សាកលវិទ្យាល័យភូមិន្ទភ្នំពេញ
ROYAL UNIVERSITY OF PHNOM PENH

A Study Research on Security Measure on IoT Device: New Cipher Algorithm

A Research Report
In Partial Fulfilment of the Requirement for the Degree of
Bachelor of Science in Information Technology Engineering

BUN NICOLAS

August 2021

សាកលវិទ្យាល័យភូមិន្ទភ្នំពេញ
ROYAL UNIVERSITY OF PHNOM PENH

A Study Research on Security Measure on IoT Device: New Cipher Algorithm

A Research Report
In Partial Fulfilment of the Requirement for the Degree of
Bachelor of Science in Information Technology Engineering

BUN NICOLAS

Examination Committee: Dr. SRUN SOVILA (Head of Department ITE)
Mr. NA SAMBATHCHATOVONG

August 2021

ABTRACT

In recent years, people are tied to internet. Unless you’re in a remote area or an old age. The Internet of Things (IoT) and the Smart Home topic being on the spot everywhere and have grown in importance. Smart Light Bulbs, Smart TVs, Smart Door Locks, and other smart home products are becoming increasingly popular, which is leading to an increase in vulnerabilities. Furthermore, there are various software and hardware vulnerabilities that complicate and complicate the security situation. Many of these systems and devices also process personal or confidential information and manage important industrial activities. Security is critical. The work of safeguarding current IoT devices is frequently overburdened for owners and administrators. Today, the range of Smart Home technologies is expanding at a quicker rate than security can be assured. Consumers' security and privacy are jeopardized by unprotected vulnerabilities. This paper aims to study and research on security vulnerability with common attacks. Consequently, the outcome of this research work will provide useful implementation to decrease the likelihood of vulnerability in IoT Device communication.

SUPERVISOR'S RESEARCH SUPERVISION STATEMENT

TO WHOM IT MAY CONCERN

Name of program: Bachelor of Information Technology Engineering
Name of candidate: BUN NICOLAS

Title of research report: A Study Research on Security Measure on IoT Device: New Cipher Algorithm

This is to certify that the research carried out for the above titled master's research report was completed by the above-named candidate under my direct supervision. This thesis material has not been used for any other degree. I played the following part in the preparation of this reacher report: ……..
……
……

Supervisor's name: Mr. NA SAMBATHCHATOVONG

Supervisor's signature: ……

Date: ……

CANDIDATE'S STATEMENT

TO WHOM IT MAY CONCERN

This is to certify that the research report that I Bun Nicolas hereby present entitled "A Study Research on Security Measure on IoT Device: New Cipher Algorithm"

for the degree of Bachelor of Engineering at the Royal University of Phnom Penh is entirely my own work and, furthermore, that it has not been used to fulfill the requirements of any other qualification in whole or in part, at this or any other University or equivalent institution.

No reference to, or quotation from, this document may be made without the written approval of the author.

Signed by **Bun Nicolas**: ……

Date: …….

Sign by Supervisor: ……..

Supervisor's signature: ……..…

Date……...……

Sign by Supervisor: ……..…..

Supervisor's signature: ……...…

Date……..…

ACKNOWLEGEMENTS

In this research cannot finished without support and encouragement from many different people in their different ways. I would like to show my deepest appreciation to the following people.

First, I would like to show my gratitude and deep regards to my supervisor, Mr. Na Sambathchatovong, who has supported me throughout my report with his guidance, comment, feedbacks, and encouragement. The help and guidance given by him from time to time will make me to complete this report.

Then I would like to show my thanks to all my lecturers during the Bachelor program at Royal University of Phnom Penh, for their strong knowledge foundation in their respective subjects to help me advancing my knowledge.

Finally, I would like to show my deepest and warmest thanks to my beloved family. Especially my parents for their support and love that make all things possible for me in this world.

Table of Contents

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 Background of the Study

Nowadays, the number of interconnected smart devices, typically referred to as “Internet of Things” (IoT) devices, is massively growing due to almost endless applications, which are quickly getting part of the everyday life, such as wearable technologies, smart buildings, or health monitoring devices. IoT devices are typically based on resource con- strained embedded processors, such as the family of ARM Cortex-M processors, and are equipped with sensing and short- range communication capabilities, e.g. Bluetooth Low Energy (BLE) modules. These devices typically collect social and environmental (potentially sensitive) data from their sensors and send them over to a remote server, located in the Internet cloud, through an IoT-Internet gateway.

The term "Internet" refers to the high-level name of protocols and applications, and is based on the more advanced related networks. It provides assistance to countless human beings on all continents 24 hours a day, 7 days a week [1]. From 1990 to the present, the development and revolution of the Internet has been very rapid, from the network environment to the real-world object, that is, the Internet of Things (IoT) [2]. Internet of Things is a network of physical devices that can detect and collect data from the external environment. IoT devices can exist in different geographic locations, and the collected data can be used for different applications in the medical, educational, banking, and other fields. The Internet of Things technology is a modern standard whose scope is quickly realized by combining communication and model sets. The main concepts of the Internet of Things include various objects such as emergency mechanisms (Bluetooth, RFID, sensors) and mobile phones. You are eligible to communicate with other mechanisms via the Internet. The rapid improvement of the Internet of Things can help people make their situations smarter and more calculable, thereby overcoming daily problems. All in all, the Internet of Things provides new opportunities for remote monitoring and management of equipment, so decisions can be made based on information received from various real-time traffic data streams [3, 4, 5].

1.2 Problem Statement

Generally, security problems in all scopes of IoT applications are considered as the first important huge challenges. Furthermore, in IoT environment, every individual equipment and sensor that are established, deployed, and synchronized may be suffering from attackers at any position. In many IoT applications, data of node sensor are extremely critical and have to be encrypted. Current encryption procedures are associated with a large complexity of

arithmetical processes to provide a high level of security. This arithmetic complexity, of course, results in consuming extra time and power. On the other hand, the sensors are suffering from limitations of storage space and power, which affect the transmission efficiency. One powerful solution to achieve high security with the above-mentioned limitations is the use of lightweight encryption technology. Recently, Panagiotou et al. [3] encouraged using stream ciphers as lightweight cryptography techniques for sensitive data in IoT devices.

The sensor, the aggregator, the channel, the external utility, and the decision trigger are the five components of the IoT architecture [6]. Sensors are electronic devices that measure physical qualities (temperature, weight, position and others). An aggregator is a piece of software that helps you turn raw data into composite data. The transmission channel is the medium through which the collected data is transmitted (USB, wireless, wired or verbal). An external utility (software or hardware) created for the aim of archiving. The end aim of the IoT is conceptually defined by decision triggers. In most circumstances, treating the decision trigger as an if-then rule makes sense [6].

1.3 Objective of the Study

The aim of this study is to develop a defensive theory that will be utilized to reduce the likelihood of security risks like broken authentication or even prevent it altogether. The objectives of the research are described as follows:

- Briefly understanding of IoT Device: Smart Light
- Review all related work on encryption algorithm
- Come up with a theory which reduce the likelihood of security risk of broken authentication

1.4 Scope and Limitation

Our Scope is focusing on:

- Study on new encryption algorithm
- Producing theory with the new algorithm
- The use of base JWT Framework to implement test encryption algorithm

1.5 Structure of Study

The following is a breakdown of the report's structure. We begin with Introduction to IoT. In Chapter 2, we present definition of Smart Light, types, types of common attacks, Security Analysis of Salsa & ChaCha20. In chapter 3, the methodology of our ChaCha20-Poly1305 and integrate ChaCha20 into JWT with custom algorithm & custom library. In Chapter 4, contain of the success result implementation test. Lastly, the chapter 5 is the conclusion.

CHAPTER 2

LITERACTURE REVIEW

2.1 Definition of IoT Device Smart Light

The Internet of Things (IoT) is a broad phrase that refers to the interconnection of many technologies. This phrase refers to technologies that enable the networking of diverse elements from various domains of global infrastructure and hence their collaboration. The Internet of Things, as used in this study, refers to the virtual representation of clearly identifiable physical items (things) on the Internet, or a network similar to or equal to the Internet.

