Attacks and analysis on Trivium

1st Karthikan Theivendran
MASc ECE. University of Waterloo
karthikan.theivendran@uwaterloo.ca

2nd Kaushal Kiran BS
MEng ECE. University of Waterloo
kkbangal@uwaterloo.ca

***Abstract*—With the advancement of technology, the Internet of Things (IoT) brings a huge number of devices that link to each other and gather massive amounts of data. As a result, IoT security requirements are critical. Currently, cryptography is used to protect networks for authentication, secrecy, data integrity, and access control. Traditional cryptography protocols, on the other hand, may not be suitable in all IoT contexts due to the resource constraints. Therefore, a variety of lightweight cryptography methods and protocols have been proposed by cryptography researchers. In this paper, we investigate the state of the art stream ciphers such as Simple-LFSR, A5/1 and Trivium ciphers and analyse their internal structure. We further implement the stream ciphers in software and conduct cryptanalysis such as known-plaintext attack on them to understand their immunity to attack. We conclude that Trivium is a secure stream cipher compared to Simple-LFSR based stream cipher and A5/1.**

***Index Terms*—Trivium, LFSR, Known-Plain Text Attack, Berelekamp-Massey Algorithm**

I. Introduction

Building safe and secure systems is a must in the age where privacy and security are big concerns. The Crypt algorithms are pervasive and have made it through every device possible from Web Browsers to Credit cards, they make a very indispensable part of a secure system. In this project, we look at stream ciphers that are a class of symmetric ciphers which encrypt bits individually. This is achieved by adding a bit from a key stream to a plain text bit. They tend to be small and fast, they are particularly relevant for applications with little computational resources, e.g. for cell phones or other small embedded devices. Cryptanalysis is the science and sometimes art of breaking cryptosystems. Attackers always look for the weakest point, and the cipher under consideration, though having a large key space does not guarantee security as it might still be vulnerable against different analytical attacks. Therefore, it's important to conduct thorough analysis on these cipher's immunity to attack before implementing them in real-world IoT applications.

II. Cryptography

A system can be considered secure if an attacker is unable to access the secret information. An attacker or a cryptanalyst is a third party who has access to all public data and attempts to deduce private secret information. In general, cryptology involves mathematical theories including number theory and the application of formulas and algorithms, and this becomes the foundation for cryptography and cryptanalysis. The objective of cryptography is to hide the meaning of a message using methods of secret encoding. Cryptanalysis is a field of study related to the breaking of these cryptosystems. This field is well studied and explored in academica as it's important to understand if a crypto method is secure and how easy is it to break it [1][20].

There are three main branches in Cryptography, namely

1) Symmetric Algorithms where two parties have an encryption and decryption method for which they share a secret key. Symmetric ciphers are popular and primarily used for data encryption and integrity checks of messages.
2) In asymmetric or public-key cryptography, a user has both a secret and a public key, similar to symmetric cryptography. This is mainly used for applications such as digital signatures and key establishment, and also for classical data encryption.
3) Cryptographic Protocols are built on symmetric and asymmetric algorithms and mainly used in internet applications such as the Transport Layer Security scheme (TLS).

A. Stream Ciphers

Stream ciphers are based on symmetric encryption algorithms and can be designed to be very fast. This makes them ideal for use in telecommunication, low-level network and IoT network encryption. Unlike block ciphers, which work with vast blocks of data, stream ciphers work with individual symbols of the underlying alphabet, which are usually bits. When the same key is used to encrypt any particular plaintext using a block cipher, the ciphertext will be the same. The encryption function of a stream cipher is time-varying, therefore the transformation of these smaller plaintext units will vary depending on when they are encountered during the encryption process [1][20].

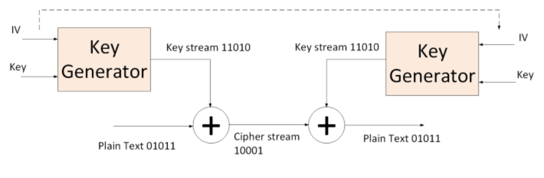
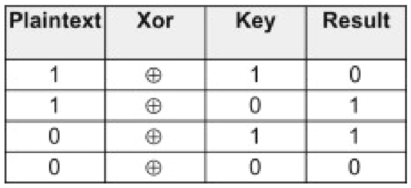


Fig. 1. Stream Cipher

In stream ciphers, a sequence of bits known as keystreams are generated. This keystream is combined with plain-text in order to encrypt the plain-text [1]. This is usually accomplished with the bitwise XOR operation as shown in below Fig. 2.



Plaintext	Xor	Key	Result
1	⊕	1	0
1	⊕	0	1
0	⊕	1	1
0	⊕	0	0

Fig. 2. Example of XOR Operation

B. *Pseudo-random generator*

PRNGs (pseudorandom number generators) produce sequences based on an initial seed value. If generating actual random numbers is too time-consuming, PRNGs can be used instead. PRNG is an algorithm that generates a lengthy sequence of bits which look random from a short sequence of random bits. In a stream cipher cryptosystem, PRNGs are used to extend a secret key into a keystream [20].

C. *One-time pads*

A one-time pad employs a string of bits generated entirely at random. Because the keystream has the same length as the plaintext message and may only be used once, a large number of keystreams may be necessary. The ciphertext is created by combining the random string with the plaintext using bitwise XOR. Even an opponent with limitless computational capabilities can only predict the plaintext if he or she sees the ciphertext since the whole keystream is random. Because the secret key (which can only be used once) is as lengthy as the message, significant key management issues arise. As a result, despite being fully secure, the one-time pad is deemed impracticable [20].

D. *Linear Feedback Shift Register*

A keystream generator's primary function is to generate symbols that appear to be as random as possible and the distribution of bits should be uniform and unpredictable. A Linear Feedback Shift Register (LFSR) can be utilized to produce such a binary bit sequence. LFSRs are extensively employed within stream ciphers, despite the fact that their direct output is not a good keystream generator because each element is just a linear combination of the preceding symbols. A LFSR is a shift register with some of its outputs as exclusive-or (XOR) of its inputs. The register is made up of a sequence of cells that can each hold one symbol at a time [20]. Specific outputs called the tabs feedback and influence the input. The state of the LFSR at time t is the content of the register at time t. The following Fig. 3. illustrates one step of an 11-bit LFSR with initial seed 01101000010 and tap position 8.

E. *A5/1 Stream Cipher*

A5/1 is a stream cipher used in the GSM cellular phone standard to ensure over-the-air communication privacy [2]. It's one of numerous A5 security protocol implementations. It was maintained a secret at first, but via leaks and reverse engineering [3] [4], it became public knowledge. The encryption has a

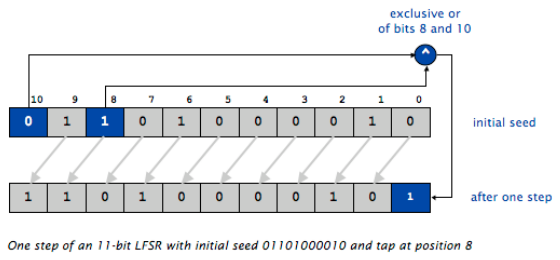


Fig. 3. LFSR

number of severe flaws that have been discovered . As shown in below Figure, Three LFSRs are used in the A5/1 stream cipher. A register is clocked if its clocking bit shown in orange matches one or both of the other two registers' clocking bits [2][3].

