Secure Smart Grid Infrastructure

CM2303 One Semester Individual Project

Final Report

Author: Fraser Orr

Supervisor: Neetesh Saxena

Moderator: Hantao Liu

Abstract

This project will aim to investigate the communication between Intelligent Electronic Devices (IEDs) alongside the security risks associated with them and finally the infrastructure in which they are used. Through research and testing, a prototype will be created to model the proposed encryption solutions for various communication types found in a smart grid system. The challenge of this project is to find an elegant solution that will provide security without hindering the operation of the smart grid.

Acknowledgements

My thanks go to my supervisor Neetesh Saxena for his guidance on the project and his expertise.

Table of Contents

Table of Figures

Chapter 1 Introduction

The power grid is a critical infrastructure and therefore, it would provide a tempting target for sophisticated, intelligent and well-equipped attackers. As smart grids slowly become more widely used, it is important to understand the protocols that are used for communication, their constraints and vulnerabilities. There is currently concern over the vulnerability of critical infrastructures and what resulting damage an attack on such infrastructures would cause.

This is a relatively new problem as legacy utility communication has previously been immune to security threats, most of the communication occurred through private networks and its communication protocols were secured through a security via obscurity principle. However, due to the standardisation of protocols this principle is no longer an effective security measure.

Communication between IED’s is handled by IEC 61850. One of the main objectives of the IEC 61850 communications standard is to provide a set of standard model structures regarding data and a set of rules defining how this data should be exchanged. IEDs from different manufacturers that comply with these model definitions can then understand, communicate and interact with one another [2].

Chapter 2 Background

There are three main types of communication this literature will investigate are used currently, they are Manufacturing Message Specification (MMS), Generic Object-Oriented Substation Events (GOOSE) and Sampled Measured Values (SMV) [3]. The MMS is an ISO 9506 standard that is used to transfer real-time process data and control information between the network devices, such as IED and the HMI (Human Machine Interface) [3] – it follows a client-server model. GOOSE is an event driven message therefore it follows a publisher-subscriber model for asynchronous multicast communication. SMV also follows a publisher-subscriber model and is used for asynchronous multicast communication with voltage and current values [3]. The multicast messages use MAC addresses for the communication via bridge routing and does not used IP-based routing.

2.1 Publisher - Subscriber Model

The Publisher-Subscriber Model is a messaging pattern where the publishers (senders) publish messages into the communication infrastructure and the subscribers (receivers) express interest in a particular message category. This is very different from the synchronous request-response model and is a much more scalable solution due to no limitations surrounding centralized data. Within the IEC 61850 framework, GOOSE messages and messages transmitting sampled values (SV) are the main types of messages that require indirect asynchronous delivery. The Publisher-Subscriber Model can take advantage of Multicast messaging, this allows the sending of a single copy which will be replicated and passed on throughout routers and forwarded to subscribers that have previously signalled interest. The communication infrastructure is responsible for delivery of the messages and maintains the subscription information.

The Publisher Subscriber architecture is vulnerable to man-in-the-middle (MITM), replay and impersonation attacks. The system also suffers from issues including publisher (sender) authentication, subscriber (receivers) authorization and data integrity [3].

2.2 MMS

The MMS (Manufacturing Message Specification) is an ISO 9506 standard that is used to transfer real-time process data and control information between the network devices, such as an IED and the HMI application running on a PC. It follows a more traditional Client-Server model for communication. MMS has the slowest time requirements compared to the publisher/subscriber methods discussed below with a wider range of data. Therefore, it has the most flexibility when using an encryption algorithm as it does not need to be as light weight.

2.3 SMV

IEC 61850-9-2 defines the Sampled Measured Values traffic which carries voltage and digital current samples. SV protocol uses OSI model Layer 2 for communication identified by MAC address and the identifier in the message body [9]. SMV also follows a Publisher-Subscriber model utilising multicast messaging.

2.4 GOOSE

Generic Object-Oriented Substation Event (GOOSE) supports the exchange of a wide range of possible common data organized by a DATA-SET. GOOSE messages are used to replace the hard-wired control signal exchange between IEDs for interlocking, protection purposes, sensitive missions, time criticality and highly reliability. GOOSE messages are exchanged at the datalink layer using the multicast functionality of ethernet cables. When triggered by a preconfigured event, an IED sends the GOOSE message containing values for the variables that need to be communicated, such as carry monitoring and control functions, tripping and interlocking information. Since these are multicast messages, there is no acknowledgement mechanism. A short GOOSE message that is handling just one digital status information in its dataset has an approximate size of 124 bytes. The actual size depends on various configured parameters in the GOOSE control block such as GoID, name of dataset and the reference object of the GOOSE control block. Typical size of GOOSE messages are between 92 bytes to 250 bytes [9].

2.5 Multicast Messaging

Both GOOSE and SMV utilise multicast messaging. Multicast address filters limit the distribution of the multicast address frames (GOOSE and SMV) to the subscriber ports that require them, rather than to every port [9]. This reduces total traffic on the network and is achieved by the management layer. Further address grouping can be done by splitting groups of subscribers that all receive information from the same sources into MAC groups and assign them a unique multicast address assigned to the flows of the MAC group.

Chapter 3 Specification and Design

3.1 Overview

This section will cover the design and specification used to create a useful comparative tool for different encryption methods. This section will show how the user should interact with the tool and what is essential to include within the functionality.

3.2 Project Planning

This segment will provide a summary of the tasks due for completion in order to finish this project on time.

Before the main project can begin, a detailed list of specifications is needed for the requirements of the three message types I have selected; GOOSE, MMS and SMV. This will determine the selection of encryption algorithms and will be used to design the testing networks.

The project has two main parts, both are required for quantitative results to be analysed. Part 1 is the successful implementation of the encryption algorithms and a GUI tool to ease their testing. The second part involves gathering the results to perform an analysis and comparison. Due to the importance of the algorithms, I have allocated them a large proportion of time within my Gantt chart (Appendix A) and each encryption algorithm was allotted a week to be tested and incorporated.

3.3 Methodologies Used

I will discuss the various approaches to solving the task and which specific methodologies I incorporated.

3.3.1 Prototype Approach

I concluded the optimum approach for this was the iterative approach, this requires quick design and building followed by a user evaluation / prototype revision loop. This model of approach would allow me to develop multiple aspects of the project in tandem and refine this once important parts of the tool were emplaced. This approach is also required as, depending on the results from the first attempt, the scope of the project may change if the encryption methods are not meeting the message type requirements. The first prototype should show that they can correctly encrypt and decrypt a message sent to it with no use of a GUI. The next stage should begin to look at the times around different stages and increase the number of tests done for an average set of results. The third step would be a basic GUI to allow the display of results, finally, a revision of the GUI to a finished level would be completed.

3.4 Specification

Within this section I will be analysing the specification for message types I will be supporting and my design approach to the problem.

3.4.1 Message Specification

When analysing the message types being used, GOOSE, SMV and MMS, it is important to distinguish them based on their different design criteria and topologies. This means that one design approach may not cover all the criteria of all message types in all scenarios.

ID	Message Type	Requirement Name	Criteria	Comment
R1	GOOSE, SMV	Time Constraint	The total time from Publisher Encryption to the message being decrypted by the subscriber should take no longer than 3ms	This is the defining requirement for the algorithm as the time complexity will dictate what solutions should discussed
R2	GOOSE, SMV	Model	The system should support multicast communication for Publisher-Subscriber model	This is to emulate the real-world communication the Smart Grids use to communicate
R3	MMS	Time Constraint	Time is less critical with total time ranging from 100-500 depending if tis a low/medium or command message	These are not very quick time conditions so a wide range of algorithms should be applicable
R3	MMS	Model	Follows a client-server model	This is a more standard model to implement

Here we can see that the main considerations for each message type are the time requirements and the model that the message type supports, this shows that the problem is split into the GOOSE/SMV side and the MMS side as each of have different consideratiosn.