The word Smart Home is a collective term. As a result, the term Smart Home refers to technological processes and systems in living spaces and dwellings (Home) whose primary goal is to improve quality of life. The term "smart home" is defined in this paper in this meaning. It refers to equipment that are designed for a human's living area and enable tasks such as energy efficiency, the networking of household components, and even remote control. Smart light is sub-definition of smart home, with the use of a light bulb along with remote a switch or relay of local connection such Bluetooth or connecting through hub, allowing this device to connect to internet, thus remoting from far away.

Smart Light types

According to Phillips and Jorge Higuera on Smart light trend. There are three smart light types on the market:

- Local connected light bulb
- Smart Light Bulb(standalone)
- Smart Light with hub

IoT and Smart light security attack

IoT Device such as Smart Light has under these two-common attacks:

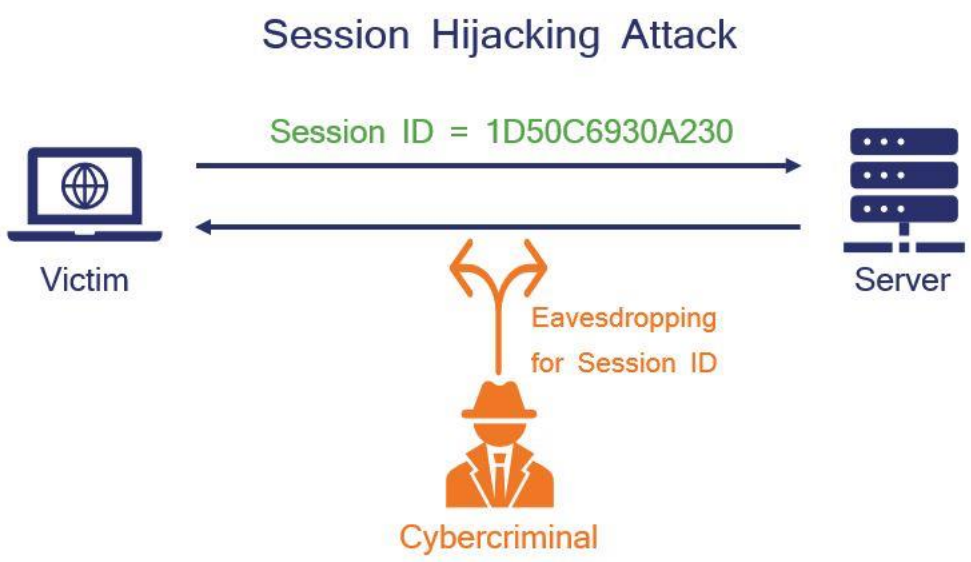


Figure 2.1a Broken Authentication

The attack is active in Eavesdropping to snoop the Session ID, imitated as original user/client to authorized into resource server.

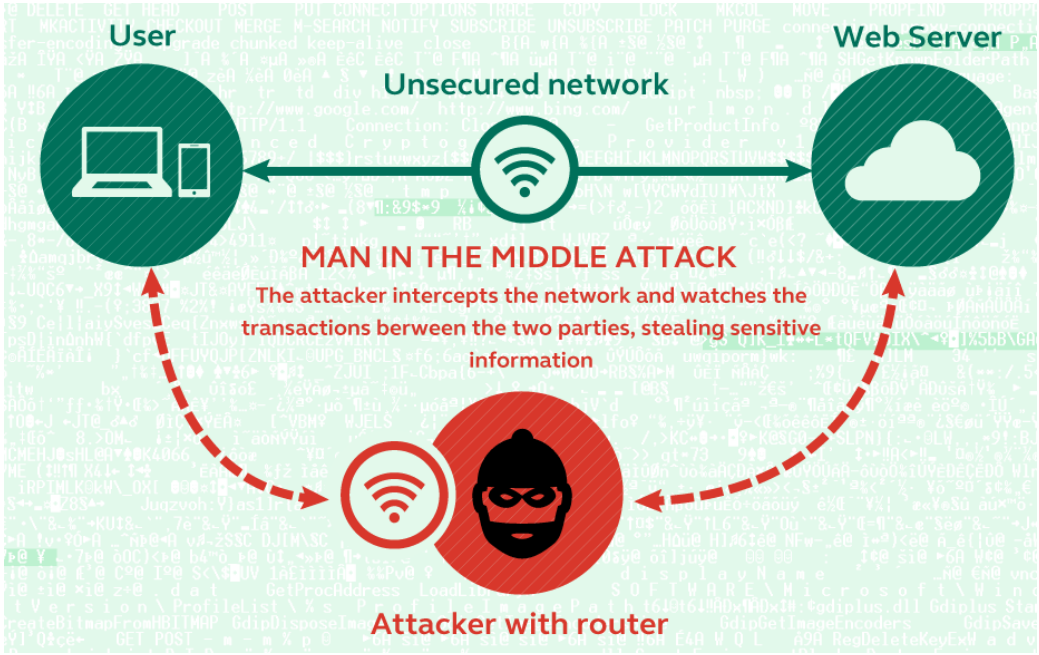


Figure 2.1b Man in the middle attack

2.2 Security Analysis of Chacha20-poly1305

This report presents a security assessment of ChaCha20Poly1305 Authentic Resolved Encryption with Associative Data (AEAD). The combination of ChaCha and Poly1305 has been found to be a secure authenticated encryption scheme assuming that ChaCha and Poly1305 are secure algorithms. In addition, Poly1305 was found to be ε-almost-Δ-universal, that is, a secure universal hash function. Therefore, we assess the security of ChaCha against existing attacks.

We have shown that there is no efficient differential analysis, linear cryptographic analysis, discriminant attack, analytical conjecture and determinism, algebraic attacks and anti-initialization attacks. The Time Memory Data intrusion attack and the side-channel attack apply to ChaCha; however, we can deal with these attacks with practical countermeasures. We therefore conclude that we were unable to identify any weaknesses in ChaCha20Poly1305 AEAD.

Chacha20

ChaCha [15] is a 32-bit word-based algorithm that accepts a 256-bit key K = (k0,k1,k2,k3,k4,k5,k6,k7) and a 32-bit counter C = (c0) as input and outputs 512-bit keystream blocks1. The 4x4 matrix of 32-bit words.

Bernstein proposed the Poly1305 [16] cryptographic message authentication code (MAC). Poly1305 takes a 256-bit one-time key and an arbitrary-length message as input. A 128-bit tag is the result.

Chacha20 is the iteration version of Salsa20, in short, an updated version of Salsa20. The Salsa20 is known as base foundation, the concept of lightweight symmetric encryption, the chacha20 based of.

The quarter-round function of ChaCha, like that of Salsa20, has four additions, four exclusive-ors, and four constant-distance rotations of 32-bit words. In comparison to Salsa20, ChaCha increases the amount of diffusion in quarter-round functions. The quarter round function of ChaCha and Salsa20 is shown in Figures 3.1 and 3.2. In the ChaCha quarter-round function, each input word is updated twice, but in the Salsa20 quarter-round function, each word is only updated once. In addition, the ChaCha quarter-round function, like the Salsa20 quarter-round function, allows each input word to affect each output word.

Cryptanalysis of Salsa 20, ChaCha20

Salsa20: Multiple works of attacked on Salsa20/6, Salsa20/7 & Salsa20/8 of 4 round differentials. The probability rate of attack success were 50%. They used four 4-round differentials as the chaining distinguisher in their attack on 256-bit Salsa20/6, which requires

273 operations and 216 keystreams and requires two rounds backwards from Salsa20/6. They used the same four-round differential that Aumasson et al. used to attack 256-bit Salsa20/7. A total of 2148 operations and 224 keystreams are required for this attack. With a time complexity of 2250 and a data complexity of 227, the same differential was used to attack the 256-bit Salsa20/8. Finally, they demonstrated an attack on 128-bit Salsa20/7 requiring 2109 operations and 219 keystreams using the same 4-round differential. In 2015, [17] Maitra re-examined the single bit differentials for 4-round Salsa and discovered more significant biases than previously reported. He also looked at the PNB technique used by Aumasson et al. [18] and found that in practice, the median of certain biases is 4 times higher than what Aumasson found.