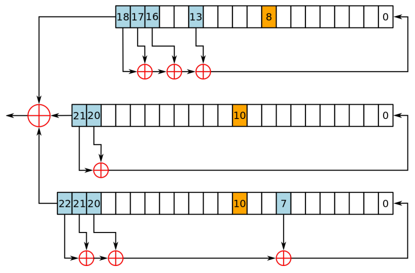


Fig. 4. A5/1 stream cipher

F. *Trivium*

Trivium is a novel stream cipher that use an 80-bit key. It's built around a set of three shift registers. Unlike the LFSRs we discussed in the previous section, these feedback shift registers have nonlinear components that are employed to generate the output of each register[20].

As shown in Fig. 5., three shift registers A, B and C form the core of Trivium and their respective lengths are 93, 84, 111. The key stream is generated by the XOR-sum of all 03 register outputs. The output of each register is linked to the input of another register, which is a unique characteristic of the encryption. As a result, the registers are arranged in a circular pattern as shown in Fig. 6. This cipher can be considered to be made up of one circular register with an overall length of 93+84+111 = 288.

The input parameters of almost all recent stream ciphers are a key k and an initialization vector (IV). The IV acts as a randomizer and should take a different value for each encryption session. It does not need to be kept secret; it only needs to change for every session. These are sometimes referred to as nonces ("number used once"). The main objective of this is to ensure that there aren't any identical keystreams produced by the cipher even if the key has not changed. If keystreams don't change, it's vulnerable to known-plaintext attacks. For example, if an attacker knows a plain text from a first encryption, the attacker will be able to determine

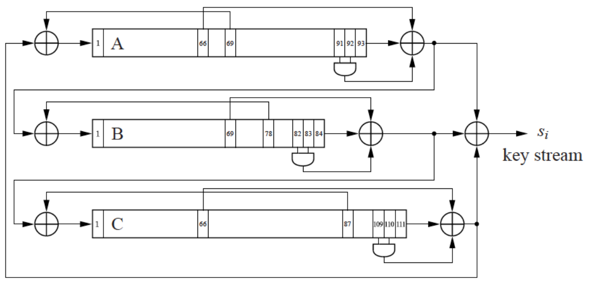


Fig. 5. Internal structure of Trivium

the corresponding key stream [20]. The second encryption, which uses the identical key stream, may now be decoded quickly. Stream cipher encryption is extremely deterministic without a changing IV.

It's worth noting that the AND operation is equivalent to modulo 2 arithmetic multiplication. When we multiply two unknowns, and the unknowns that an attacker wishes to retrieve are in the register contents, the resultant equations are no longer linear since they include products of two unknowns. As a result, feedforward pathways containing the AND operation are critical for Trivium security because they prohibit attacks that use the cipher's linearity, such as the one demonstrated in the preceding section for simple LFSRs [20].

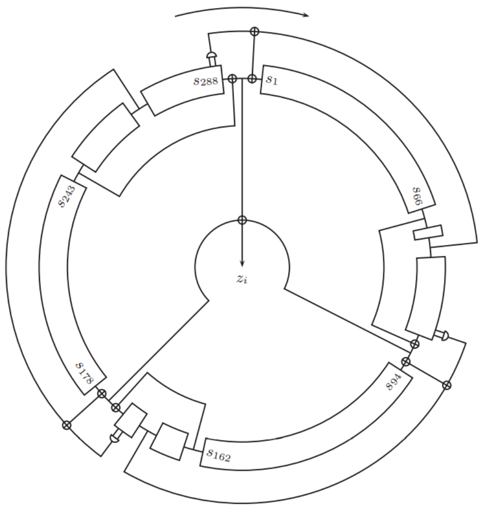


Fig. 6. Trivium

Trivium only requires a 288-bit shift register, seven 3-input X-OR gates, and three AND gates in terms of hardware [20]. For each clock cycle, we also generated 64-bit output.

III. Design and Implementation

A. Tools

A typical sensor network normally consists of several remote devices talking to each other over a wireless protocol. These sensors carry important information regarding the environment under consideration. The information would be helpful in taking decisions that ensures safe operation of the required task.

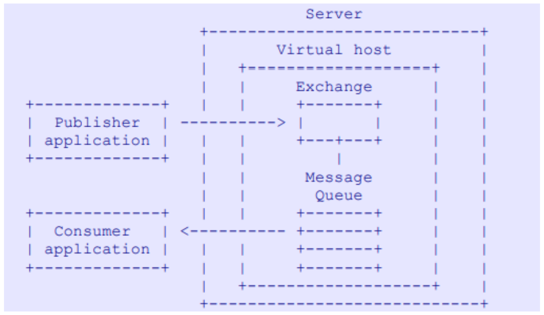
As discussed it becomes very important to have a secure mode of communication between these devices. To demonstrate a real wireless sensor network scenario, we use the following software tools, that would help us understand the need of secure message communication between two devices.

- Erlang OTP
- AMQP
- Rabbit MQ
- Wireshark

1) Erlang OTP: Erlang is a programming language used to build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance [5].

2) AMQP: AMQP is a lightweight M2M protocol, which was developed by John O'Hara at JPMorgan Chase in London, UK in 2003. It is a corporate messaging protocol designed for reliability, security, provisioning and interoperability [6]. It offers a wide range of features related to messaging such as a reliable queuing, topic-based publish-and-subscribe messaging, flexible routing and transaction. The main components that make up the up the AMQP protocol are:

- **Exchange**: Receives messages from publisher primarily based programs and routes them to 'message queues'.
- **Message Queue**: Stores messages until they may thoroughly process via the eating client software.
- **Binding**: States the connection between the message queue and the change.
  AMQP does two main things with messages, it routes them to different consumers depending on arbitrary criteria, and it buffers them in memory or on disk when consumers are not able to accept them fast enough [7].



```
                                 Server
                       +-----------------------------+
                       |        Virtual host         |
                       |   +-------------------+     |
                       |   |     Exchange      |     |
+--------------+       |   |     +--------+    |     |
|  Publisher   | ----------> |     |        |    |     |
| application  |       |   |     +---+----+    |     |
+--------------+       |   |         |         |     |
                       |   |      Message      |     |
                       |   |       Queue       |     |
+--------------+       |   |     +--------+    |     |
|  Consumer    | <----------     +--------+    |     |
| application  |       |   |     +--------+    |     |
+--------------+       |   |     +--------+    |     |
                       |   +-------------------+     |
                       +-----------------------------+
```

Fig. 7. Simple AMQP model

In this protocol the information is organised into frames and frames carry protocol methods and other information and the Rabbit MQ implementation works with TCP/IP as the underlying Transport layer protocol.