3.4.2 Prototype and Algorithm Specifications

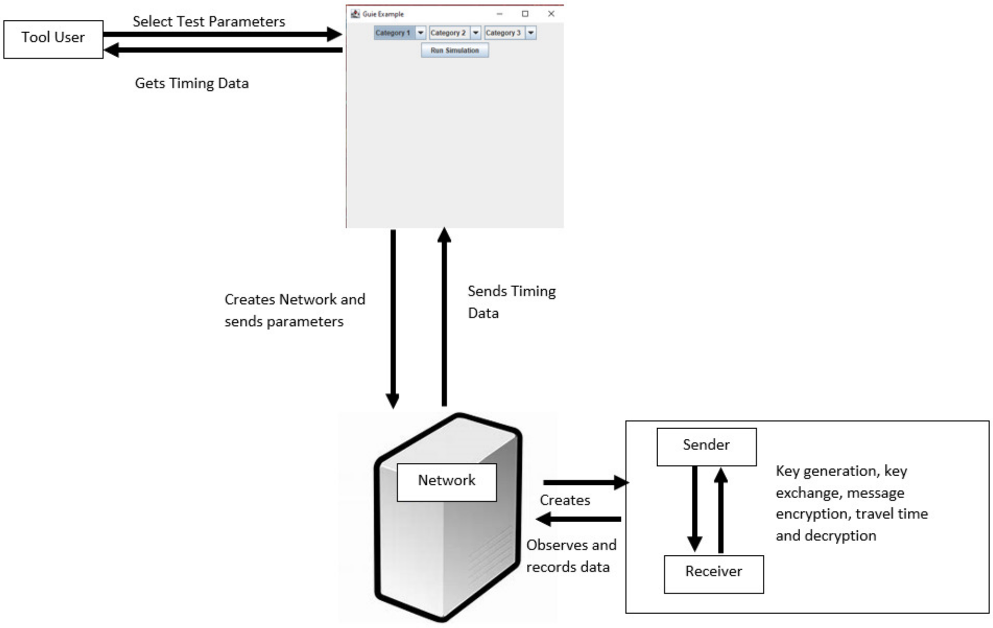
The messages take part on Intelligent Electronic Devices (IEDs) and such, to make the model practical other design considerations should be considered when examining the system for any encryption algorithm introduced. The non-functional requirements for the prototype and algorithm are detailed below.

ID	Requirement Name	Priority	Criteria	Comment
NF1	The algorithms must be light weight	High	In order to be deployed to IEDs the algorithms must be able to run in a limited environment	This directly ties into how quickly the algorithm runs with most fast encryption's methods based around lightweight methods
NF2	Time constraints	High	The specification for message types must be met	The constraints for each message type as mentioned previously must be met by any implementation
NF3	Message Integrity	High	The message sent on any model must be able to be decoded and read by the recipient	All encryption must be able to be decrypted by the intended recipient
NF4	Attack prevention	High	All encryption algorithms used must not be susceptible to any known attack	The implementation must be cryptographically secure from known attacks such as Man-in-the-middle or replay attacks
F1	User Comparison	High	It should be easy for the user to compare different algorithms with different message types.	Using the tool, it should be easy to look at results of

3.5 System Design

3.5.1 System Model

I will be using a system model and class diagram to show how the tool will interact. It will show the outline of how the system should work as well as presenting an idea of architecture.



From this we can see the basic flow of the network for the system model alongside the data exchange. The sender and receiver are treated as two nodes in a network with communication between the nodes handled by the network class to simulate communication over a network. We can consider the network to be unsecure so all communication between the nodes should be encrypted. For the encryption, the nodes will have access to the encryption methods as they should handle the entire process with no help other than transmission of the ciphertext and keys, which the network would handle.

I have included in appendix D an in-depth class diagram exhibiting the class relationships.

The SpeedTester is the class name of the GUI which uses action listener and is an extension of the awt frame class. It contains all the UI elements and handlers and is used to setup the network when the test is started via the start button.

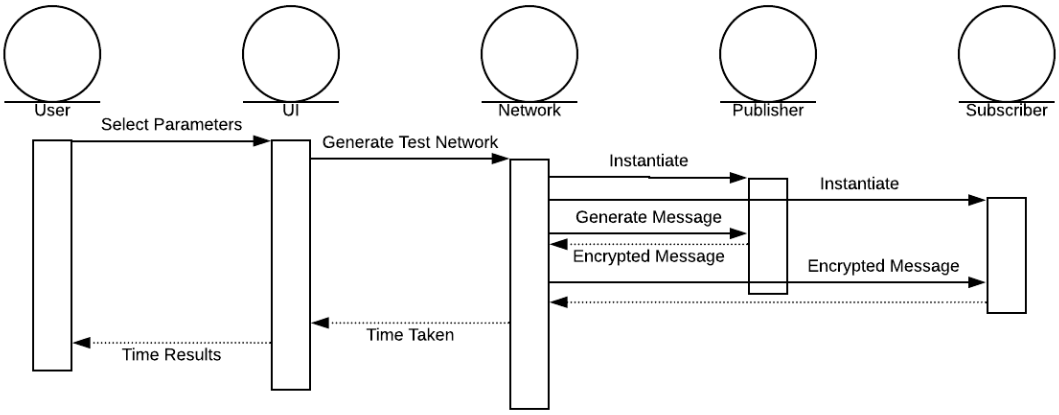
The Network class stores all the sent network parameters and the timing data generated from each test. It is also responsible for the instantiation of the Sender and Receiver class which act as our IED nodes in the network architecture. It has a unique setup and runtime for each encryption type.

The Sender class, also doubling as our Publisher in the relevant network architecture, contains methods to instantiate each encryption class and a method for using the encryption class to convert a byte[] input into an encrypted form. For RSA it requires the public key to be sent to it for encryption. It also has methods to retrieve the secret key and if needed the initialise vector IV.

The receiver class uses methods to instantiate each encryption type but uses different constructors, as the secret key should already be known. Therefore, the cipher should be made using this to achieve original plaintext. For RSA the receiver class contains the key generation for the private-public key pair.

Each encryption method has its own class that includes two constructors, one for a new cipher and one to generate an existing cipher based on the secret key and/or initial vector. It also includes the encoding and decoding methods.

3.5.2 Detailed Design



The above sequence diagram shows the full user process from selecting the input parameters to the returning of the timing results. It details the user and the main classes, not including the encryption classes that the publisher and subscriber employ for the encryption/decryption process. It shows how the network is being used to instantiate the publisher and subscriber nodes and how it acts as the main data stream. This also highlights communication flow around the classes. For the timing results, the time will start from the message generation and stop when the encrypted message has been decrypted by the subscriber.

Chapter 4 Solution and Implementation

4.1 Overview

Within this section I will be discussing the implementation and approach of the different algorithms, languages and language specific modules I will be using in my solution. I will explain how the algorithms work as the majority of the algorithms are computed through the java crypto modules. A clear understanding of the encryption methods is still required to justify their use and limitations.

I will not be aiming to directly replicate an exact model of a smart grid setup with all the protocols included, but rather I will be developing a took to test encryption algorithms so that they can be evaluated for a smart grid application.

4.2 Nodes

For the publisher subscribers, you can construct two separate classes, sender and receiver, as they don’t have to fill both roles on the network. However, this means there is limited functionality to handle peer-peer communication between the “IEDS” on this test network. When used for modelling a Publisher Subscriber model the sender class acts as the publisher and is used to generate the secret key. The main reason for having separate classes for nodes and not using a superclass is that in the Publisher Subscriber, the publisher plays a very different role to the subscriber nodes and should not be treated as a similar type. The senders need to able to generate Keys for AES and ChaCHa20 and the receiver would need the ability to generate the RSA keys. Both nodes would need to be able to perform, encrypt or decrypt operations given the secret key and any other algorithm specific required input.

4.3 Network Class

For testing the algorithms, a testing class needed to be developed to simulate a test network that the GUI could run to get the results. This required a sender object and a receiver object that could either work as a simple client-server model for simple peer-peer communication or, work as a publisher-subscriber model for the GOOSE and SMV communication types. The network would then act as the data stream maintaining subscriber lists in the case of the publisher subscriber topology and forward messages to the designated locations. It would also handle the setup for the algorithms such as the secure transmission of the secret keys for the symmetric communication.