ChaCha20: Because ChaCha and Salsa20 have a similar design, the cryptanalysis discussed above also applies to ChaCha. However, because ChaCha is designed to achieve rapid diffusion, it is said to be more difficult to break than Salsa. The attacks on Salsa and ChaCha that have been published are proof of this. In Cache Timing Attack; tested on a wide range of CPUs, ChaCha implementation takes constant time. Because ChaCha does not include variant-time operations like S-box, the execution time is input-independent. As a result, analyzing cache timing against ChaCha is as difficult as cryptanalysis of the ChaCha output. On the other hand, Side Channel Attack has least success attacked, while Fault Injection posed a concern on resource restriction such as RISC architecture with Cortex-M0, M3, M4.

Procter [19] demonstrated that the combination of ChaCha and Poly1305 is a secure authenticated encryption scheme assuming ChaCha is a pseudo-random function (PRF), and Poly1305 is ε-almost-Δ-universal. It's shows that the outline of the security model and proof, where demonstrated as Game 0 & 1, Game 1 & 2, Game 2 & 3, Game 3 & 4. It's shown that an adversary can't tell the difference between these two games, and can't tell the difference between ChaCha and a function chosen at random.

From the assessed of the security of ChaCha20-Poly1305 AEAD. According to Procter [19], ChaCha20-Poly1305 AEAD is secure if both ChaCha and Poly1305 are secure.

The table shown below comparison of Existing attacks on Salsa and ChaCha:

Cipher	Round/Key length	Time	Data	Reference
Salsa20	5/256	2^{165}	2^{6}	Crowley [Cro06]
		2^{167}	2^{7}	Velichkov et al. [VMCP12]
		2^{55}	2^{10}	Shi et al. [SZFW12]
		2^{8}	2^{8}	Choudhuri [CM17]
	6/256	2^{177}	2^{16}	Fischer et al. [FMB+06]
		2^{73}	2^{16}	Shi et al. [SZFW12]
		2^{32}	2^{32}	Choudhuri and Maitra [CM17]
	7/128	2^{111}	2^{21}	Aumasson et al. [AFK+08]
		2^{109}	2^{19}	Shi et al. [SZFW12]
	7/256	2^{190}	$2^{11.4}$	Tsunoo et al. [TSK+07]
		2^{151}	2^{26}	Aumasson et al. [AFK+08]
		2^{148}	2^{24}	Shi et al. [SZFW12]
		2^{139}	2^{32}	Choudhuri and Maitra [CM17]
		2^{137}	2^{61}	Choudhuri and Maitra [CM17]
	8/256	2^{255}	$2^{11.4}$	Tsunoo et al. [TSK+07]
		2^{251}	2^{31}	Aumasson et al. [AFK+08]
		2^{250}	2^{27}	Shi et al. [SZFW12]
		$2^{247.2}$	2^{27}	Maitra et al. [MPM15]
		$2^{245.5}$	2^{96}	Maitra [Mai16]
		$2^{244.9}$	2^{96}	Choudhuri and Maitra [CM17]
ChaCha	4/256	2^{6}	2^{6}	Choudhuri and Maitra [CM17]
	4.5/256	2^{12}	2^{12}	Choudhuri and Maitra [CM17]
	5/256	2^{16}	2^{16}	Choudhuri and Maitra [CM17]
	6/128	2^{107}	2^{30}	Aumasson et al. [AFK+08]
		2^{105}	2^{28}	Shi et al. [SZFW12]
	6/256	2^{139}	2^{30}	Aumasson et al. [AFK+08]
		2^{136}	2^{28}	Shi et al. [SZFW12]
		2^{130}	2^{35}	Choudhuri and Maitra [CM17]
		$2^{127.5}$	$2^{37.5}$	Choudhuri and Maitra [CM17]
		2^{116}	2^{116}	Choudhuri and Maitra [CM17]
	7/256	2^{248}	2^{27}	Aumasson et al. [AFK+08]
		$2^{246.5}$	2^{27}	Shi et al. [SZFW12]
		$2^{238.9}$	2^{24}	Maitra [Mai16]
		$2^{233.7}$	2^{96}	Choudhuri and Maitra [CM17]
		2^{233}	2^{28}	Choudhuri and Maitra [CM17]

Figure 2.2 Comparison of existing attack

Performance of AES128 GCM and ChaCha20-Poly1305

In the study, on ARM Cortex-M4 processors, it's demonstrated high-speed, constant-time, and compact implementations of the ChaCha20 stream cipher, Poly1305-ChaCha20 authenticator, and ChaCha20-Poly1305 authenticated encryption scheme. According to RFC7905 and RFC7539, the ChaCha20 and Poly1305 ciphers are among the most promising alternatives for securing upcoming IoT applications with strict performance and space constraints, and for supporting TLS protected communication on ARM Cortex-M4 devices.

TABLE I
ENCRYPTION OF 64-BYTE INPUT DATA. MAC OF 128-BYTE INPUT DATA. AEAD OF 16-BYTE INPUT DATA AND 16-BYTE ASSOCIATED DATA.

	Platform	Algorithm	Runtime [Cycles]	Speed‡ [Cycles/Byte]	Size [Byte]	Stack [Byte]
Encryption	8-bit AVR ATmega [9]	Salsa20	17,787	268.0	–	268
	32-bit ARM Cortex-M4 [10]	Salsa20	3,311	–	1,272	552
	32-bit ARM Cortex-M0 [11]	ChaCha20	–	39.9	–	–
	32-bit ARM Cortex-M4 [10]	ChaCha20	3,468	–	1,328	544
	32-bit ARM Cortex-M4 [12]†	ChaCha20	1,287	17.6	3,174	228
	32-bit ARM Cortex-M4 (This Work)	**ChaCha20**	**1,487**	**20.6**	**734**	**232**
MAC	32-bit ARM Cortex-M0 [13]	Chaskey	–	18.3	1,308	–
	32-bit ARM Cortex-M4 [13]	Chaskey	–	7.0	908	–
	8-bit AVR ATmega [9]	Poly1305	–	195.0	–	148
	32-bit ARM Cortex-M4 (This Work)	**Poly1305**	**747**	**3.6**	**744**	**120**
	32-bit ARM Cortex-M4 (This Work)	**Poly1305-ChaCha20**	**1,945**	**3.6**	**1,322**	**224**
AEAD	32-bit ARM Cortex-M4 [10]	AES128-GCM	43,657	–	2,644	812
	32-bit ARM Cortex-M4 [10]	AES128-EAX	32,159	–	2,780	932
	32-bit ARM Cortex-M4 [10]	AES128-CCM	23,949	–	2,256	780
	32-bit ARM Cortex-M4 [10]	NORX32	6,855	–	1,820	320
	32-bit ARM Cortex-M4 (This Work)	**ChaCha20-Poly1305**	**3,364**	**28.4**	**1,946**	**332**

† These results were obtained by including the π-ChaCha20 function (cf. https://github.com/joostrijneveld/chacha-arm-cortex-m/) in our code.
‡ Asymptotic values.

Fig. 2.3 Table of ChaCha20 & AES 128 GCM performance cycle

2.3 Related Works

The ChaCha ciphers are a family of symmetric stream ciphers proposed by D. J. Bernstein [10] as an evolution of the earlier published Salsa ciphers [11]. On numerous studies, there're research upon research on ChaCha20 from year to year on security s well updating with the proposed algorithm. Salsa20/12 and ChaCha12 are sufficient for 256-bit keys under the attack model. With ChaCha20 is 8 more rounds than ChaCha12. It is a more conservative design than AES, as evidenced by cryptanalysis papers of the Salsa and ChaCha ciphers [10], [11]. It provides a similar or better level of security than AES. ChaCha is an Add-Rotate-XOR cipher, which means that all of the ChaCha algorithms' operations are based on 32bit data word additions, XOR operations, and rotations (cyclic bit shifts). The ChaChaN cipher family has three standardized variations, each with a different number of operations done on the cipher state (N = 8, 12, or 20 rounds).