Within a single socket connection, there can be multiple independent threads of control, called "channels".Each frame is numbered with a channel number. By interleaving their frames, different channels share the connection. For any given channel, frames run in a strict sequence that can be used to drive a protocol parser.

3) Rabbit MQ: RabbitMQ is one of the most widely used open-source message brokers. It is originally based on the Advanced Message Queuing Protocol (AMQP) and also supports many messaging protocols like MQTT, and STOMP it is also called the hybrid broker. It supports several variations of pub-sub, points to point, request-reply messaging techniques [8].

It is a message broker: which accepts and forwards messages. You can think about it as a post office: when you put the mail that you want posting in a post box, you can be sure that Mr. or Ms. Mailperson will eventually deliver the mail to your recipient. In this analogy, RabbitMQ is a post box, a post office and a postman.

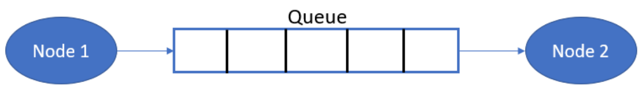


Fig. 8. Simple Rabbit MQ data pipeline

The Rabbit MQ consists:

- **Producer** : A program that sends messages
- **Queue** : It's essentially a large message buffer. Many producers can send messages that go to one queue, and many consumers can try to receive data from one queue.
- **Consumer** : Is a program that mostly waits to receive messages

4) Wireshark: Wireshark is a free and open-source packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development, and education [9]. It is the most often-used packet sniffer in the world and does the following three things Packet Capture, Filtering and Visualization.Here, we use this to understand the AMQP message formation, generation and to guarantee delivery of message between two communicating nodes .

This forms a necessary tool as an attacker can always use this to sniff the network packets and understand the details of the message delivered which shows us why sending ciphered information over the network forms the very basic of building secure systems.

In this project we have used Wireshark to understand and inspect the AMQP traffic. Wireshark provides a lot of information explaining what clients (applications) and RabbitMQ nodes do. This information can and should be used to derive insights into system behavior that is difficult to observe otherwise.

This will be explained in detail when we talk about Software Implementation in the next section.

B. Software Implementation

In this section we give talk about the details of the simulated wireless network set-up that was used to understand how encryption techniques can be used to deliver ciphered messages that cannot be understood by the attacker unless he know details of the method used to conver the plain text information to cipher text.

Here, we consider the case where two remote devices Node 1 and Node 2 communication over a wireless medium using the AMQP protocol.We declare Node 1 to send a message to Node 2 and for the sake of simplicity we consider text messages.

We also consider an attacker who has gained access to the channel with which he is able to sniff data that is being communicated between the two devices.This attacker performs the following Cryptanalysis techniques that will be discussed in section 3.

- Known-Plain Text attack
- Berlekamp-Massey Algorithm to determine the linear span of the cipher

The main language used to simulate such an environment was python.

The project contains the following files for simulating a WSN:

- TriviumTX.py : Node 1 that sends message to Node 2
- TriviumRX.py : Node 2 that receives information
- TriviumAttacker.py : An attacker with access to the channel of communication

Following are library files that implement Trivium, Simple LFSR and certain cryptanalysis techniques that help the attacker to decipher the information sent.

- CipherGen.py : Methods to generate cipher text
- CryptanalysisLib.py : Set of cryptanalysis techniques

1) Wireless sensor network simulation: The above discussed files for python simulation of WSN works as shown in the Fig. 9. below.

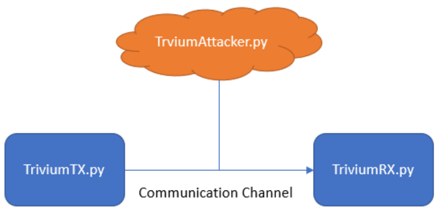


Fig. 9. Simple WSN

Here, *TriviumTX.py* and *TriviumRX.py* are two nodes and in the language of Rabbit MQ the latter is a consumer and the other is a producer. To understand how an attacker with illegal access to the channel and can decipher vital information ,we consider *TriviumAttacker.py* just another consumer who has no or very little information about the messages sent by the publisher to the consumer.It is his task to perform cryptanaylsis techniques to understand the encrypted messages sent over the network.

This architecture is implemented on Rabbit MQ as publish-subscribe pattern.In this system every running copy of the receiver program or the nodes will get the messages sent by the sender node.

The below Fig. 10. explains in general the Rabbit MQ implementation of a simple WSN.

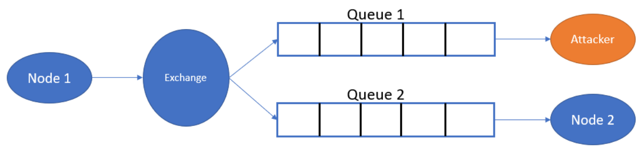


Fig. 10. Publish-Subscribe

- **TriviumTX.py**
  - Sending Information : The first thing we need to do is to establish a connection with RabbitMQ server [3].

```
import pika
def rabbit_send(cipherMessage):
   connection = pika.BlockingConnection(
      pika.ConnectionParameters(host='localhost'))
   channel = connection.channel()
```

Here, we learn about exchnage in Rabbit MQ,Instead of sending messages to queues we can send messages to an exchange. An exchange is a very simple thing. On one side it receives messages from producers and the other side it pushes them to queues. The exchange must know exactly what to do with a message it receives. Should it be appended to a particular queue ? Should it be appended to many queues ? Or should it get discarded. The rules for that are defined by the exchange type [10].
We define the exchange as follows.

```
channel.exchange_declare(exchange='exchangeOne',
     exchange_type='fanout')
```

Fanout There are a few exchange types available: direct, topic, headers and fanout. We'll focus on the last one – the fanout. A fanout exchange is an exchange which routes the received message to all the queues bound to it. When the producer sends the message to fanout exchange, it copies the message and routes to all the queues that are bound to it. It just ignores the routing key or any pattern matching provided by the producer. This type of exchange is useful when the same message is needs to be stored in one or more queues [11].
Publishing message to exchange and before exiting the program we need to make sure the network buffers were flushed and our message was actually delivered to RabbitMQ. We can do it by gently closing the connection.

```
channel.basic_publish(exchange='exchangeOne',
     routing_key='', body=cipherMessage)
print(" [x] Sent %r" % cipherMessage)
connection.close()
```

- **TriviumRX.py** This RX acts as a node that listens to a Rabbit MQ queue. This is the intended node that is aware of the details of the Cryptographic algorithm used to convert plain text information to cipher text.
The receiver first has to make connection to the Rabbit MQ server as shown in the code excerpt below.

```
def main():
   connection = pika.BlockingConnection(
      pika.ConnectionParameters(host='localhost'))
   channel = connection.channel()
```

The next step, just like before, is to make sure that the queue exists. Creating a queue using queue_declare is idempotent we can run the command as many times as we like, and only one will be created.

```
result = channel.queue_declare(queue='QueueTwo',
     exclusive=True)
queue_name = result.method.queue
```

