Encryption	0.3
DLIES 1024 Encryption with precomputation	0.58
DLIES 1024 Decryption	0.47
RSA 1024 Signature	13.56
RSA 1024 Verification	0.55
RSA 1024 Encryption	0.03
RSA 1024 Decryption	0.44
LUC 1024 Signature	0.87

LUC 1024 Verification	0.03
DLIES 2048 Encryption	1.49
DLIES 2048 Decryption	1.41
DH 2048 Key-Pair Generation	0.76
DH 2048 Key-Pair Generation with precomputation	0.87
DH 2048 Key Agreement	1.41
LUCDIF 512 Key-Pair Generation	0.1
LUCDIF 512 Key-Pair Generation with precomputation	0.1
LUCDIF 512 Key Agreement	0.24
DLIES 1024 Encryption	0.3
DLIES 1024 Encryption with precomputation	0.58
DLIES 1024 Decryption	0.47
LUCELG 1024 Encryption	0.68
LUCELG 1024 Encryption with precomputation	0.68

Table 7: Output of Crypto++® in MAC OS

B. ***RESULTS USING FLECC_IN_C.***

First Host: Raspbian Jessie

	Big Integer	**Prime Galois**	**Protocols**	**ECC**	**RFC**	**Hashing**
secp192r1	0.02	0.03	0.28	0.09	-0.01	-0.01

secp224r1	0.03	0.03	0.4	0.11	-0.01	-0.01
secp256r1	0.03	0.04	0.52	0.14	0.02	-0.01
secp384r1	0.05	0.08	1.6	0.4	0.04	-0.01
secp521r1	0.1	0.1	3.85	1.01	0.09	-0.01
Hashing	-0.01	-0.01	-0.01	-0.01	-0.01	0.04

Table 8: Output of *FLECC in Raspbian Jessie*

Second Host: MAC OS

	Big Integer	**Prime Galois**	**Protocols**	**ECC**	**RFC**	**Hashing**
secp192r1	0.02	0.03	0.28	0.09	-0.01	-0.01
secp224r1	0.03	0.03	0.4	0.11	-0.01	-0.01
secp256r1	0.03	0.04	0.52	0.14	0.02	-0.01
secp384r1	0.05	0.08	1.6	0.4	0.04	-0.01
secp521r1	0.1	0.1	3.85	1.01	0.09	-0.01
Hashing	-0.01	-0.01	-0.01	-0.01	-0.01	0.04

Table 9: Output of *FLECC in MAC OS*