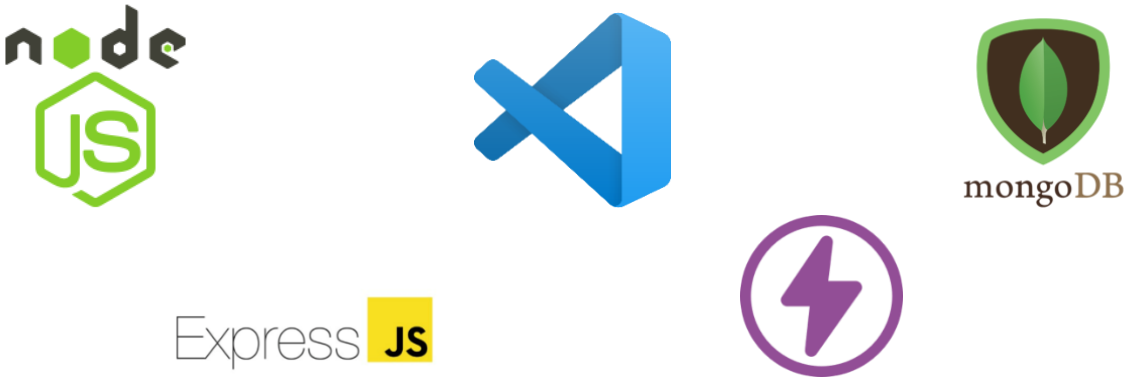
Studies on hardware implementations have been provided for ciphers like AES [14], [12], however publications like [13] largely highlight optimizations for efficient software implementations for ChaCha ciphers. There are three hardware implementations of the ChaCha cipher that we are aware of: Xiphera, Inc. [8] has developed a commercial ChaCha20 implementation. Because this IP core was designed to be a TLS accelerator, it includes a Poly1305 implementation for calculating message authentication codes. It does not, however, allow the number of rounds to be customized, and it does not support ChaCha versions with varied counter or nonce sizes.

[BLB19] The published article is pointing to new mechanism of Ciphersuite, Wireguard. Wireguard is a relatively newcomer to the protocol scene, and it has been praised for permitting speeds comparable to some of the older, less secure protocols while still providing some security advantages. [WRD01] WireGuard uses state-of-the-art cryptography, like the Noise protocol framework, Curve25519, ChaCha20, Poly1305, BLAKE2, SipHash24, HKDF, and secure trusted constructions. It makes conservative and reasonable choices and has been reviewed by cryptographers. The protocol offers strong perfect forward secrecy as well as a high level of identity concealment. ChaCha20Poly1305 authenticated-encryption for packet encapsulation in UDP is used to increase transport speed. In addition, Wireguard foundation on mobile computer such as Smartphone, PC platform providing a new VPN protocol using ChaCha20-Poly1305, similarity to our research. Currently, there's no published paper and articles implemented the encryption on low to medium performance devices such IoT Smart home devices to our research paper.

CHAPTER 3

METHODOLOGY

3.1 Technologies



NodeJS

Node.js is a back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside of a web browser. Node.js allows developers to use JavaScript to write command-line tools and server-side scripting, which involves running scripts on the server to generate dynamic web page content before sending the page to the user's browser. As a result, Node.js represents a "JavaScript everywhere" paradigm,[6] uniting web-app development around a single programming language rather than separate languages for server-side and client-side scripts.

Despite the fact that.js is the standard filename extension for JavaScript code, the term "Node.js" refers to the product rather than a specific file. The event-driven architecture of Node.js allows for asynchronous I/O. These design choices are intended to improve throughput and scalability in web applications with a large number of input/output operations, as well as real-time Web applications (such as real-time communication programs and browser games) .[7]

ExpressJS

Express.js is a framework for running web servers that is built on top of Node.js. It eliminates the need to code complex server-side integrations for Node.js.

Express.js functions as a JavaScript server in a box, allowing you to employ templating solutions and manage your application's routing with understandable and well-organized code.

The only issue with Node.js is that, while it's great for setting up servers, it wasn't created with that application in mind. Express.js can help with that.

MongoDB

MongoDB is a cross-platform document-oriented database program that is open source. MongoDB is a NoSQL database application that works with JSON-like documents and optional schemas. MongoDB is a database that was created by MongoDB Inc. and is distributed under the Server Side Public License (SSPL).

3.2 System Architecture

System Architecture express work flow or process in the web application.

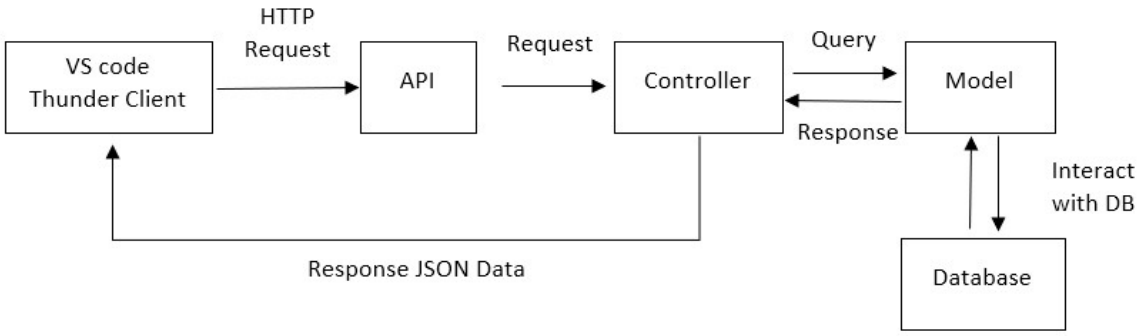


Figure 3.2 System Architecture

3.3 ChaCha20-poly1305 AEAD

The ChaCha Quarter Round

The basic operation of the ChaCha algorithm is the quarter round. It operates on four 32-bit unsigned integers, denoted a, b, c, and d.

The operation is as follows (in C-like notation):

1. a += b; d ^= a; d <<<= 16;
2. c += d; b ^= c; b <<<= 12;
3. a += b; d ^= a; d <<<= 8;

4. c += d; b ^= c; b <<<= 7;

Where "+" denotes integer addition modulo 2^32, "^" denotes a bitwise

Exclusive OR (XOR), and "<<< n" denotes an n-bit left rotation

(towards the high bits).

For example, let's see the add, XOR, and roll operations from the fourth line with sample numbers:

- a = 0x11111111
- b = 0x01020304
- c = 0x77777777
- d = 0x01234567
- c = c + d = 0x77777777 + 0x01234567 = 0x789abcde
- b = b ^ c = 0x01020304 ^ 0x789abcde = 0x7998bfda
- b = b <<< 7 = 0x7998bfda <<< 7 = 0xcc5fed3c

A Quarter Round on the ChaCha State

The ChaCha state does not have four integer numbers: it has 16. So the quarter-round operation works on only four of them -- hence the name. Each quarter round operates on four predetermined numbers in the ChaCha state. We will denote by QUARTERROUND(x,y,z,w) a quarter-round operation on the numbers at indices x, y, z, and w of the ChaCha state when viewed as a vector. For example, if we apply QUARTERROUND(1,5,9,13) to a state, this means running the quarter-round operation on the elements marked with an asterisk, while leaving the others alone:

```
 0  *a   2   3
 4  *b   6   7
 8  *c  10  11
12  *d  14  15
```

Test Vector for the Quarter Round on the ChaCha State

For a test vector, we will use a ChaCha state that was generated randomly:

Sample ChaCha State

```
879531e0  c5ecf37d  516461b1  c9a62f8a
44c20ef3  3390af7f  d9fc690b  2a5f714c
53372767  b00a5631  974c541a  359e9963
5c971061  3d631689  2098d9d6  91dbd320
```

We will apply the QUARTERROUND(2,7,8,13) operation to this state. For obvious reasons, this one is part of what is called a "diagonal

round":

After applying QUARTERROUND(2,7,8,13)

879531e0 c5ecf37d *bdb886dc c9a62f8a
44c20ef3 3390af7f d9fc690b *cfacafd2
*e46bea80 b00a5631 974c541a 359e9963
5c971061 *ccc07c79 2098d9d6 91dbd320

Note that only the numbers in positions 2, 7, 8, and 13 changed.

The Chacha20 Block Function

The ChaCha block function transforms a ChaCha state by running multiple quarter rounds.

The inputs to ChaCha20 are:

- A 256-bit key, treated as a concatenation of eight 32-bit little-
- endian integers.
- A 96-bit nonce, treated as a concatenation of three 32-bit little-
- endian integers.
- A 32-bit block count parameter, treated as a 32-bit little-endian
- integer.

The output is 64 random-looking bytes.