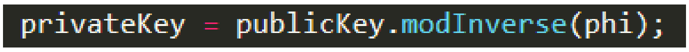
4.4 Protecting Communicated Information – Confidentiality

Encryption algorithms can be split into two kinds, asymmetric and symmetric ciphers, considering the time consideration I will be mostly looking at symmetric ciphers due to their quicker computation times. However, I will include an RSA example in the work due to how secure it is and to use this as a baseline comparison for other algorithms. All the encryption algorithms will need to take a byte input of variable size for encryption and decryption.

4.4.1 Asymmetric Communication RSA

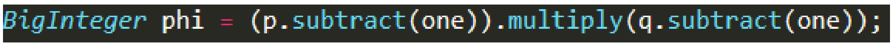
For complete security RSA is commonly used for secure communication. It is an asymmetric encryption that has two distinct keys, a public key for encryption and a private key which is used to decrypt. The security is based on the difficulty in factorising the product of two large prime numbers. The algorithm can be split into for major sections: key generation, key sharing, encryption, and decryption.

Key generation for RSA first revolves around selecting two distinct prime numbers p,q which should be chosen at random and have similar, but different lengths, to increase the difficulty of factorising. The next step is computing n = pq, n is the modulus for both the public and private keys, its length is the key length. N is released as part of the public key. For the private key it is the lcm of (p-1)(q-1) which is calculated as following. Code Extract from RSA class.



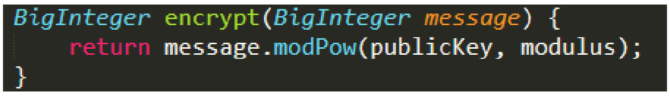
```
privateKey = publicKey.modInverse(phi);
```

With phi being:



```
BigInteger phi = (p.subtract(one)).multiply(q.subtract(one));
```

For encryption and decryption I have used javas inbuilt modPow function using the respective public or private keys, RSA has the added benefit of being able to transmit the public key and modulus as part of the public key as plain text without compromising the security of the communication. Encryption example code is below.



```
BigInteger encrypt(BigInteger message) {
    return message.modPow(publicKey, modulus);
}
```

4.4.2 Block Cipher AES

One algorithm I focused on in particular is the widely used block cipher AES (Advanced Encryption Standard) which replaced the Data encryption Standard (DES) used by the American government. It uses a fixed block size of 128 bits and a key size of 128,192 or 256 bits. AES is a great block cipher to use in the comparison due to its efficient implementation in both software and hardware. The key size used in AES dictates the number of rounds that the plaintext goes through. Step 1 is a key expansion that derives a round key from the cipher key. AES requires a separate 128-bit round key block for each round plus a final one. Next is an initial round key addition which combines (using bitwise XOR) each byte of the states with a byte from the round key. For the next rounds (depending on key size), run as follows.

1. a non-linear substitution step where each byte is replaced with another according to a lookup table.
2. a transposition step where the last three rows of the stat are shifted cyclically a certain number of steps
3. a linear mixing operation which operates on the columns of the state, combining the four bytes in each column.
4. adding the round key where the subkey is combined with the state

Then there is a final round which involves the same steps as the previous rounds but with Step 3, the linear mixing operation, removed.

In terms of security, despite a lot of research into AES there are no practical attacks on AES with some side channel attacks on the implementation of AES which are easy to avoid.

4.4.3 Stream Cipher ChaCha20

Salsa20 and ChaCha20 are a close family of stream ciphers with ChaCha variant aiming to increase the diffusion per round and reduce the number of rounds needed for security. The Salsa20 encryption function is a long chain of three operations on 32-bit words

- a 32-bit addition, producing the sum a+b mod 2^32 of two 32-bit words a, b.
- a 32-bit exclusive-or, producing the xor a, b of two 32-bit words a, b; and
- constant-distance 32-bit rotation, producing the rotation a <<< b of a 32-bit word a by b bits to the left where b is a constant.

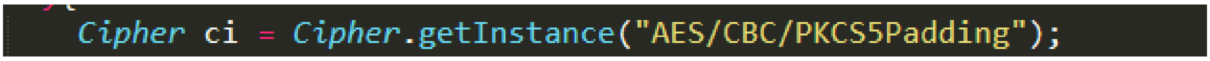
Salsa20 expands a 256-bit key and a 64-bit nonce into a 2^70-byte stream. Encryption is achieved by xor'ing the b-byte plaintext with the first b bytes of the stream and discarding the rest. For decryption it follows the same process. The stream is generated in 64-byte (512-bit) blocks. Each block is an independent hash of the key, the nonce and a 64-bit block number; there is no chaining from one block to the next. This means the output block can therefore be accessed randomly and any number of blocks can be computed in parallel.

Chacha follows the same base design principles as Salsa20 but has an increased diffusion per round allowing a smaller minimum number of secure rounds for ChaCHa. The extra diffusion does not come at the expense of extra operations as ChaCha round has 16 additions, 16 xors and 16 constant distance rotations of 32-bit words, just like a Slasa20 round. The main speed and diffusion differences comes from the quarter round where each word is updated twice and the word matrix, where it is built with attacker controlled inputs words on the bottom. ChaCha sweeps the matrix through rows in the same order every round.

4.5 Java Modules

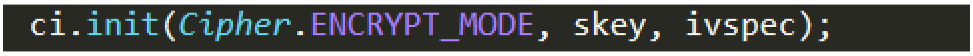
I will be using Java as the implementation language for its wide range of importable modules such as the java crypto packages for cryptographic operations for AES and ChaCha. Java crypto specific modules that are most useful are the Cipher and key generator class. For the RSA algorithm, which was used more as a comparison tool to show the difference to these quicker encryption methods, the java security module was used for the secure random feature.

The Javax.crypto.cipher class provides the functionality of a cryptographic cipher. In order to create a Cipher object, the class calls the Cipher's getInstance() method, and passes the name of the requested transformation to it. A transformation is a string that describes the operation (or set of operations) to be performed. It includes the name of a cryptographic algorithm (e.g., AES), and may be followed by a feedback mode and padding scheme. In my case it includes the mode type of AES and a padding scheme for extra security.



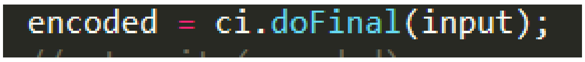
```
    Cipher ci = Cipher.getInstance("AES/CBC/PKCS5Padding");
```

The cipher is then initialised with the mode (either encrypt or decrypt), the secret key and an initial vector.



```
ci.init(Cipher.ENCRYPT_MODE, skey, ivspec);
```

The final function of the cipher class used is the final encryption of the plain text using the doFinal() function which will encrypt or decrypt a byte form input so if you are sending a string it must be converted into a bytes first.

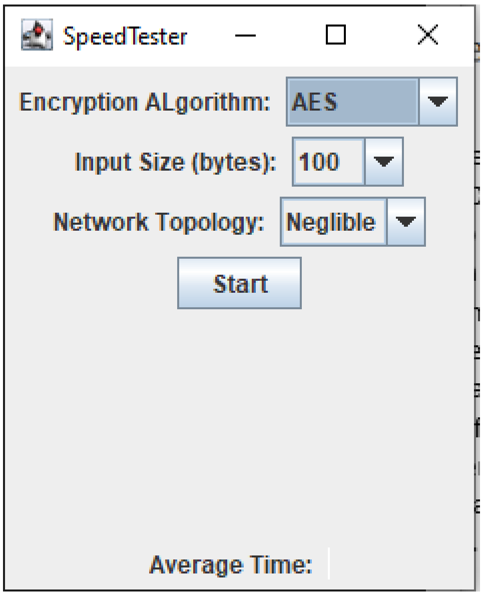


```
encoded = ci.doFinal(input);
```

The other crypto modules used are the Key generator class which is a re-usable key generation object. It can only be used for symmetric secret key generation. There are two ways to generate a key; in an algorithm-independent manner and in an algorithm-specific manner. The only difference between the two is the initialization of the object.

4.6 GUI

For the GUI I have used the java swing and awt modules which are common graphical interfaces. I will use the combo boxes to select pre-selected options for which to test the encryption methods. There should be a combo box for Algorithm, Input size and network topology which will add a latency time between the encryption and decryption to simulate travel time on a network. There should also be an output area to collect readings using the GUI.