Receiving messages from the queue is more complex. It works by subscribing a callback function to a queue. Whenever we receive a message, this callback function is called by the Pika library. In our case this function will print on the screen the contents of the message.

```
def callback(ch, method, properties, body):
      print(" [x] Received Encrypted %r" % body)
      print("Decrypting Message :")
      decrypted_message = decrypt_message(key, iv,
           body.decode('utf-8'), 2)
      print(" [x] Decrypted Message %r" %
           decrypted_message)
```

The functions and working of the decrypt method is dealt in the Cryptanalysis section in further detail.
Next, we need to tell RabbitMQ that this particular callback function should receive messages from our 'QueueTwo' queue.

```
channel.basic_consume(
    queue=queue_name, on_message_callback=callback,
         auto_ack=True)
channel.start_consuming()
```

- **TriviumAttacker.py**
This python program acts as an attacker sniffing data that is communicated between nodes 1 and 2 implemented as explained above.
The attacker performs two type of cryptanalysis method implemented as part of the function *perform_attack*. The *attackType* argument takes values 1 or 2 to distinguish between the two different type of attacks.

```
def perform_attack(attackType, messageBody):
   if attackType == 2:
      keystream_decr = CryptanalysisLib.cipher_attack(
         2, "HELLO", messageBody.decode('utf-8'))
      plaintext_decr =
           CryptanalysisLib.decrypt_message(
         2, keystream_decr,
              messageBody.decode('utf-8'))
      return plaintext_decr
   elif attackType == 1:
      keystream_decr = CryptanalysisLib.cipher_attack(
         1, "HELLO", messageBody.decode('utf-8'))
```

```
        plaintext_decr =
            CryptanalysisLib.decrypt_message(
          1, keystream_decr,
                messageBody.decode('utf-8'))
        return plaintext_decr
    pass
```

– Known-plain text attack : This type of attack is performed when the attacker is aware of some parts of the message that is sent and is also having minimal information about the LFSR used.
– Berlekamp-Massey analysis : When the attacker does not know the linear span, period of the random numbers generated and the Initial state he can use this algorithm to find information required to generate keystream that can be XORed with the cipher text to get back the information.

2) Trivium Implementation: Python has implementation of Trivium stream cipher which is based on the C-implementation of the stream cipher.As discussed in Section 2 trivium is a successful stream cipher.

The trivium.c file in the library *pytrivium* forms the basis of the implementation. In this section we go through the source code to understand the working of the Non-linear LFSR trivium.

- **Initializing Trivium**: The trivium as explained needs to be run for 1152 number of iterations after which the output is considered to generate a random keystream This is taken care of the function:

```
void TRIVIUM_init(TRIVIUM_ctx* ctx, const uint8_t
     key[], const uint8_t iv[], uint8_t keylen,
     uint8_t ivlen) { }
```

  This function also calls the below functions that setup the key and initialization vector to in the trivium that is made of three LFSRs A , B and C. After loading the LFSRs we initalize cipher.
- **Key Setup**: The Key setup function is used to load 80-bit key into trivium.Which is done by the function.

```
void TRIVIUM_keysetup(TRIVIUM_ctx* ctx, const uint8_t
     key[]) { }
```

- **IV Setup**: The IV setup function is used to load 80-bit initialization vector into trivium.Which is done by the function.

```
void TRIVIUM_ivsetup(TRIVIUM_ctx* ctx, const uint8_t
     iv[]) { }
```

- **Generating Keystream**: This fnction generates set of 32-bit keystreams by rotation, this has been implemented using the function.

```
void TRIVIUM_genkeystream32(TRIVIUM_ctx* ctx, uint32_t
     output[], uint32_t n) { }
```

  The Non-linear functions of the LFSR is implemented as follows.

```
for (uint32_t i = 0; i < n; i++) {
    // Update LFSRs
    t1 = S64(ctx->lfsr_a, 66) ^ S64(ctx->lfsr_a, 93);
    t2 = S64(ctx->lfsr_b, 69) ^ S64(ctx->lfsr_b, 84);
    t3 = S64(ctx->lfsr_c, 66) ^ S96(ctx->lfsr_c, 111);
    output[i] = REVERT_U32(t1 ^ t2 ^ t3);
    t1 ^= (S64(ctx->lfsr_a, 91) & S64(ctx->lfsr_a, 92))
         ^ S64(ctx->lfsr_b, 78);
    t2 ^= (S64(ctx->lfsr_b, 82) & S64(ctx->lfsr_b, 83))
         ^ S64(ctx->lfsr_c, 87);
    t3 ^= (S96(ctx->lfsr_c, 109) & S96(ctx->lfsr_c,
         110)) ^ S64(ctx->lfsr_a, 69);
    // Rotate LFSRs
    ROTATE_LFSR_3(ctx->lfsr_a, t3);
    ROTATE_LFSR_3(ctx->lfsr_b, t1);
    ROTATE_LFSR_4(ctx->lfsr_c, t2);
}
```

Using pytrivium library we can generate a 32-bit random numbers or keystream by passing the keystream and initialization vector as shown below.

- Selecting the *Key* and *IV* value :

```
key = [0xfa, 0xa7, 0x54, 0x01, 0xae, 0x5b, 0x08, 0xb5,
     0x62, 0x0f]
iv = [0xc7, 0x60, 0xf9, 0x92, 0x2b, 0xc4, 0x5d, 0xf6,
     0x8f, 0x28]
```

- Creating the trivium object :

```
engine = Trivium()
```

- Initializing Trivium with *Key* and *IV* values :

```
engine.initialize(key, iv)
```

- Update and generate random numbers or keystream :

```
engine.update(8)
output = engine.finalize()
print([hex(i) for i in output])
```

  This generates 8 x 32 bit = 256 bits of keystream.

```
Keystream :
['0xa4386c6d', '0x7624983f',
'0xea8dbe73', '0x14e5fe1f',
'0x9d102004', '0xc2cec99a',
'0xc3bfbf00', '0x3a66433f']
```

3) Simple-LFSR Implementation: The simple LFSR is implemented in python using the *pylfsr* package.With this library we can define the initial state of the LFSR, and the feedback polynomial. With this we will be able to generate a keystream that is needed to convert plain text to cipher text.

The function *simple_LFSR()* in the python file *CipherGen.py* returns keystream that is needed to encrypt the length of the plain text information.

```
def simple_LFSR(messsageLength, initstate,
     polynomial,fromAttack):
    if fromAttack == False:
        state = [1, 1, 1]
        fpoly = [3, 2]
        keystream = []
        L = LFSR(initstate=state, fpoly=fpoly,
             counter_start_zero=False)
        for _ in range(messsageLength):
            keystream.append(L.outbit)
            L.next()
        return keystream
    elif fromAttack == True:
```

```
    state = initstate
    fpoly = polynomial
    keystream = []
    L = LFSR(initstate=state, fpoly=fpoly,
         counter_start_zero=False)
    for _ in range(messsageLength):
       keystream.append(L.outbit)
       L.next()
    return keystream
```

The simple_LFSR function takes the arguments:

- *messageLength* : Defines the length of the Plain Text data and also defines the length of the keystream needed to encrypt the data.
- *initstate* : Is a python list that defines the initial state of the LFSR
- *polynomial* : Is a python list which mentions the feedback polynomial used to generate the keystream.
- *fromAttack* : When *True* the function accepts the passed arguments and normally this arguments can be found for an keystream by using the Berlekamp-Massey algorithm.