The ChaCha algorithm described here uses a 256-bit key. The original algorithm also specified 128-bit keys and 8- and 12-round variants, but these are out of scope for this document. In this section, we describe the ChaCha block function.

Note also that the original ChaCha had a 64-bit nonce and 64-bit block count. We have modified this here to be more consistent with recommendations in Section 3.2 of [RFC5116]. This limits the use of a single (key,nonce) combination to 2^32 blocks, or 256 GB, but that is enough for most uses. In cases where a single key is used by multiple senders, it is important to make sure that they don't use the same nonces. This can be assured by partitioning the nonce space so that the first 32 bits are unique per sender, while the other 64 bits come from a counter.

The ChaCha20 state is initialized as follows:

- The first four words (0-3) are constants: 0x61707865, 0x3320646e, 0x79622d32, 0x6b206574.

- The next eight words (4-11) are taken from the 256-bit key by reading the bytes in little-endian order, in 4-byte chunks.

- Word 12 is a block counter. Since each block is 64-byte, a 32-bit

word is enough for 256 gigabytes of data.

- Words 13-15 are a nonce, which should not be repeated for the same key. The 13th word is the first 32 bits of the input nonce taken as a little-endian integer, while the 15th word is the last 32 bits.

```
cccccccc  cccccccc  cccccccc  cccccccc
kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
bbbbbbbb  nnnnnnnn  nnnnnnnn  nnnnnnnn
```

c=constant k=key b=blockcount n=nonce

ChaCha20 runs 20 rounds, alternating between "column rounds" and "diagonal rounds". Each round consists of four quarter-rounds, and they are run as follows. Quarter rounds 1-4 are part of a "column" round, while 5-8 are part of a "diagonal" round:

1. QUARTERROUND (0, 4, 8,12)
2. QUARTERROUND (1, 5, 9,13)
3. QUARTERROUND (2, 6,10,14)
4. QUARTERROUND (3, 7,11,15)
5. QUARTERROUND (0, 5,10,15)
6. QUARTERROUND (1, 6,11,12)
7. QUARTERROUND (2, 7, 8,13)
8. QUARTERROUND (3, 4, 9,14)

At the end of 20 rounds (or 10 iterations of the above list), we add the original input words to the output words, and serialize the result by sequencing the words one-by-one in little-endian order.

Note: "addition" in the above paragraph is done modulo 2^32. In some machine languages, this is called carryless addition on a 32-bit word.

3.4 The ChaCha20 Encryption Algorithm

ChaCha20 is a stream cipher designed by D. J. Bernstein. It is a refinement of the Salsa20 algorithm, and it uses a 256-bit key. ChaCha20 successively calls the ChaCha20 block function, with the same key and nonce, and with successively increasing block counter parameters. ChaCha20 then serializes the resulting state by writing the numbers in little-endian order, creating a keystream block.

Concatenating the keystream blocks from the successive blocks forms a keystream. The ChaCha20 function then performs an XOR of this keystream with the plaintext. Alternatively, each keystream block can be XORed with a plaintext block before proceeding to create the next block, saving some memory. There is no requirement for the plaintext to be an integral multiple of 512 bits. If there is extra keystream from the last block, it is discarded. Specific protocols may require that the plaintext and ciphertext have certain length. Such protocols need to specify how the plaintext is padded and how much padding it receives.

The inputs to ChaCha20 are:

- A 256-bit key
- A 32-bit initial counter. This can be set to any number, but will usually be zero or one. It makes sense to use one if we use the zero block for something else, such as generating a one-time authenticator key as part of an AEAD algorithm.
- A 96-bit nonce. In some protocols, this is known as the
- Initialization Vector.
- An arbitrary-length plaintext

The output is an encrypted message, or "ciphertext", of the same length. Decryption is done in the same way. The ChaCha20 block function is used to expand the key into a keystream, which is XORed with the ciphertext giving back the plaintext.

Example and Test Vector for the ChaCha20 Cipher

For a test vector, we will use the following inputs to the ChaCha20

block function:

- Key = 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10:11:12:13:14:15:16:17:18:19:1a:1b:1c:1d:1e:1f.
- Nonce = (00:00:00:00:00:00:00:4a:00:00:00:00).
- Initial Counter = 1.

We use the following for the plaintext. It was chosen to be long enough to require more than one block, but not so long that it would make this example cumbersome (so, less than 3 blocks):

```
Plaintext Sunscreen:
000  4c 61 64 69 65 73 20 61 6e 64 20 47 65 6e 74 6c  Ladies and Gentl
016  65 6d 65 6e 20 6f 66 20 74 68 65 20 63 6c 61 73  emen of the clas
032  73 20 6f 66 20 27 39 39 3a 20 49 66 20 49 20 63  s of '99: If I c
048  6f 75 6c 64 20 6f 66 66 65 72 20 79 6f 75 20 6f  ould offer you o
064  6e 6c 79 20 6f 6e 65 20 74 69 70 20 66 6f 72 20  nly one tip for
080  74 68 65 20 66 75 74 75 72 65 2c 20 73 75 6e 73  the future, suns
096  63 72 65 65 6e 20 77 6f 75 6c 64 20 62 65 20 69  creen would be i
112  74 2e                                            t.
```

Figure 3.3a Plaintext Sunscreen

The following figure shows four ChaCha state matrices:

1. First block as it is set up.
2. Second block as it is set up. Note that these blocks are only two bits apart -- only the counter in position 12 is different.

3. Third block is the first block after the ChaCha20 block operation.
4. Final block is the second block after the ChaCha20 block operation was applied.

After that, we show the keystream.

First block setup:

61707865 3320646e 79622d32 6b206574

03020100 07060504 0b0a0908 0f0e0d0c

13121110 17161514 1b1a1918 1f1e1d1c

00000001 00000000 4a000000 00000000

Second block setup:

61707865 3320646e 79622d32 6b206574

03020100 07060504 0b0a0908 0f0e0d0c

13121110 17161514 1b1a1918 1f1e1d1c

00000002 00000000 4a000000 00000000

First block after block operation:

f3514f22 e1d91b40 6f27de2f ed1d63b8

821f138c e2062c3d ecca4f7e 78cff39e

a30a3b8a 920a6072 cd7479b5 34932bed

40ba4c79 cd343ec6 4c2c21ea b7417df0

Second block after block operation:

9f74a669 410f633f 28feca22 7ec44dec

6d34d426 738cb970 3ac5e9f3 45590cc4

da6e8b39 892c831a cdea67c1 2b7e1d90

037463f3 a11a2073 e8bcfb88 edc49139

Keystream:

22:4f:51:f3:40:1b:d9:e1:2f:de:27:6f:b8:63:1d:ed:8c:13:1f:82:3d:2c:06

e2:7e:4f:ca:ec:9e:f3:cf:78:8a:3b:0a:a3:72:60:0a:92:b5:79:74:cd:ed:2b

93:34:79:4c:ba:40:c6:3e:34:cd:ea:21:2c:4c:f0:7d:41:b7:69:a6:74:9f:3f

63:0f:41:22:ca:fe:28:ec:4d:c4:7e:26:d4:34:6d:70:b9:8c:73:f3:e9:c5:3a

c4:0c:59:45:39:8b:6e:da:1a:83:2c:89:c1:67:ea:cd:90:1d:7e:2b:f3:63

Finally, we XOR the keystream with the plaintext, yielding the ciphertext:

```
Ciphertext Sunscreen:
000  6e 2e 35 9a 25 68 f9 80 41 ba 07 28 dd 0d 69 81  n.5.%h..A..(..i.
016  e9 7e 7a ec 1d 43 60 c2 0a 27 af cc fd 9f ae 0b  .~z..C`..'......
032  f9 1b 65 c5 52 47 33 ab 8f 59 3d ab cd 62 b3 57  ..e.RG3..Y=..b.W
048  16 39 d6 24 e6 51 52 ab 8f 53 0c 35 9f 08 61 d8  .9.$.QR..S.5..a.
064  07 ca 0d bf 50 0d 6a 61 56 a3 8e 08 8a 22 b6 5e  ....P.jaV...."."^
080  52 bc 51 4d 16 cc f8 06 81 8c e9 1a b7 79 37 36  R.QM.........y76
096  5a f9 0b bf 74 a3 5b e6 b4 0b 8e ed f2 78 5e 42  Z...t.[......x^B
112  87 4d                                            .M
```

Figure 3.3b Ciphertext Sunscreen

The Poly1305 Algorithm

Poly1305 is a one-time authenticator designed by D. J. Bernstein. Poly1305 takes a 32-byte one-time key and a message and produces a 16-byte tag. This tag is used to authenticate the message.