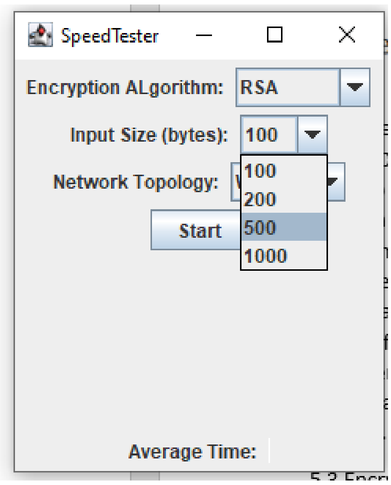
Encryption ALgorithm: AES

Input Size (bytes): 100

Network Topology: Neglible

Average Time:

This is what the GUI will look like when it is ran, it displays the choices available to you on top and indicates where the readings will be displayed on the bottom.



Encryption ALgorithm: RSA

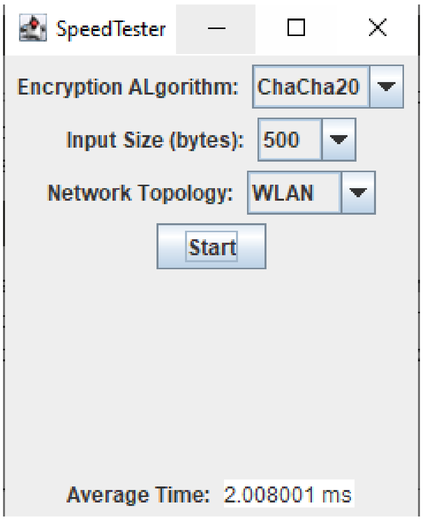
Input Size (bytes): 100

Network Topology:

Average Time:

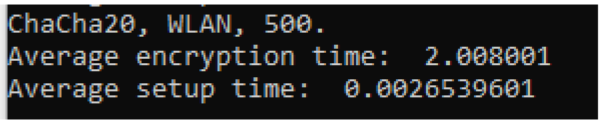
This is what the drop downs look like for selecting a variety of choices of the tool. When all the choices have been selected, the users should press the button and an action listener will send the choices into the network start function which will setup the test network to run on the indicated parameters. After testing, the main result should be returned this is the average time. However there are other results which can be displayed to the command line that could be displayed in the tool, should the user want advanced reading such as setup time or a break-down of the total time.

This is what a run simulation should look like with the output displayed at the bottom with the units. All units taken are in milliseconds as that is what the specification units of a smart grid are given in. This example shows ChaCha20, 20 rounds, being used with an input size of 500 bytes using a WLAN which gives a network latency of 2 ms.



Encryption ALgorithm: ChaCha20
Input Size (bytes): 500
Network Topology: WLAN
Average Time: 2.008001 ms

We can see an example of an extra console output, of the average time taken for key generation which can be an important consideration for results analysis.



```
ChaCha20, WLAN, 500.
Average encryption time:  2.008001
Average setup time:  0.0026539601
```

Chapter 5 Results and Evaluation

5.1 Overview

Within this segment I will be discussing the results I have gathered from the testing tool and evaluate the potential usefulness of these results with regards to the implementation into a smart grid infrastructure. I have tested all three

algorithms over various message sizes to ensure a fair result was reached. When discussing the results, I will be grouping them suitably in order to draw comparison and attention to specific points; I will not just be discussing the encryption and decryption times.

5.2 Cycles per Byte

To gain an improved understanding of the timing, we can use the data to work out other aspects of the performance of the algorithm’s software implementation. Due to the wide variety of variables that can affect the time, it is not as reliable as an evidence source. We can use readings to focus on the cycles per byte performance of the algorithm.

$$\text{cycles per byte} = \frac{\text{cycles per second}}{\text{bytes per second}}$$

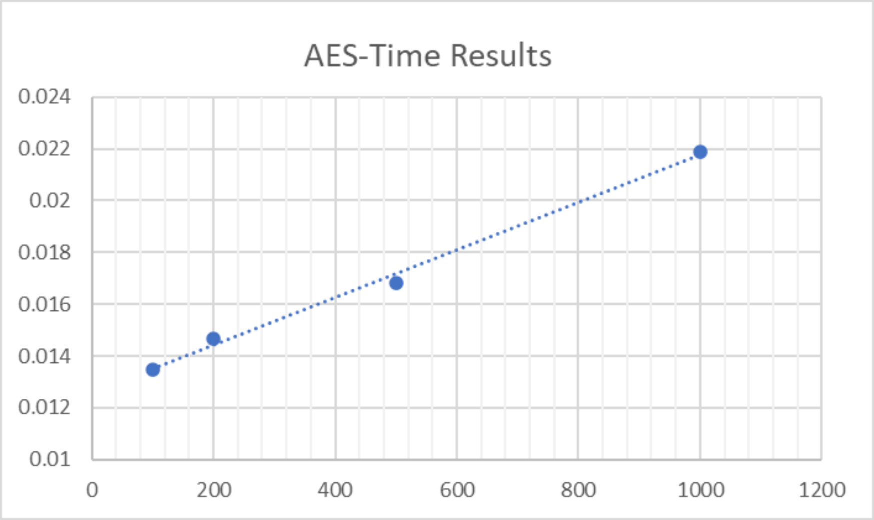
The bytes per second is very easy to calculate as it is the size of the packet being sent in this case 100, 200, 500 or 100-bytes and we have recorded the time taken in each of these algorithms. We can also split this into time taken to encrypt and time taken to decrypt to work out a cycles per byte for each specific process. For the ease of simplicity, we are using symmetric algorithms, I took half of the total time to calculate the cycles per byte for either encrypting or decrypting. The cycles per byte is machine dependant and can be widely affected. Some processors also are optimised to perform certain algorithms at a quicker pace. The tool am I using is running on a personal machine with a CPU speed of 3.8 GHz, using the process manager in task manager during the algorithm run time I can see what percentage of the total processor is being used by the application. In most cases, this was a single digit percentage. I can use the percent value to view the cycles per second being used for the encryption process.

5.3 Encryption Times

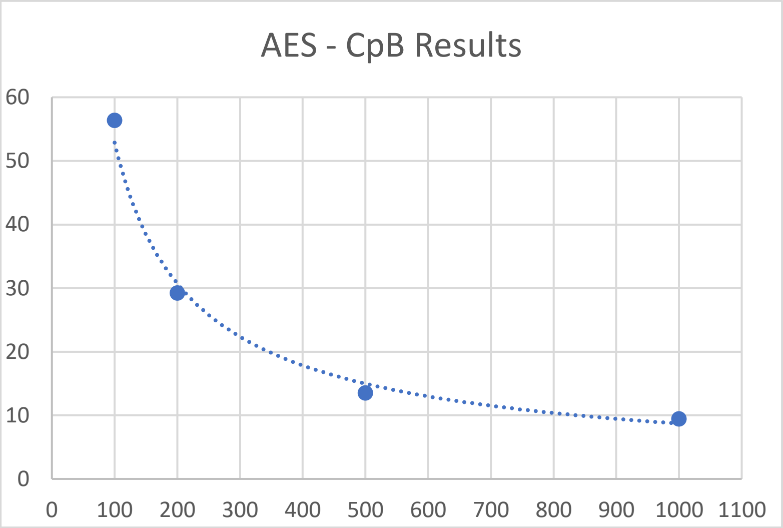
Most results labelled under time taken was the time for the plaintext to be encrypted, sent, network delay, received, decrypted. There will be other periods where I will be discussing additional times such as, setup times and key generation times and how this affects the usefulness of results.

5.3.1 AES

The AES encryption times using CBC mode with a padding were taken using 100, 200, 500- and 1000-byte message sizes and ran on a 1 to 1 connection. The results were taken 10,000 times with a different key generated for each connection.