IV. Cryptanalysis

Classical cryptanalysis is the study of recovering the plaintext x from the ciphertext y, or, alternately, recovering the key k from the ciphertext y. Classical cryptanalysis can be further split into analytical attacks, which analyzes and attacks the internal structure of the encryption method, and brute-force attacks, which consider the encryption algorithm as a black box and exhaustively tests all possible keys [20].

Another category of attack is implementation attacks which are carried out by monitoring the electrical power consumption of a processor that works on the secret key; side-channel analysis may be utilized to derive a secret key. It's also worth noting that implementation assaults are particularly relevant to cryptosystems to which an attacker has physical access, such as smart cards [20].

Social engineering attack is another category of attack where bribery, extortion, trickery, or traditional espionage can all be employed to get a secret key when people are involved [20].

A. Brute-Force Attack

Brute-Force or Exhaustive Key Search attack work on the basis of a basic concept: the attacker obtains the ciphertext from eavesdropping on the channel and a small amount of plaintext, such as the header of an encrypted file. The attacker now just uses all available keys to decipher the first piece of ciphertext. The substitution table is the key to this encryption. The attacker knows they've found the proper key if the resultant plaintext matches the small piece of plaintext [22].

B. Algebraic Cryptanalysis

The algebraic attack's fundamental principle is as easy as converting a problem of attacking a crypto-system into finding and solving a set of polynomial equations. The main issue with the algebraic attack is the solving the system of polynomial equations over any finite field is NP-Complete problem [23].

C. Known-Plain text attack

LFSRs are linear, as their name suggests. The inputs and outputs of linear systems are regulated by linear relationships. Because linear dependencies are relatively easy to examine, this may be a significant benefit in communication systems, for example. A cryptosystem in which the key bits exclusively appear in linear connections, on the other hand, is a very unsafe cipher. Below, we'll look at how an LFSR's linear nature leads to an attack [20].

D. Berlekamp-Massey Algorithm

The Berlekamp-Massey algorithm is a deterministic algorithm which solves the issue of finding the shortest linear feedback shift register capable of creating a finite series of components. The linear complexity of a finite sequence is frequently referred to as the length of this linear feedback shift register. The algorithm determines the shortest LFSR capable of producing the first n elements of any sequence [25].

Assume we wish to discover the smallest LFSR that generates the sequence s, where s is now a T-period semi-infinite periodic sequence. If the input is a 2T sequence of length, the Berlekamp-Massey algorithm can offer a solution. If we wish to determine the linear complexity of a series but don't know the period of the sequence, we may apply the technique described above. Then, after we know that enough consecutive symbols have been processed, we can execute the Berlekamp-Massey method to get both the connection polynomial and the linear complexity [26] [27].

E. Cube Attack

A powerful cryptanalytic approach is the cube attack. It is mostly used on Trivium and Grain in the literature. [12] The cube attack, on the other hand, may be used on any symmetric encryption, stream, or block. There are few cube attack few implementations on the internet. Especially, in 2010, Zhu et al. established a practical framework for cube attack like cryptanalysis. According to literature, a cube attack tool may effectively attack up to 720 rounds of Trivium. [13] [14] [15]

V. Cryptanalysis Implementation

A. Known-Plain text attack

The message delivered between two node 1 and 2 is 'HELLO ALL WELCOME TO ECE 659'.In this method the attacker is considered to know some part of the plain text or message being sent.To demonstrate this we assume that the attacker knows 'HELLO' as being a part of the message.

- **Attack on Simple LFSR**: A simple LFSR as explained in Section 2 is implemented using a the python library *pylsfr* as shown below in the *CipherGen.py* python library file using the function.

```
from pylfsr import LFSR
def simple_LFSR(messsageLength, initstate,
     polynomial,fromAttack):
   if fromAttack == False:
      state = [1, 1, 1]
      fpoly = [3, 2]
      keystream = []
```

```
    L = LFSR(initstate=state, fpoly=fpoly,
         counter_start_zero=False)
    for _ in range(messsageLength):
       keystream.append(L.outbit)
       L.next()
    return keystream
```

The state [1,1,1] defines the inital bits of the register and fpoly [3,2] defines the feedback polynomial. This LFSR can be visualised as.

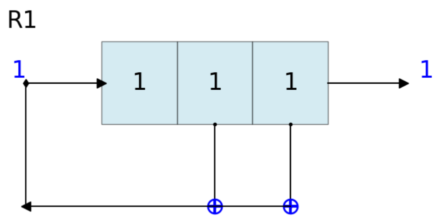


Fig. 11. Simple LFSR

The bit sequence generated as follows

```
Output:[1 1 1 0 0 1 0 1 1 1 0 0 1 0 1 .]
```

It can be seen the message 'HELLO ALL WELCOME TO ECE 659' can be represented as the following in the hex format:

```
PlainText Hex Format:
48454C4C4F20
414C4C205745
4C434F4D4520
544F20454345
20363539
```

When encrypted with the 224 bit PRNG generated by the simple LFSR the ciphered text is as follows:

```
Encrypted Message:
EF96A53875BD
0FEB9FC9237F
D10DE89EAC54
6ED26EE290AC
540CA877
```

Knowing the message 'HELLO' the attacker can XOR the plain text and the cipher text to get keystream.It can be seen that the key repeats every 7 bits hence we can recreate the keystream and decipher the text. This shows the flaws of a simple LFSR and explains the need of non-linear feedback shift registers.

Below is the output of such an implementation.

```
[*] Waiting for exchangeOne.
To exit press CTRL+C

[x] Encrypted
b'EF96A53875BD
0FEB9FC9237F
D10DE89EAC54
6ED26EE290AC
540CA877'
```

```
Performing Known-Plain Text Attack :
HELLO ALL WELCOME TO ECE 659
```

- **Attack on Trivium**: As we know Known-plain text attack in Trivium is not possible and the cipher text was generated with the following *key* and *Initialization vector* values.

```
key = [0xfa, 0xa7, 0x54, 0x01, 0xae, 0x5b, 0x08, 0xb5,
     0x62, 0x0f]
iv = [0xc7, 0x60, 0xf9, 0x92, 0x2b, 0xc4, 0x5d, 0xf6,
     0x8f, 0x28]
```

The cipher text turns out to be:

```
Encrypted Message:
6D597AFA2104
58B01B912A8B
64E430B5FC28
506F6336D01C
E3CBC839
```

But using the same method of Known-plain text attack the deciphered message turns out as:

```
[*] Waiting for exchangeOne.
To exit press CTRL+C

[x] Encrypted
b'6D597AFA2104
58B01B912A8B
64E430B5FC28
506F6336D01C
E3CBC839'

Performing Known-Plain Text Attack:
---§I8˜U|z)]s
```