The original article ([Poly1305]) is titled "The Poly1305-AES message-authentication code", and the MAC function there requires a 128-bit AES key, a 128-bit "additional key", and a 128-bit (non-secret) nonce. AES is used there for encrypting the nonce, so as to get a unique (and secret) 128-bit string, but as the paper states, "There is nothing special about AES here. One can replace AES with an arbitrary keyed function from an arbitrary set of nonces to 16-byte strings."

Regardless of how the key is generated, the key is partitioned into two parts, called "r" and "s". The pair (r,s) should be unique, and must be unpredictable for each invocation (that is why it was originally obtained by encrypting a nonce), while "r" may be constant, but needs to be modified as follows before being used: ("r" is treated as a 16-octet little-endian number):

- r[3], r[7], r[11], and r[15] are required to have their top four bits clear (be smaller than 16)
- r[4], r[8], and r[12] are required to have their bottom two bits clear (be divisible by 4)

The following sample code clamps "r" to be appropriate:

```
/*
Adapted from poly1305aes_test_clamp.c version 20050207
D. J. Bernstein
Public domain.
*/
#include "poly1305aes_test.h"
void poly1305aes_test_clamp(unsigned char r[16])
{
  r[3] &= 15;
```

```
  r[7] &= 15;

  r[11] &= 15;

  r[15] &= 15;

  r[4] &= 252;

  r[8] &= 252;

  r[12] &= 252;

}
```

The "s" should be unpredictable, but it is perfectly acceptable to generate both "r" and "s" uniquely each time. Because each of them is 128 bits, pseudorandomly generating them (see Section 2.6) is also acceptable.

The inputs to Poly1305 are:

- A 256-bit one-time key
- An arbitrary length message
- The output is a 128-bit tag.
- First, the "r" value should be clamped.

Next, set the constant prime "P" be 2^130-5:

3fffffffffffffffffffffffffffffffb. Also set a variable "accumulator"

to zero.

Next, divide the message into 16-byte blocks. The last one might be

shorter:

- Read the block as a little-endian number. Add one bit beyond the number of octets. For a 16-byte block, this is equivalent to adding 2^128 to the number. For the shorter block, it can be 2^120, 2^112, or any power of two that is evenly divisible by 8, all the way down to 2^8.
- If the block is not 17 bytes long (the last block), pad it with zeros. This is meaningless if you are treating the blocks as numbers.
- Add this number to the accumulator.
- Multiply by "r".

- Set the accumulator to the result modulo p. To summarize: Acc = ((Acc+block)*r) % p.

Finally, the value of the secret key "s" is added to the accumulator,

and the 128 least significant bits are serialized in little-endian

order to form the tag.

Poly1305 Example and Test Vector

For our example, we will dispense with generating the one-time key using AES, and assume that we got the following keying material:

- Key Material: 85:d6:be:78:57:55:6d:33:7f:44:52:fe:42:d5:06:a8:01:03:80:8a:fb:0d:b2:fd:4a:bf:f6:af:41:49:f5:1b
- s as an octet string: 01:03:80:8a:fb:0d:b2:fd:4a:bf:f6:af:41:49:f5:1bs as a 128-bit number: 1bf54941aff6bf4afdb20dfb8a800301
- r before clamping: 85:d6:be:78:57:55:6d:33:7f:44:52:fe:42:d5:06:a8
- Clamped r as a number: 806d5400e52447c036d555408bed685

For our message, we'll use a short text:

Message to be Authenticated:

000 43 72 79 70 74 6f 67 72 61 70 68 69 63 20 46 6f Cryptographic Fo

016 72 75 6d 20 52 65 73 65 61 72 63 68 20 47 72 6f rum Research Gro

032 75 70 up

Since Poly1305 works in 16-byte chunks, the 34-byte message divides into three blocks. In the following calculation, "Acc" denotes the accumulator and "Block" the current block:

Block #1

Acc = 00

Block = 6f4620636968706172676f7470797243

Block with 0x01 byte = 016f4620636968706172676f7470797243

Acc + block = 016f4620636968706172676f7470797243

(Acc+Block) * r =

b83fe991ca66800489155dcd69e8426ba2779453994ac90ed284034da565ecf

Acc = ((Acc+Block)*r) % P = 2c88c77849d64ae9147ddeb88e69c83fc

Block #2

Acc = 2c88c77849d64ae9147ddeb88e69c83fc

Block = 6f72472068637261657365522206d7572

Block with 0x01 byte = 016f72472068637261657365522206d7572

Acc + block = 437febea505c820f2ad5150db0709f96e

(Acc+Block) * r =

21dcc992d0c659ba4036f65bb7f88562ae59b32c2b3b8f7efc8b00f78e548a26

Acc = ((Acc+Block)*r) % P = 2d8adaf23b0337fa7cccfb4ea344b30de

Last Block

Acc = 2d8adaf23b0337fa7cccfb4ea344b30de

Block = 7075

Block with 0x01 byte = 017075

Acc + block = 2d8adaf23b0337fa7cccfb4ea344ca153

(Acc + Block) * r =

16d8e08a0f3fe1de4fe4a15486aca7a270a29f1e6c849221e4a6798b8e45321f ((Acc + Block) * r) % P = 28d31b7caff946c77c8844335369d03a7

3.5 Integration of JWT

A JSON web token(JWT) is JSON Object which is used to securely transfer information over the web(between two parties). It can be used for an authentication system and can also be used for information exchange.The token is mainly composed of header, payload, signature. These three parts are separated by dots(.). JWT defines the structure of information we are sending from one party to the another, and it comes in two forms – Serialized, Deserialized. The Serialized approach is mainly used to transfer the data through the network with each request and response. While the deserialized approach is used to read and write data to the web token.

Deserialized

JWT in the deserialized form contains only the header and the payload. Both of them are plain JSON objects.

Nounce

A header in a JWT is mostly used to describe the cryptographic operations applied to the JWT like signing/decryption technique used on it. In our integration, we use Nounce instead of base header. This information is present as a JSON object then this JSON object is encoded to BASE64URL. The cryptographic operations in the header define whether nounce is random value is used only once. A simple header of a JWT looks like the code below:

```
{
   “ nounce”
 }
```

The ‘nounce’ indicating us that the random numbers/values in only used once, if the ‘nounce’ is popup the same numbers/values again, it’ll be invalid. The nounce is 12 byte form in part of ChaCha20-Poly1305

Payload

The payload is the part of the where all the user data is actually added. This information is readable by anyone so it is always not contained any confidential information in here. This part generally contains user information. This information is present as a JSON object then this JSON object is encoded to BASE64URL. The JWT with the payload will look something like this:

```
{
    "userId":"b07f85be-45da",
    "iss": "https://provider.domain.com/",
    "sub": "auth/some-hash-here",
    "exp": 153452683
 }
```

The above JWT contains userId, Iss, sub, and exp. All these play a different role as userId is the ID of the user we are storing, ‘iss’ tells us about the issuer, ‘sub’ stands for subject, and ‘exp’ stands for expiration date.

Serialized

JWT in the serialized form represents a string of the following format:

[nounce].[payload].[signature]

all these three components make up the serialized JWT. We already know what nounce and payload are and what they are used for. Let’s talk about signature.

Signature

This is the third part of JWT and used to verify the authenticity of token. BASE64URL encoded header and payload are joined together with dot(.) and it is then hashed using the poly1305 algorithm with a secret key. This signature is then appended to nounce and payload using dot(.) which forms our actual token nounce.payload.signature

Syntax :

ChaCha20(nounce, base64UrlEncode(payload), secret)

So all these above components together are what makes up a token. Now let’s see how our actual token will look like:

Example :

Nounce: f6jyoo6y4r2r (96 bit nounce)

Payload:

{

"email" : "nick@admin.com",

"password" : "Nick@017"

}

Secret: ChaCha20

JSON Web Token

ZjZqeW9vNnk0cjJy.eyJ1c2VySUQiOiI2MTA2NmJjMmM3N2NjNTY5OThlYjAzND
UiLCJ1c2VybmFtZSI6Ik5pY2sifQ.ld3FbJ57hi6cnLoF6oiJjftBy71-
tRXMW7KAW5FQ0VGt6yFbJzcGulcE8C-_-
zJJrLS7pleu9uKKnZFZ7vqKJG4F3G_4dPcGbAM

Figure 3.3c Custom JWT Token

For a clarity, Figure 3.3c show our custom implementation of token with ChaCha20-Poly1305.