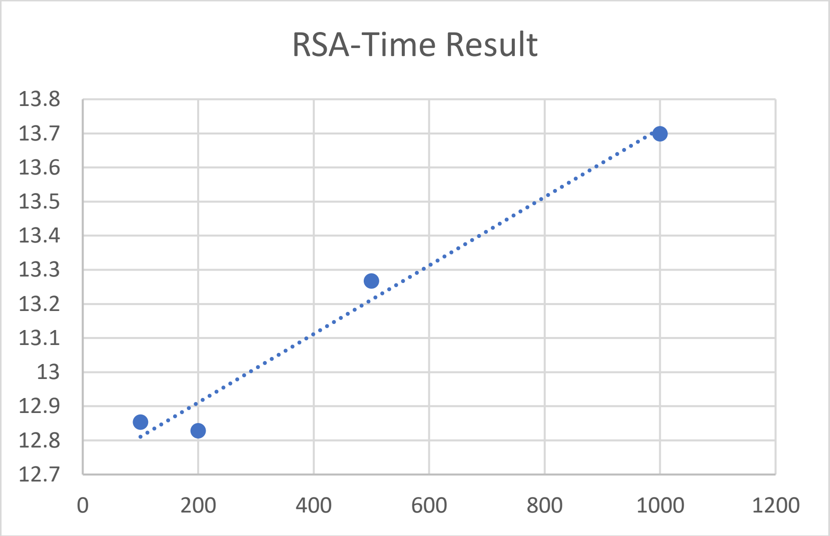
For the publisher-subscriber message byte ranges of 150-250 bytes, on the local machine I was using, they had a total time of 0.0147ms for the 200 bytes.



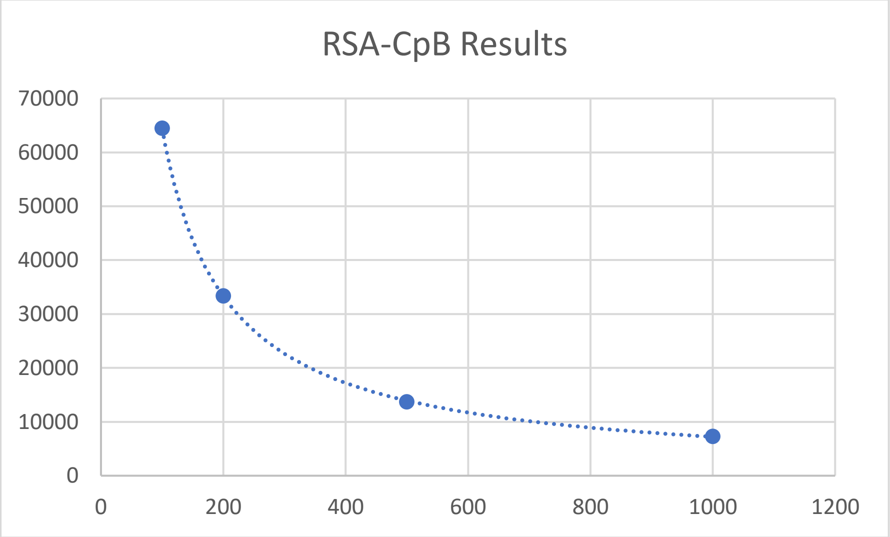
According to benchmark studies [10] on a test platform using 2.9e+009Hz using CBC mode 128 bit key which is the same encryption method, AES reached a CpS of 28. This is comparable to the results of 200 byte input that ran at an estimated of 29 cycles per byte 7.8e+008HZ. Interestingly, as the byte input increases the cycles per bytes drops. This is because the gradient of the trendline on the AES time results is very small so despite large increase to the byte size it currently suggests little increase to the time taken to perform the operation. AES is also widely optimized for in most current hardware, meaning in desktops, some AES configuration could still be a faster alternative.

5.3.2 RSA

The RSA used had 2048-bit big integer keys as this is deemed totally secure for now and the foreseeable future with some of the smaller keys, though quicker, are still deemed to pose a possible security risk. It takes 22ms to generate the private and public key and the public key needs to be sent to the sender. In the case of a Publisher-Subscriber model where it is a one to many ratio, the publisher would have to store a list of all the public keys of the subscribers subscribed in a subscriber list. This takes additional setup time and care and would require more validation for new subscribers.

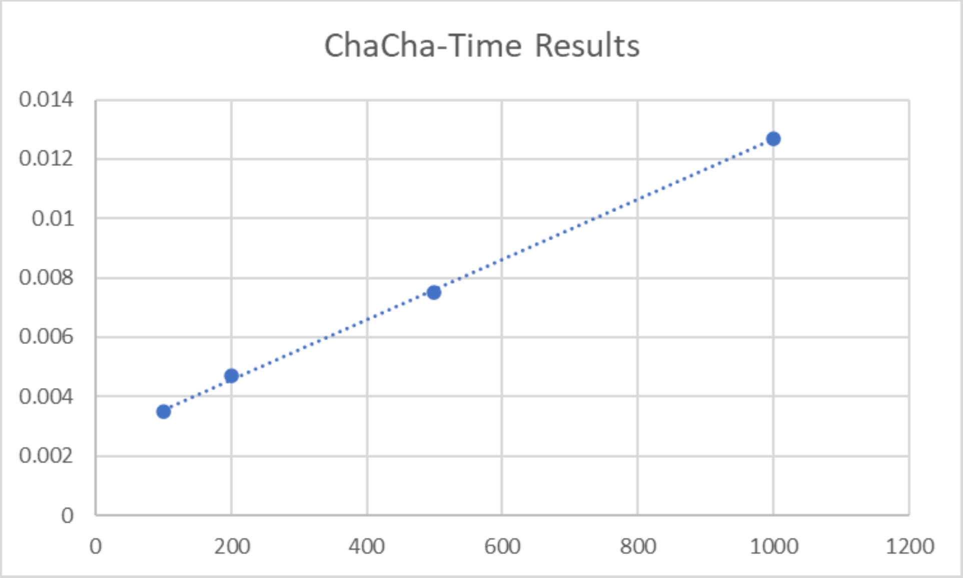


Even on a powerful machine the run time of RSA is much slower even with no network latency. The smallest packet size of RSA was 100 bytes and took an average time of just under 12.9 ms.

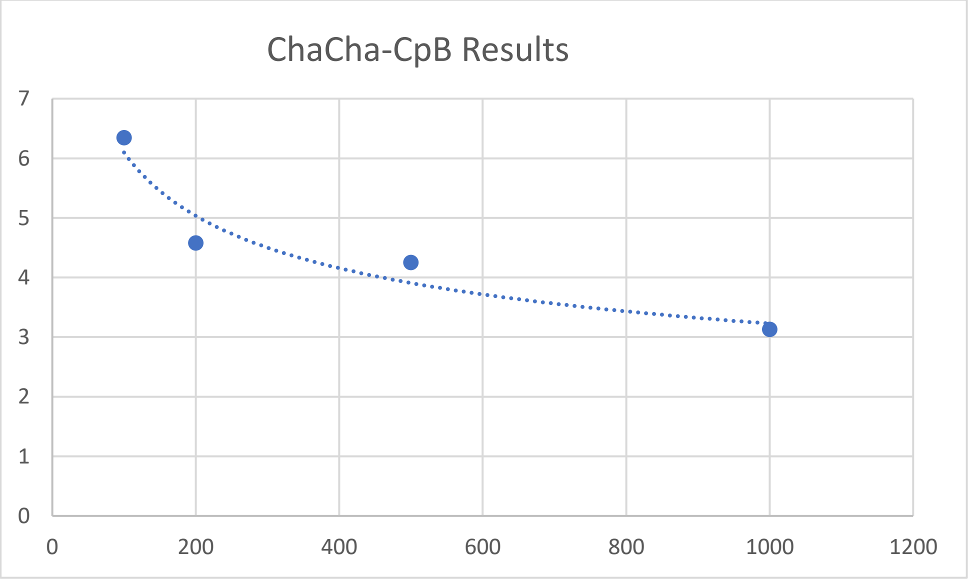


As you can see RSA could more appropriately be measured in mega cycles which means it requires a lot more cycles per byte. To achieve low speeds would require an extremely fast cycle speed using optimization not currently available.

5.3.3 ChaCha20



These are the time results for a software implementation of ChaCha20 which belongs to the salsa encryption family. ChaCha is the only stream cipher I am testing and would expect it to produce the best results out of anything I have used tested so far. It is being tested using the exact same parameters as the previous algorithms, running each message size 10,000 times for an average result.