B. Berlekamp-Massey analysis

- **Attack on Simple LFSR**:We have understood in the previous section that Berlekamp–Massey algorithm is an algorithm that will find the shortest linear feedback shift register (LFSR) for a given binary output sequence [16]. This algorithm is implemented in *CryptanalysisLib.py* file as below.It takes in argument as the sequence of bits or keystream and return the linear span of the LFSR.

```
def Berlekamp_Massey_algorithm(sequence):
   N = len(sequence)
   s = sequence[:]
   ..
   ..
   return (poly\_coeff(f), l)
```

Using the above algorithm we get the linear span of the LFSR as 3. Hence, the first 3 bits keystream $s_2 = 1$, $s_1 = 1$ and $s_0 = 1$ form the initialisation vector. Since we know the part of the message as 'HELLO' we have retrieved the part of the keystream and by solving the below simultaneous linear equations we get the feedback polynomial.

$$s_2p_2 + s_1p_1 + s_0p_0 = s_3 \quad (1)$$

$$s_3p_2 + s_2p_1 + s_1p_0 = s_4 \quad (2)$$

$$s_4p_2 + s_3p_1 + s_2p_0 = s_5 \tag{3}$$

The feedback polynomial we get is [3,2] which is passed as argument to generate the keystream to decipher the cipher text.
This is implemented in the *CryptanalysisLib.py* in the function

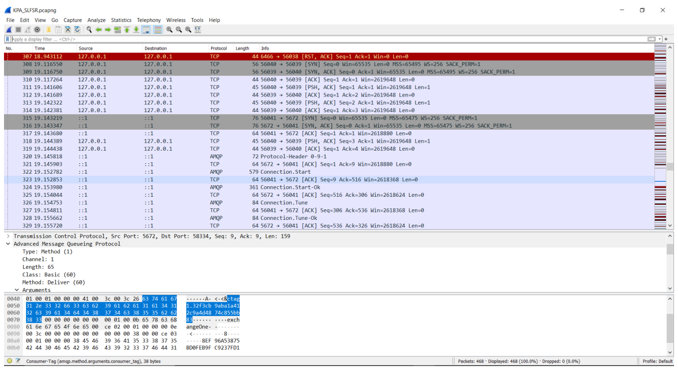
```
def find_sequence_lfsr(randomkeystream, cipherTxt):
   m_value =
        Berlekamp_Massey_algorithm(randomkeystream)
   print("Solving for simultaneous linear equations :")
   ..
   ..
   actualkeystream = CipherGen.simple_LFSR(
      len(cipherTxt), m_stream, polynomial, True)
   return actualkeystream
```

- **Attack on Trivium**:The feedforward paths involving the AND operation are crucial for the security of Trivium as they prevent attacks that exploit the Stream linearity of the cipher, as the one applicable to plain LFSRs.So Berelkamp-Massey algorithm will not break Trivium.

C. Wireshark Packet Capture

We use the Wireshark tool to capture the Rabbit MQ traffic and analyze the working of the AMQP message protocol with TCP/IP as the underlying transport layer protocol.

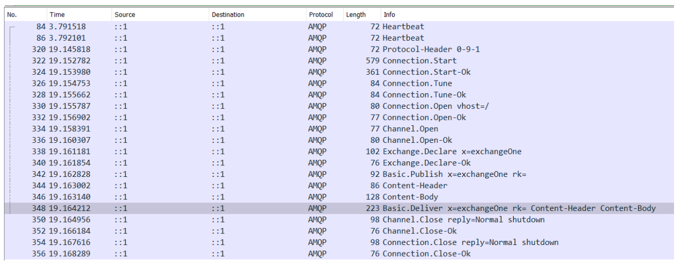
Fig. 12 is a screenshot of a typical wireshark capture file.


KPA_SLFSR.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.    Time       Source     Destination  Protocol Length Info
307 18.943112    127.0.0.1  127.0.0.1    TCP        44 6466 → 56038 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
308 19.116550    127.0.0.1  127.0.0.1    TCP        56 56040 → 56039 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
309 19.116750    127.0.0.1  127.0.0.1    TCP        56 56039 → 56040 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
310 19.117264    127.0.0.1  127.0.0.1    TCP        44 56040 → 56039 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
311 19.141606    127.0.0.1  127.0.0.1    TCP        45 56040 → 56039 [PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=1
312 19.141689    127.0.0.1  127.0.0.1    TCP        44 56039 → 56040 [ACK] Seq=1 Ack=2 Win=2619648 Len=0
313 19.142322    127.0.0.1  127.0.0.1    TCP        45 56040 → 56039 [PSH, ACK] Seq=2 Ack=1 Win=2619648 Len=1
314 19.142381    127.0.0.1  127.0.0.1    TCP        44 56039 → 56040 [ACK] Seq=1 Ack=3 Win=2619648 Len=0
315 19.143219    ::1        ::1          TCP        76 56041 → 5672 [SYN] Seq=0 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM=1
316 19.143347    ::1        ::1          TCP        76 5672 → 56041 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM=1
317 19.143680    ::1        ::1          TCP        64 56041 → 5672 [ACK] Seq=1 Ack=1 Win=2618880 Len=0
318 19.144389    127.0.0.1  127.0.0.1    TCP        45 56040 → 56039 [PSH, ACK] Seq=3 Ack=1 Win=2619648 Len=1
319 19.144438    127.0.0.1  127.0.0.1    TCP        44 56039 → 56040 [ACK] Seq=1 Ack=4 Win=2619648 Len=0
320 19.145818    ::1        ::1          AMQP       72 Protocol-Header 0-9-1
321 19.145903    ::1        ::1          TCP        64 5672 → 56041 [ACK] Seq=1 Ack=9 Win=2618880 Len=0
322 19.152782    ::1        ::1          AMQP      579 Connection.Start
323 19.152853    ::1        ::1          TCP        64 56041 → 5672 [ACK] Seq=9 Ack=516 Win=2618368 Len=0
324 19.153980    ::1        ::1          AMQP      361 Connection.Start-Ok
325 19.154044    ::1        ::1          TCP        64 5672 → 56041 [ACK] Seq=516 Ack=306 Win=2618624 Len=0
326 19.154753    ::1        ::1          AMQP       84 Connection.Tune
327 19.154811    ::1        ::1          TCP        64 56041 → 5672 [ACK] Seq=306 Ack=536 Win=2618368 Len=0
328 19.155662    ::1        ::1          AMQP       84 Connection.Tune-Ok
329 19.155720    ::1        ::1          TCP        64 5672 → 56041 [ACK] Seq=536 Ack=326 Win=2618624 Len=0

> Transmission Control Protocol, Src Port: 5672, Dst Port: 58334, Seq: 9, Ack: 9, Len: 159
v Advanced Message Queueing Protocol
    Type: Method (1)
    Channel: 1
    Length: 65
    Class: Basic (60)
    Method: Deliver (60)
  v Arguments