CHAPTER 4

RESULT AND DISCUSSION

RESULT TEST RUN OF JWT: ChaCha20 – Poly1305

To begin with, we have implemented ChaCha20 in our server and hosted it in Heroku. The following context explaining the result.

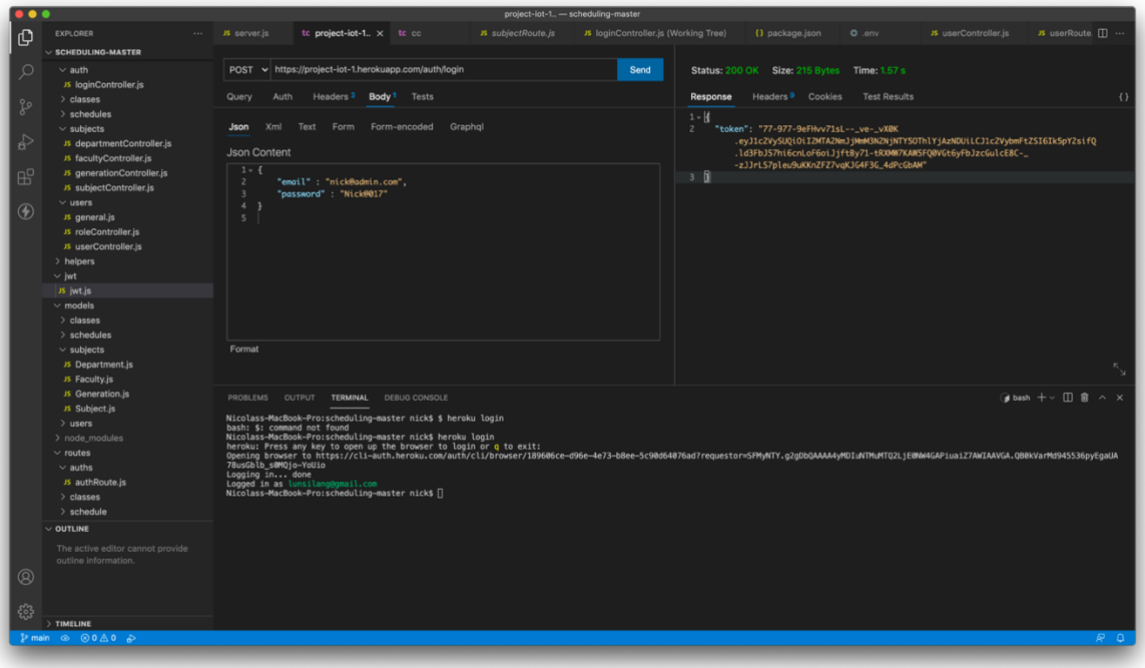


Figure 4a Received Token Result

In Figure 4a, Admin attempted to login into server. If the login credential matched to the database, the server will respond with generated token. We will use this token for authorization.

```
{
  "token": "77-977-9eFHvv71sL--_ve-_vX0K
      .eyJ1c2VySUQiOiI2MTA2NmJjMmM3N2NjNTY5OThlYjAzNDUiLCJ1c2VybmFtZSI6Ik5pY2sifQ
      .ld3FbJ57hi6cnLoF6oiJjftBy71-tRXMW7KAW5FQ0VGt6yFbJzcGulcE8C-_
      -zJJrLS7pleu9uKKnZFZ7vqKJG4F3G_4dPcGbAM"
}
```


Figure 4b Token

This token generated from server in Fig. 4b. The token contained three separated parts identified with “.” The first part before “.” is nounce. The second part between “.” is payload, contained sensitive data. And the last part, after the last “.” is signature which has ChaCha20-Poly1305 encrypted of payload with confidential data.

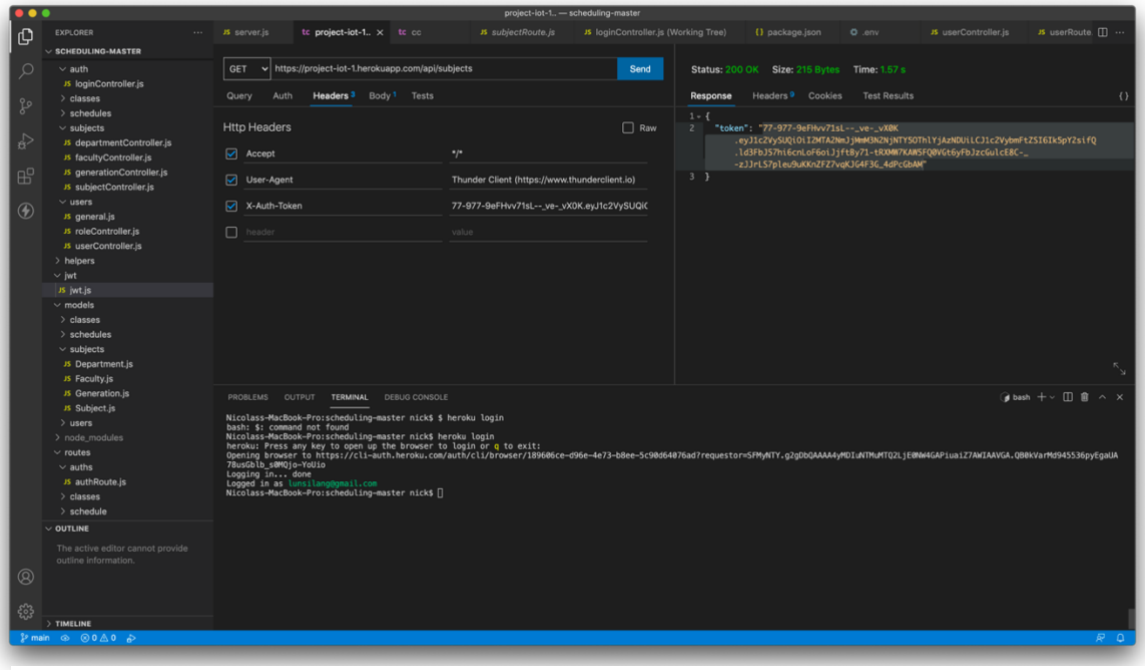


Figure 4c Apply Received Token to X-Auth-Token

In Figure 4c, the token will be used to access the authorized route. There’re some route putting limitation access, only with the valid token can be gained access.

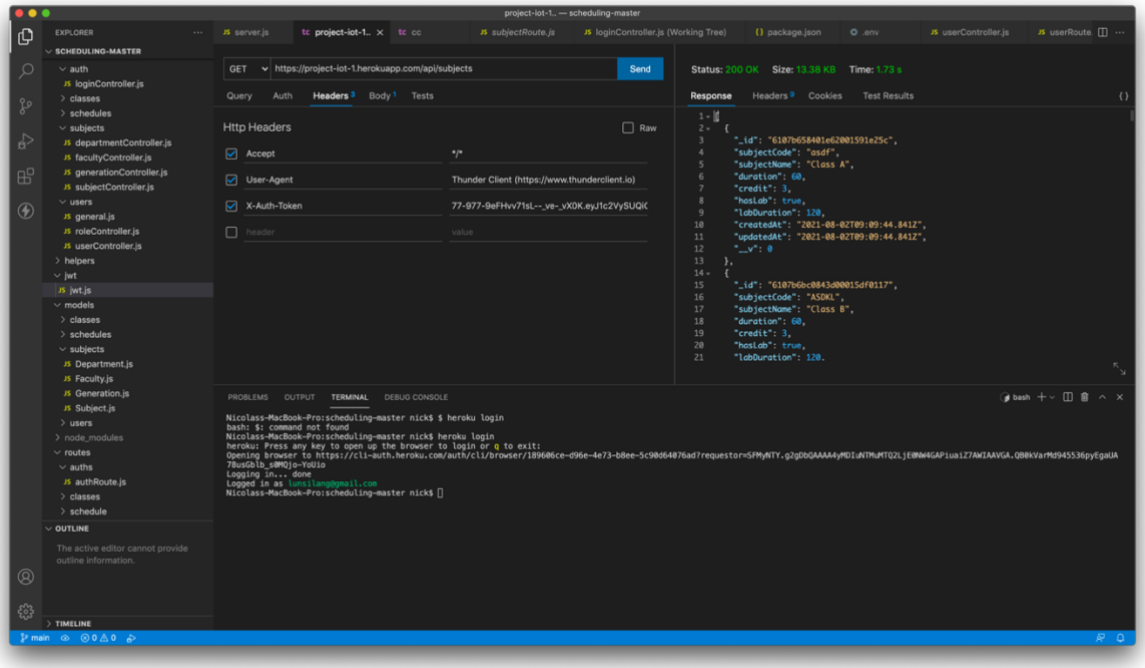


Figure 4d Token Successfully Authorized

In this case Fig 4d, the token is successfully gained access on admin level, able to view plenty of "subjects" that is displayed on the right pane.