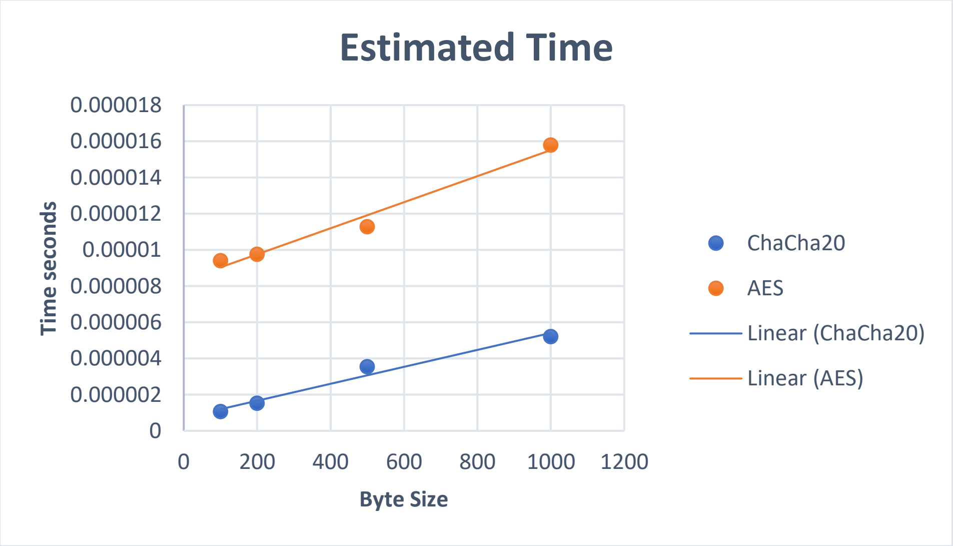
ChaCha20 had a very low cycles per byte, we also know that ChaCha20 performs better than other algorithms on lower performance devices. We can see this by on lower byte inputs that cycles per byte does not increase as dramatically as AES or RSA. In comparison to the tables of figures(appendix B) for Salsa20 and ChaCha software speeds taken from [6], it shows that cycles per byte in the ranges of 3.95 to 15.03 depending on hardware used. It is also reasonable to suggest lower cycles per byte as more optimization comes in for chach20 as it becomes more widely used.

5.4 Evaluation

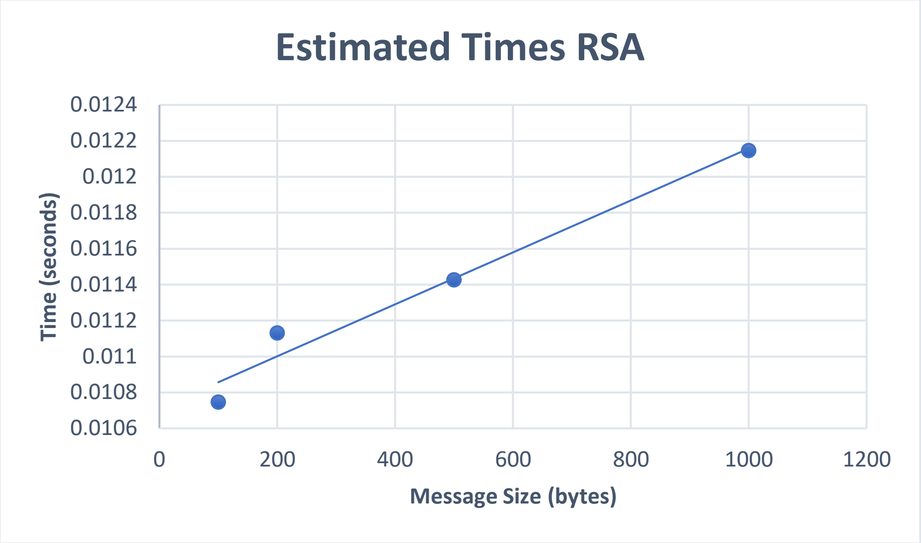
Here I will discuss suggested implementation for SMV, MMS and GOOSE based on the findings I have just discussed.

5.4.1 Estimated Timings

To increase the usability of the findings we can use the cycles per byte, calculated at 5.3 and find the CPU speed of processors used in a protection relay to estimate projected times for an IED. There are many different processors they could use and a wide variety of different architectures. I will be using The Sitara™ AM335x processor, which is one of the most popular processors for industrial HMI applications [12]. This has two different processor speeds; I have calculated more timings based on the lower one of 600 MHZ.



This shows estimated times in seconds for both AES and ChaCha20 on this processor using the cycles per byte at each message size. It is also worth pointing out that ChaCha20 performs better than AES in mobile and lower power devices due to being more lightweight while in hardware on optimized machines AES can outperform ChaCha20.



5.4.2 GOOSE and SMV

The timing requirements of GOOSE and SMV is 3ms and the size is 92-250 and 150 bytes respectively, on average, we can see from the graph that both AES and ChaCha20 can be used even if the network latency is around 2ms such as a WLAN topology. The network latency of a 4G network means that it will not meet timing requirements. The theoretical limit of 5G is 1ms although it would not reach this time in a high connectivity area, it could be possible to have an IED on a 5G connection. RSA has too low of a latency time, as seen from the Estimated Times RSA graph, to be considered for this connection time and due to the Publisher-Subscriber architecture would not be a suitable encryption method. In an IED scenario, due to ChaCha20 performing better on less powerful machines, it is a better choice for a smart grid environment and is less common than AES which had more attention and cryptanalysis for potential attacks. Both algorithms would stop man-in-the-middle attacks if the secret keys remain secure.

5.4.3 MMS

MMS has much slower critical times depending on the messages that are using the protocol at the time, they range from 500 to 1000 ms and work on a more traditional peer-peer architecture. Though the RSA could be used to support this message type it would require each sender to have the public keys for every node it would be sending. It would be simpler to implement one of the other algorithms proposed, such as ChaCha20, and would be easier to setup as all message types would be using the same communication method.

Chapter 6 Future Work

There are still parts of the project that could have additional work carried out. There are many different algorithms that could be tested as well as potentially more accurate ways to obtain the results. Setting up more complex simulations of different hardware simulations would produce better results for a smart grid environment.

For better testing of the encryption methods, a small test network could be built to help replicate a real-world smart grid. This would help gather accurate measurements of the latency within the network and opens up more areas for testing such as security, performance and increased power drain, this would improve the implementation into real systems.

A better testing system could also be used to test more real-world scalability while taking into account the structure of a Publisher Subscriber system, there is already some existing work on topics such as scalable key management.

Chapter 7 Conclusions

In conclusions, this investigation was aimed at proposing secure solutions to solve security issues in a smart grid setup. The solutions were tested for time, performance, and security.

I have also developed a tool to aid in testing the encryption algorithms with a variety of different affecting factors such as the network type and size of the message being sent. The readings given by the tool where then used to create more accurate depictions of how the algorithms were performing, which gave backing to my proposed solutions for message encryption. I have considered the architecture of the smart grids systems when conducting testing and believe the solutions could be scaled up and employed in larger network setups, in my estimated timings section I have also suggested timings that you would expect if you employed my software solutions using machines that would be commonly found within the smart grid environment, Overall, this project has successful met the criteria of proposing a workable encryption solution that would meet the criteria of a smart grid system.

Chapter 8 Reflection

8.1 Technical Reflection

During this process I have used a number features I have not used before despite have a good understanding of java, the java security and crypto modules were very interesting in use and allowed me to get better timing results due to optimisation. I had also never made a GUI before and learning how to incorporate the awt modules into the tool was interesting to learn.

It was an initial struggle to understand what I wanted to achieve by this project due to the complex nature of smart grid systems, there architecture and the IEC regulations that define the message types. However, through multiple papers I gained a understanding of how the smart grids communicated and the structure of the message types I was trying to encrypt. It was an enlightening experience to research algorithms from papers written by their creators themselves.

8.2 Management

I believe that, while I created a good project plan with some realistic time considerations, I did not make a good attempt to stick to said timing plan nor did I use any kind of task tracking for the implementation of the tool.

While I was aware of my next task and made good progress with the implementation, i hit setbacks with algorithm implementation while attempting to reach performance requirements. If I had managed some of the tasks better and done more design work before starting the coding, I could have created a more efficient structure requiring less time to incorporate the tool and gui with the encryption algorithms. This may have left me more time to work on other security aspects relating to the problem such as key transmission, secure key storage.