0040  01 00 01 00 00 00 41 00  3c 00 3c 26 63 74 61 67   ······A· <·<&ctag
0050  31 2e 33 32 66 33 63 62  39 61 62 61 31 61 34 31   1.32f3cb 9aba1a41
0060  32 63 39 61 34 64 34 38  37 34 63 38 35 35 62 62   2c9a4d48 74c855bb
0070  38 33 00 00 00 00 00 00  00 01 00 0b 65 78 63 68   83······ ····exch
0080  61 6e 67 65 4f 6e 65 00  ce 02 00 01 00 00 00 0e   angeOne· ········
0090  00 3c 00 00 00 00 00 00  00 00 00 38 00 00 ce 03   ·<······ ···8····
00a0  00 01 00 00 00 38 45 46  39 36 41 35 33 38 37 35   ·····8EF 96A53875
00b0  42 44 30 46 45 42 39 46  43 39 32 33 37 46 44 31   BD0FEB9F C9237FD1

Consumer-Tag (amqp.method.arguments.consumer_tag), 38 bytes    Packets: 468 · Displayed: 468 (100.0%) · Dropped: 0 (0.0%)    Profile: Default
```

Fig. 12. Wireshark Capture

When running the Wireshark tool we see a lot of different type of packets and messages deliverd with different protcols. We can analyse the AMQP messages by applying the AMQP filter.



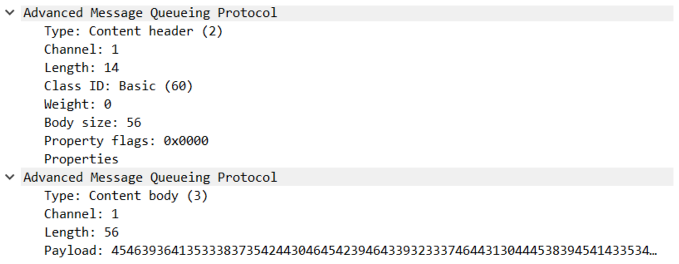
```
No.    Time       Source  Destination  Protocol  Length Info
 84 3.791518     ::1     ::1          AMQP        72 Heartbeat
 86 3.792101     ::1     ::1          AMQP        72 Heartbeat
320 19.145818    ::1     ::1          AMQP        72 Protocol-Header 0-9-1
322 19.152782    ::1     ::1          AMQP       579 Connection.Start
324 19.153980    ::1     ::1          AMQP       361 Connection.Start-Ok
326 19.154753    ::1     ::1          AMQP        84 Connection.Tune
328 19.155662    ::1     ::1          AMQP        84 Connection.Tune-Ok
330 19.155787    ::1     ::1          AMQP        80 Connection.Open vhost=/
332 19.156902    ::1     ::1          AMQP        77 Connection.Open-Ok
334 19.158391    ::1     ::1          AMQP        77 Channel.Open
336 19.160307    ::1     ::1          AMQP        80 Channel.Open-Ok
338 19.161181    ::1     ::1          AMQP       102 Exchange.Declare x=exchangeOne
340 19.161854    ::1     ::1          AMQP        76 Exchange.Declare-Ok
342 19.162828    ::1     ::1          AMQP        92 Basic.Publish x=exchangeOne rk=
344 19.163002    ::1     ::1          AMQP        86 Content-Header
346 19.163140    ::1     ::1          AMQP       128 Content-Body
348 19.164212    ::1     ::1          AMQP       223 Basic.Deliver x=exchangeOne rk= Content-Header Content-Body
350 19.164956    ::1     ::1          AMQP        98 Channel.Close reply=Normal shutdown
352 19.166184    ::1     ::1          AMQP        76 Channel.Close-Ok
354 19.167616    ::1     ::1          AMQP        98 Connection.Close reply=Normal shutdown
356 19.168289    ::1     ::1          AMQP        76 Connection.Close-Ok
```

Fig. 13. AMQP messages

Packet List provides a summary of protocol frames and methods exchanged by a client and a RabbitMQ node. The Info column indicates the Class and Method (e.g. Basic.Publish) and then the most significant arguments. For example:

- Connection.Open vhost=(vhost)
- Connection.Close reply=(reply-text)
- Channel.Close reply=(reply-text)
- Exchange.Declare x=(exchange-name)
- Basic.Publish x=(exchange-name) rk=(routing-key)
- Basic.Deliver x=(exchangeOne) rk=(Content-Header Content-Body)

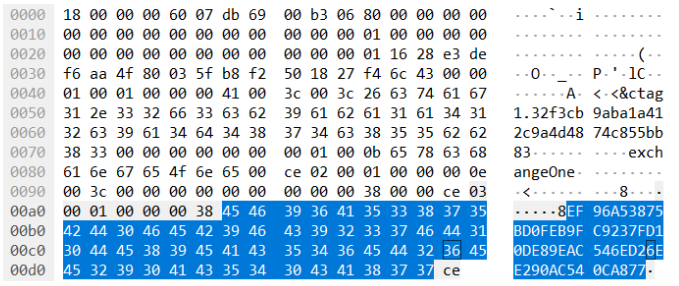
One of the important packets captured is Basic.Deliver, and looking at Packet Details indicate all arguments of the frame. It also includes dynamically calculated values enclosed in square brackets.



```
v Advanced Message Queueing Protocol
    Type: Content header (2)
    Channel: 1
    Length: 14
    Class ID: Basic (60)
    Weight: 0
    Body size: 56
    Property flags: 0x0000
    Properties
v Advanced Message Queueing Protocol
    Type: Content body (3)
    Channel: 1
    Length: 56
    Payload: 454639364135333837354244304645423946433932333746443130444538394541433534…
```

Fig. 14. Header and Body of AMQP message

The Packet-Bytes view of the Wireshark provides more details about the content body.Here we can see that the Frame 348, has 223 bytes (1784 bits). We can also observe that the payload transferred is encrypted and cannot be easily understood until the attacker knows the algorithm used.