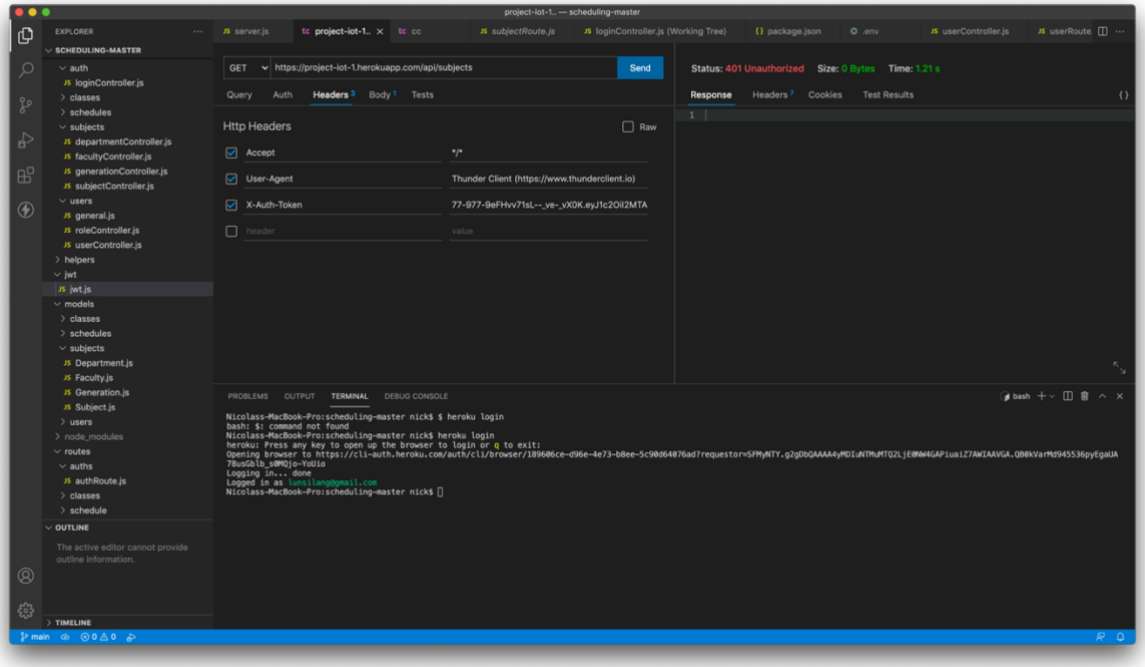


Figure 4e Invalid Token & Unauthorized

However, if the token has been mishandled or leading to lose the integrity, the server will respond with status invalid/unauthorized

CHAPTER 5

CONCLUSION AND FUTUREWORK

In this paper, we study the issue of IoT Devices, Security Measure of IoT Smart Light. Most of the previous research related to nature of ChaCha cipher, such as Salsa cipher as well. With ChaCha20 is found only on security analysis, but there lack of implementation in JWT as well as ChaCha20 in IoT Devices used in smart home itself. We have come up with solution the use custom algorithm of custom ChahCha20-Poly1305 library implemented on server API for authentication with signature. Our future study will take on ChaCha20 Encryption in End to End encryption usage.

REFERENCES

[1] Hafner K, Lyon M (1998) Where wizards stay up late: the origins of the Internet. Simon and Schuster. https://www.amazon.com/Where-Wizards-Stay-Up-Late/dp/0684832674

[2] Sundmaeker H, Guillemin P, Friess P, Woelfflé S (2010) Vision and challenges for realising the Internet of things. Eurpean Research Projects, March. http://www.internet-of-things-research.eu/pdf/IoT_Clusterbook_March_2010.pdf.

[3] Panagiotou P, Sklavos N, Darra E, Zaharakis ID (2020) Cryptographic system for data applications, in the context of internet of things. Microprocess Microsyst 72:102921. https://doi.org/10.1016/j.micpro.2019.102921

[4] Atzori L, Iera A, Morabito G (2010) The internet of things: a survey. Comput Netw 54(15):2787–2805. https://doi.org/10.1016/j.comnet.2010.05.010

[5] Theodorou S, Sklavos N (2019) Blockchain based security and privacy in smart cities. In: DB Rawat, K Ayhan, Z Ghafoor (eds) Chapter in the book: smart cities cybersecurity and privacy. Elsevier Press, 2019 ISBN: 9780128150320.

[6] Subhamoy Maitra, Goutam Paul, and Willi Meier. Salsa20 Cryptanalysis: New Moves and Revisiting Old Styles. https://eprint.iacr.org/2015/217, 2015.

[7] B. Lipp, B. Blanchet and K. Bhargavan, "A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol," 2019 IEEE European Symposium on Security and Privacy (EuroS&P), 2019, pp. 231-246, doi: 10.1109/EuroSP.2019.00026.

[8] D. J. Bernstein, “Chacha, a variant of salsa20,” 2008. [Online]. Available: http://cr.yp.to/chacha/chacha-20080120.pdf

[9] ——, “The salsa20 family of stream ciphers,” in New stream cipher designs. Springer, 2008, pp. 84–97.

[10] S. Maitra, “Chosen iv cryptanalysis on reduced round chacha and salsa,” Discrete Applied Mathematics, vol. 208, pp. 88 – 97, 2016.

[11] A.Silitonga,F.Schade,G.Jiang,andJ.Becker,“Hls-basedperformance and resource optimization of cryptographic modules,” in Proceedings of the 16th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA2018), Melbourne, Australia, 11th- 13th December 2018. IEEE, 2018, pp. 1009–1016.

[12] F. De Santis, A. Schauer, and G. Sigl, “Chacha20-poly1305 authenticated encryption for high-speed embedded IoT applications,” in Proceedings of the Conference on Design, Automation & Test in Europe, ser. DATE ’17. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2017, pp. 692–697.

[13] Xiphera Ltd. Chacha20-poly1305 product brief. [Online]. Available: https://xiphera.com/product brief/ChaCha20 Poly1305 MPSoC.pdf

[14] J.-P. Aumasson, S. Neves, Z. Wilcox-O’Hearn, and C. Winnerlein, “Blake2: Simpler, smaller, fast as md5,” in Applied Cryptography and Network Security, M. Jacobson, M. Locasto, P. Mohassel, and R. Safavi- Naini, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 119–135.

[15] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Proto- cols. RFC 7539 (Informational), may 2015.

[16] Daniel Julius Bernstein. Salsa20 security. https://cr.yp.to/snuf- fle/security.pdf, 2005.

[17] Subhamoy Maitra, Goutam Paul, and Willi Meier. Salsa20 Cryptanalysis: New Moves and Revisiting Old Styles. https://eprint.iacr.org/2015/217, 2015.

[18] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba. In Prof. of 15th Fast Software Encryption Workshop (FSE 2008), Lecture Notes in Computer Science, volume 5086, pages 470–488, 2008. https://eprint.iacr.org/2007/472.

[19] Gordon Procter. A Security Analysis of the Composition of ChaCha20 and Poly1305. https://eprint.iacr.org/2014/613, 2014.