Abbreviations

AES – Advanced Encryptions Standard

RSA - Rivest–Shamir–Adleman

GOOSE - Generic Object-Oriented Substation Event

MMS – Manufacturing Message Specification

SMV - Sampled Measured Values

ISO – the International Organization for Standardization

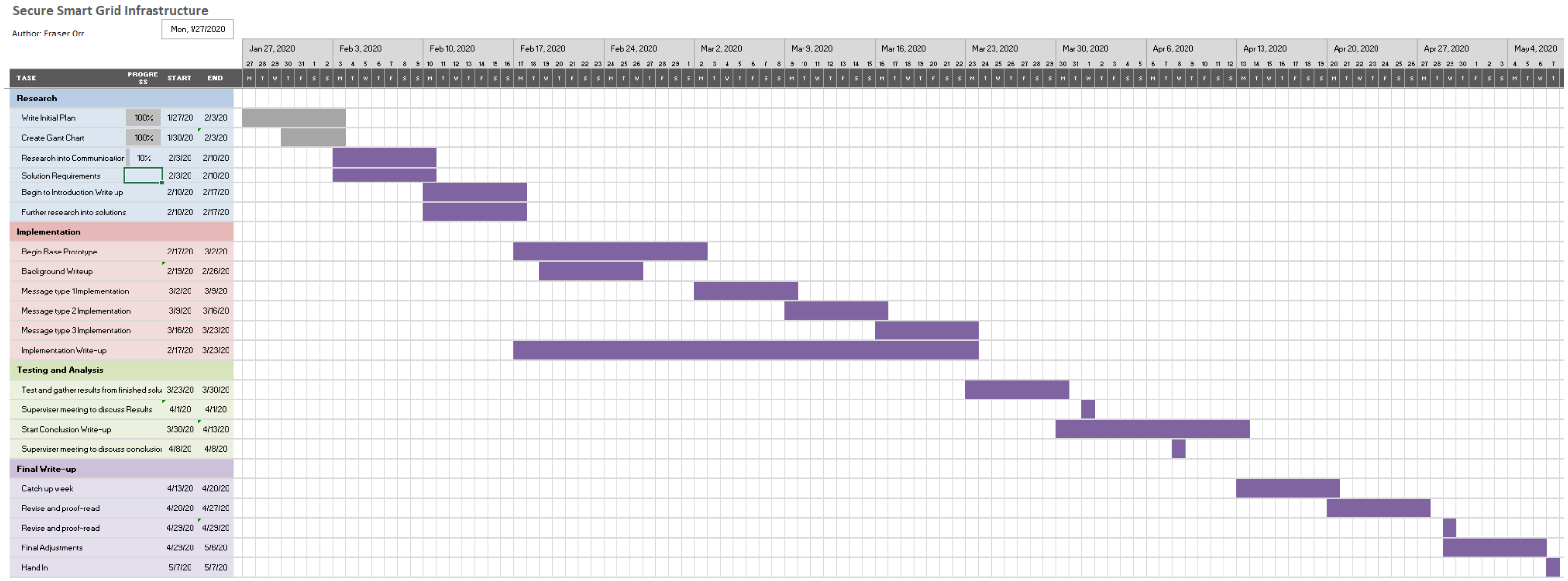
IEC - International Electrotechnical Commission

CpS – Cycles per second

CpB – Cycles per Byte

Appendix A

Gantt Chart



Secure Smart Grid Infrastructure

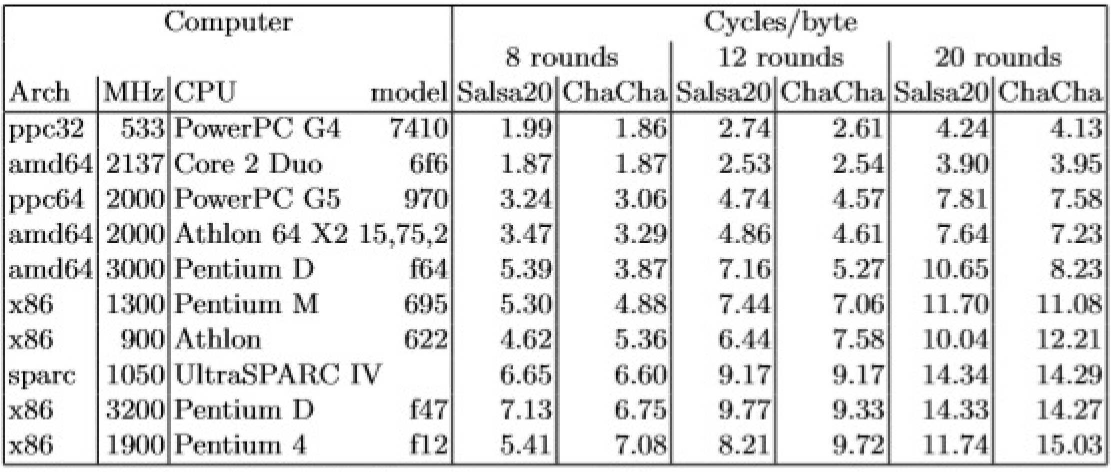
Author: Fraser Orr

Mon, 1/27/2020

TASK	PROGRESS	START	END
Research			
Write Initial Plan	100%	1/27/20	2/3/20
Create Gant Chart	100%	1/30/20	2/3/20
Research into Communicatior	10%	2/3/20	2/10/20
Solution Requirements		2/3/20	2/10/20
Begin to Introduction Write up		2/10/20	2/17/20
Further research into solutions		2/10/20	2/17/20
Implementation			
Begin Base Prototype		2/17/20	3/2/20
Background Writeup		2/19/20	2/26/20
Message type 1 Implementation		3/2/20	3/9/20
Message type 2 Implementation		3/9/20	3/16/20
Message type 3 Implementation		3/16/20	3/23/20
Implementation Write-up		2/17/20	3/23/20
Testing and Analysis			
Test and gather results from finished solu		3/23/20	3/30/20
Superviser meeting to discuss Results		4/1/20	4/1/20
Start Conclusion Write-up		3/30/20	4/13/20
Superviser meeting to discuss conclusioı		4/8/20	4/8/20
Final Write-up			
Catch up week		4/13/20	4/20/20
Revise and proof-read		4/20/20	4/27/20
Revise and proof-read		4/29/20	4/29/20
Final Adjustments		4/29/20	5/6/20
Hand In		5/7/20	5/7/20

Appendix B

ChaCha model times from [6]



Computer				Cycles/byte					
				8 rounds		12 rounds		20 rounds	
Arch	MHz	CPU	model	Salsa20	ChaCha	Salsa20	ChaCha	Salsa20	ChaCha
ppc32	533	PowerPC G4	7410	1.99	1.86	2.74	2.61	4.24	4.13
amd64	2137	Core 2 Duo	6f6	1.87	1.87	2.53	2.54	3.90	3.95
ppc64	2000	PowerPC G5	970	3.24	3.06	4.74	4.57	7.81	7.58
amd64	2000	Athlon 64 X2	15,75,2	3.47	3.29	4.86	4.61	7.64	7.23
amd64	3000	Pentium D	f64	5.39	3.87	7.16	5.27	10.65	8.23
x86	1300	Pentium M	695	5.30	4.88	7.44	7.06	11.70	11.08
x86	900	Athlon	622	4.62	5.36	6.44	7.58	10.04	12.21
sparc	1050	UltraSPARC IV		6.65	6.60	9.17	9.17	14.34	14.29
x86	3200	Pentium D	f47	7.13	6.75	9.77	9.33	14.33	14.27
x86	1900	Pentium 4	f12	5.41	7.08	8.21	9.72	11.74	15.03