```
0000  18 00 00 00 60 07 db 69  00 b3 06 80 00 00 00 00   ····`··i ········
0010  00 00 00 00 00 00 00 00  00 00 00 01 00 00 00 00   ········ ········
0020  00 00 00 00 00 00 00 00  00 00 00 01 16 28 e3 de   ········ ·····(··
0030  f6 aa 4f 80 03 5f b8 f2  50 18 27 f4 6c 43 00 00   ··O··_·· P·'·lC··
0040  01 00 01 00 00 00 41 00  3c 00 3c 26 63 74 61 67   ······A· <·<&ctag
0050  31 2e 33 32 66 33 63 62  39 61 62 61 31 61 34 31   1.32f3cb 9aba1a41
0060  32 63 39 61 34 64 34 38  37 34 63 38 35 35 62 62   2c9a4d48 74c855bb
0070  38 33 00 00 00 00 00 00  00 01 00 0b 65 78 63 68   83······ ····exch
0080  61 6e 67 65 4f 6e 65 00  ce 02 00 01 00 00 00 0e   angeOne· ········
0090  00 3c 00 00 00 00 00 00  00 00 00 38 00 00 ce 03   ·<······ ···8····
00a0  00 01 00 00 00 38 45 46  39 36 41 35 33 38 37 35   ·····8EF 96A53875
00b0  42 44 30 46 45 42 39 46  43 39 32 33 37 46 44 31   BD0FEB9F C9237FD1
00c0  30 44 45 38 39 45 41 43  35 34 36 45 44 32 36 45   0DE89EAC 546ED26E
00d0  45 32 39 30 41 43 35 34  30 43 41 38 37 37 ce      E290AC54 0CA877·
```

Fig. 15. Frame and Payload

VI. Conclusion

The main goal of this paper was to study and analyze the security of the Stream Ciphers such as Simple LFSR based stream cipher and Trivium. Further, in this paper, we study the internal structure of LFSR and Trivium and implement both algorithms on python . In the last two decades, and especially in the last five years, lightweight cryptography has gotten a lot of attention. In this work, we compare Trivium with a basic LFSR-based stream cipher to provide an unbiased and thorough examination of the state-of-the-art in lightweight cryptography.

Based on the Known-plaintext on simple LFSR, it is clear that LFSRs by themselves are extremely insecure. However, Trivium is robust and it is based on a combination of three shift registers. Even though these are feedback shift registers, there are nonlinear components used to derive the output of each register, unlike the LFSRs. This makes it impossible for Known-plain-text attack on Trivium.

According to the literature study, cube attack is a relatively modern cryptanalysis approach, and its applicability to various new stream ciphers such as Trivium is important [14] [15] [24].

A. Future Work

1) VTR Implemenation: The Verilog-to-Routing (VTR) project is a world-wide collaborative effort to provide a open-source framework for conducting FPGA architecture and CAD research and development. The VTR design flow takes as input a Verilog description of a digital circuit, and a description of the target FPGA architecture [17].

Trivium is a class of cipher that aims to make the best use of hardware resources. The scope of the future work is to implement and compile the verilog version of trivium into VTR. As mentioned VTR is one of the most popular academic and open source framework for understanding the FPGA workflow.

It has many rich features and our aim would be to find the right architecture of the FPGA that will give the best Area-Delay performance for Trivium.

References

[1] Obe, B. B. (2018, September 24). Light-weight Cryptography: Trivium - ASecuritySite: When Bob Met Alice. Medium. https://medium.com/asecuritysite-when-bob-met-alice/light-weight-cryptography-trivium-aa986f0b881

[2] Wikipedia contributors. (2021, July 7). A5/1. Wikipedia. https://en.wikipedia.org/wiki/A5/1

[3] Two Trivial Attacks on A5/1: A GSM Stream Cipher https://arxiv.org/vc/arxiv/papers/1305/1305.6817v1.pdf

[4] A New Guess-and-Determine Attack on the A5/1 Stream Cipher

[5] Erlang Programming Language https://www.erlang.org/

[6] A. Foster, “Messaging technologies for the industrial internet and the internet of things whitepaper,” PrismTech, 2015

[7] Messaging that just works — RabbitMQ. (n.d.). RabbitMQ. Retrieved August 15, 2021, from https://www.rabbitmq.com/

[8] What is RabbitMQ? — A Quick Glance of What is RabbitMQ? https://www.educba.com/what-is-rabbitmq/

[9] Wireshark - Wikipedia https://en.wikipedia.org/wiki/Wireshark https://www.rabbitmq.com/tutorials/tutorial-one-python.html

[10] RabbitMQ tutorial - Publish/Subscribe — RabbitMQ. (n.d.). RabbitMQ. Retrieved August 15, 2021, from https://www.rabbitmq.com/tutorials/tutorial-three-python.html

[11] Fanout Exchange — RabbitMQ Exchange Types — RabbitMQ Tutorial (codedestine.com) https://codedestine.com/rabbitmq-fanout-exchange/

[12] Islam, Saad & Ul Haq, Inam. (2016). Cube attack on Trivium and A5/1 stream ciphers. 409-415. 10.1109/IBCAST.2016.7429911.

[13] Bedi, S.S., Pillai, N.R.: Cube Attacks on Trivium. Cryptology ePrint Archive, Report 2009/015 (2009) http://eprint.iacr.org/

[14] Sandip Karmakar, Prasanna Mishra, Navneet Gaba & Dipanwita Roy Chowdhury (2018) An algebraic cryptanalysis tool for cube attack on symmetric ciphers, Journal of Information and Optimization Sciences, 39:6, 1231-1243, DOI: 10.1080/02522667.2017.1317957

[15] B. Zhu, W. Yu, and T. Wang, “A practical platform for cube-attack-like cryptanalyses,” Cryptology ePrint Archive, Report 2010/644, 2010,http://eprint.iacr.org/

[16] Zhu, B. (n.d.). An Online Calculator of Berlekamp-Massey Algorithm. An Online Calculator of Berlekamp-Massey Algorithm. Retrieved August 15, 2021, from https://bma.bozhu.me/

[17] VTR — Verilog-to-Routing 8.1.0-dev documentation. (n.d.). VTR — Verilog-to-Routing 8.1.0-Dev Documentation. Retrieved August 15, 2021, from https://docs.verilogtorouting.org/en/latest/vtr/

[18] Communication theory of secrecy systems. Tech. Report 28. Bell Laboratories, Inc.. 1949.

[19] Stream Ciphers - an overview — ScienceDirect Topics. (n.d.). Sciencedirect. Retrieved August 15, 2021, from https://www.sciencedirect.com/topics/computer-science/stream-ciphers

[20] Paar, Christof, 1963-. Understanding Cryptography : a Textbook for Students and Practitioners. Berlin ; London :Springer, 2009.

[21] Dinur, I., Shamir, A.: Breaking Grain-128 with Dynamic Cube Attacks. Cryptology ePrint Archive, Report 2010/570 (2010) http://eprint.iacr.org/.

[22] Wikipedia contributors. (2021b, August 9). Brute-force attack. Wikipedia. https://en.wikipedia.org/wiki/Brute-force_attack.

[23] Afzal, M., & Masood, A. (2008). Algebraic analysis of Trivium and Trivium/128. International Journal of Electronic Security and Digital Forensics, 1(4), 344. https://doi.org/10.1504/ijesdf.2008.021452.

[24] Mroczkowski, P., & Szmidt, J. (2012). The Cube Attack on Stream Cipher Trivium and Quadraticity Tests. Fundamenta Informaticae, 114(3–4), 309–318. https://doi.org/10.3233/fi-2012-631.

[25] Wikipedia contributors. (2021a, April 13). Berlekamp–Massey algorithm. Wikipedia. https://en.wikipedia.org/wiki/Berlekamp

[26] Martin-Navarro, J. L., & Fúster-Sabater, A. (2021). Review of the Lineal Complexity Calculation through Binomial Decomposition-Based Algorithms. Mathematics, 9(5), 478. https://doi.org/10.3390/math9050478

[27] Klapper, A., & Goresky, M. (1997). Feedback shift registers, 2-adic span, and combiners with memory. Journal of Cryptology, 10(2), 111–147. https://doi.org/10.1007/s001459900024