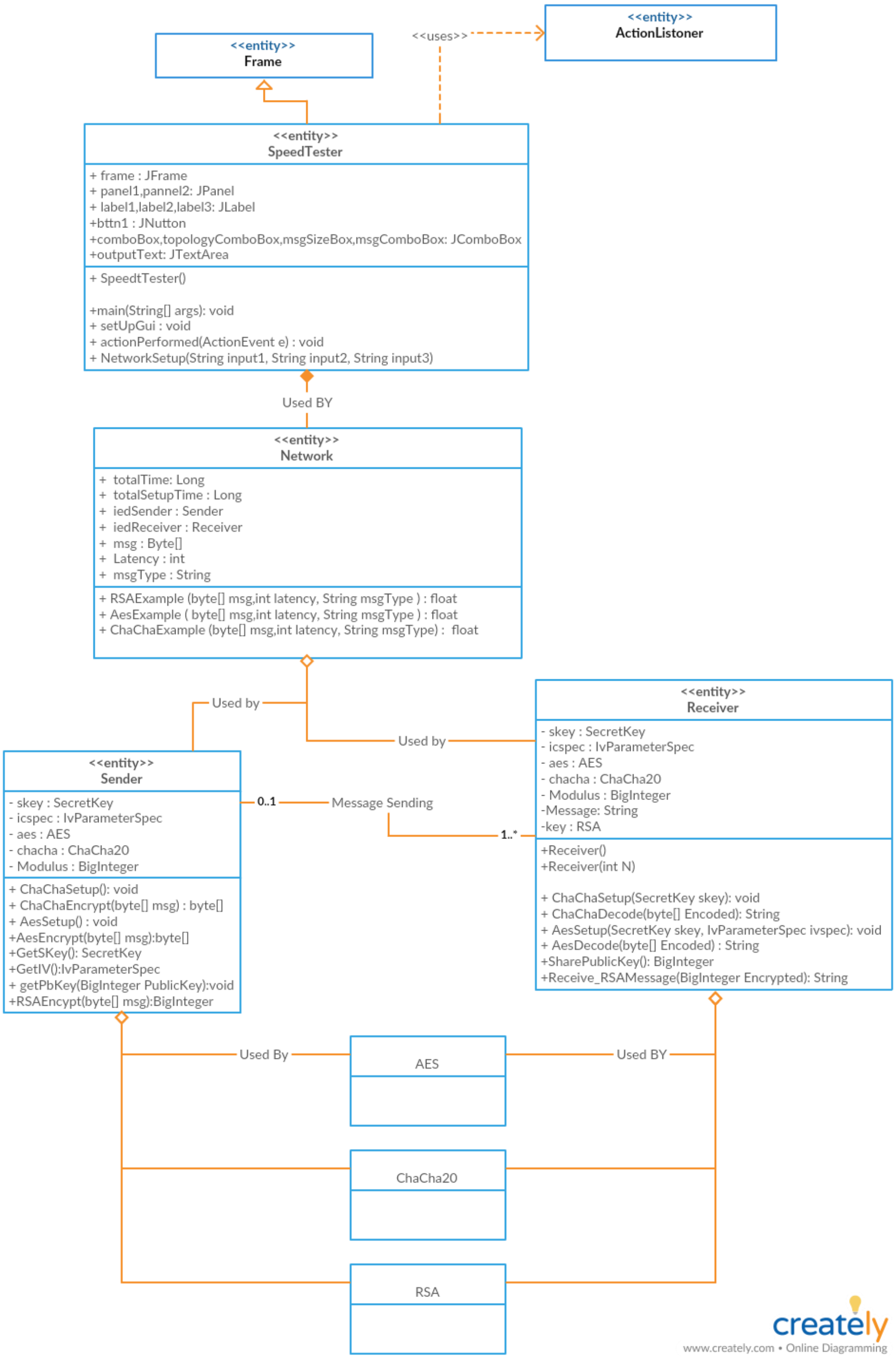
Table 1.3. Salsa20 and ChaCha software speeds for long streams. Sorted by ChaCha8 column.

Appendix C – User Guide

To run the support code, everything should be compiled and you should run the java file Speed Tester, via command line. I.e. java SpeedTester

Appendic D

Class Diagram



<<uses>>
<<entity>>
ActionListoner
<<entity>>
Frame
<<entity>>
SpeedTester
+ frame : JFrame
+ panel1,pannel2: JPanel
+ label1,label2,label3: JLabel
+bttn1 : JNutton
+comboBox,topologyComboBox,msgSizeBox,msgComboBox: JComboBox
+outputText: JTextArea
+ SpeedtTester()
+main(String[] args): void
+ setUpGui : void
+ actionPerformed(ActionEvent e) : void
+ NetworkSetup(String input1, String input2, String input3)
Used BY
<<entity>>
Network
+ totalTime: Long
+ totalSetupTime : Long
+ iedSender : Sender
+ iedReceiver : Receiver
+ msg : Byte[]
+ Latency : int
+ msgType : String
+ RSAExample (byte[] msg,int latency, String msgType) : float
+ AesExample (byte[] msg,int latency, String msgType) : float
+ ChaChaExample (byte[] msg,int latency, String msgType) : float
Used by
Used by
<<entity>>
Receiver
- skey : SecretKey
- icspec : IvParameterSpec
- aes : AES
- chacha : ChaCha20
- Modulus : BigInteger
-Message: String
-key : RSA
+Receiver()
+Receiver(int N)
+ ChaChaSetup(SecretKey skey): void
+ ChaChaDecode(byte[] Encoded): String
+ AesSetup(SecretKey skey, IvParameterSpec ivspec): void
+ AesDecode(byte[] Encoded) : String
+SharePublicKey(): BigInteger
+Receive_RSAMessage(BigInteger Encrypted): String
<<entity>>
Sender
- skey : SecretKey
- icspec : IvParameterSpec
- aes : AES
- chacha : ChaCha20
- Modulus : BigInteger
+ ChaChaSetup(): void
+ ChaChaEncrypt(byte[] msg) : byte[]
+ AesSetup() : void
+AesEncrypt(byte[] msg):byte[]
+GetSKey(): SecretKey
+GetIV():IvParameterSpec
+ getPbKey(BigInteger PublicKey):void
+RSAEncypt(byte[] msg):BigInteger
0..1
Message Sending
1..*
Used By
AES
Used BY
ChaCha20
RSA
creately
www.creately.com • Online Diagramming

References

[1] C. R. Ozansoy, A. Zayegh and A. Kalam, "The Real-Time Publisher/Subscriber Communication Model for Distributed Substation Systems," in IEEE Transactions on Power Delivery, vol. 22, no. 3, pp. 1411-1423, July 2007, doi: 10.1109/TPWRD.2007.893939

[2] Xia, Fei & Xia, Zongze & Huang, Xiaobo. (2015). Summary of GOOSE Substation Communication. MATEC Web of Conferences. 25. 01007. 10.1051/matecconf/20152501007.

[3] N. Saxena, S. Grijalva and B. J. Choi, "Securing restricted publisher-subscriber communications in smart grid substations," 2018 10th International Conference on Communication Systems & Networks (COMSNETS), Bengaluru, 2018, pp. 364-371, doi: 10.1109/COMSNETS.2018.8328220.

[4] Nhat Nguyen-Dinh, Gwan-Su Kim and Hong-Hee Lee, "A study on GOOSE communication based on IEC 61850 using MMS ease lite," 2007 International Conference on Control, Automation and Systems, Seoul, 2007, pp. 1873-1877, doi: 10.1109/ICCAS.2007.4406651.

[5] More, S. and Bansode, R., 2015. Implementation Of AES With Time Complexity Measurement For Various Input. [ebook] Global Journals Inc. (USA). Available at: <https://globaljournals.org/GJCST_Volume15/3-Implementation-of-AES.pdf> [Accessed 22 March 2020].

[6] Bernstein, Daniel. (2008). ChaCha, a variant of Salsa20.

[7] Daniel J. Bernstein. 2008. The Salsa20 Family of Stream Ciphers. New Stream Cipher Designs: The eSTREAM Finalists. Springer-Verlag, Berlin, Heidelberg, 84–97. DOI:https://doi.org/10.1007/978-3-540-68351-3_8

[8] Docs.oracle.com. n.d. Cipher (Java Platform SE 7). [online] Available at: <https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html>

[9] A. Ingalalli, K. S. Silpa and R. Gore, "SCD based IEC 61850 traffic estimation for substation automation networks," 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Limassol, 2017, pp. 1-8.

[10] Cryptopp.com. n.d. [online] Available at: https://www.cryptopp.com/benchmarks-p4.html

[11] Cryptopp.com. n.d. [online] Available at: https://www.cryptopp.com/benchmarks-p4.html

[12] MUNDRA, A. and AGRAWAL, M., 2017. Human Machine Interface (HMI) For Protection Relay Reference Design. Texas Instruments Incorporated.