Lightweight Secure CoAP for Internet of Things

Sahand Sabour
1614650

***Abstract*—The Internet of Things refers to a network of interconnected electronic devices or things, capable of sharing and gathering information within this network. This field of study has gained substantial interest in the recent years and considerable research has been conducted regarding its enhancements. Similar to other networks, the IoT requires a set of standard protocols to ensure security for both its devices and users. In addition, as some devices lack resources, the computational cost of proposed methods has to be considered likewise. From the existing IoT protocols, due to its comparatively small overhead and benefits in resource-limited devices, the security of the Constrained Application Protocol (CoAP) is discussed. Accordingly, the existing approaches regarding the security enhancements of this protocol are provided and thoroughly analyzed. Moreover, the problems that the proposed solutions fail to address are highlighted and a novel method to address these problems is proposed. The proposed approach and its relative methodology would be provided. The implementation and experimental results of this method as well as relative comparisons are provided likewise.**

Keywords—Internet of Things, Security, Constrained Application Protocol (COAP), COAPS, Datagram Transport Layer Security (DTLS), Lightweight Cryptography

I. Introduction, Motivation and Background

With the rapid improvements in the modern-day technology the means for connectivity have become highly pivotal. To this end, significant research has been conducted regarding Internet of Things (IoT) and its various protocols. The Internet of Things refers to the field of applications and systems regarding remote control of electrical and electronic devices (i.e. things). Similar to any other type of networks, the aim of the designed protocols for IoT networks is to provide the security model of CIA principles: confidentiality, integrity and availability. However, due to the common computational limitations of numerous devices' resources, it may not possible to simultaneously provide all three principles in many modern approaches.

General implementations of IoT consist of three layers: perception layer, network layer, and application layer. Perception layer represents the sensors and actuators that are used to collect data for the network while the network layer is concerned with routing and transmission of this data. This would include cloud computing, switching and routing devices. In the application layer, the transmitted data would be authenticated, analyzed and converted into meaningful and actionable data. This paper focuses on the security of the application layer.

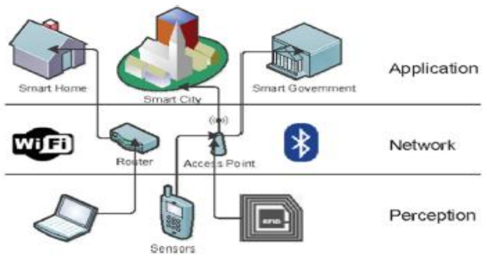


Figure 1: General IoT Architecture [1]

Traditionally, application layers of different network have achieved their globally recognized standard protocols. For instance, HyperText Transfer Protocol (HTTP) is the standard for the World Wide Web (WWW). The main application protocols that are commonly used in modern IoT scenarios are WebSockets, Constrained Application Protocol (CoAp), and Message Queuing Telemetry Transport (MQTT). However, as the current state of IoT networks requires considerable improvements, although significant research has been conducted regarding this topic but a globally standard protocol for this layer has yet to be established.

The research of Mijovic et al. [2] demonstrated a sufficient comparison of the mentioned protocols. The obtained results demonstrated that the CoAP protocol was able to outperform the existing protocols by providing more efficiency for the similar amount of payload. Moreover, the Round Trip Time (RTT) was observed to be comparatively lower than WebSockets and MQTT protocols. Hence, this project focuses on the CoAP and aims to propose a novel method to enhance its security.

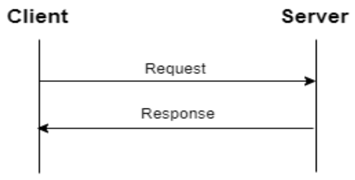


Figure 2: COAP Request/Response

CoAP is an asynchronous application layer protocol, designed by IETF and based on the request and response model, which targets devices with low availability of computational resources [3]. That is, CoAP implements UDP as its transportation channel to remove the extra overhead of TCP handshakes. Due to this implementation, the order of the messages are not guaranteed to be reserved. Furthermore, it features multicasting and provides the usage of URI addresses, which are substantial benefits in comparison with other protocols. However, as mentioned by [4] and [5], the CoAP was mainly designed for Machine-2-Machine (M2M) communications, with no built-in security features included.

This paper aims to critically analyze the existing work regarding the security of the CoAP in the IoT application layer, describe the proposed solutions that address the issues and problems in this protocol, and highlight the shortcomings of the proposed approaches. In addition, a novel method addressing the mentioned shortcomings would be proposed, implemented, and tested.

The rest of this paper is organized as follows: Section II discusses the related work regarding the proposed state-of-the-art security approaches in enhancing the application layer's CoAP. Subsequently, Section III focuses on explaining the shortcomings of the current approaches. In Section IV, a novel method to address these shortcomings and enhance the security of the application layer is proposed. The implementations of the proposed solution as well as its obtained experimental results are provided and thoroughly discussed in Section V. Conclusively, Section VI summarizes the paper and highlights the main conclusions.

II. Literature Survey

Datagram Transport Layer Security (DTLS) is the most notable and widely-use protocol that was proposed for the security of UDP channels in the CoAP. It is inspired by the Transport Layer Security (TLS) protocol, which is widely implemented on secure TCP channels, and can provide data confidentiality and integrity [6]. The DTLS implementation in the CoAP is also referred to as CoAPS (CoAP Secure). However, implementing DTLS increases the number of sent packets in each message which requires additional resources and network traffic, and could decrease the battery of the devices in the network more rapidly. Furthermore, DTLS does not support multicasting, which is a great benefit of implementing CoAP. Hence, the suitability of DTLS for the application layer could be argued.

Zhao et al. [7] proposed a mutual authentication channel between different devices and terminals. In their approach, feature extraction methods and hashing functions are utilized to ensure secure connection between the nodes. However, this approach was believe to be sufficient in theory but not practical in implementation [8]. The method proposed by Wen et al. [9] suggested using one-time pad for encryption and decryption of the messages between different devices. This approach required a pre-shared key matrix between the parties involved in the request. However, secure establishment of these set of keys would not be practical. Raza et al. [10] proposed a compression of the request headers in the DTLS handshakes to reduce the overhead in the network. Considering this approach's efficiency in reducing the overall computational costs, the additional overhead could still be problematic in resource-limited devices. The enhanced DTLS approach proposed by Kumar and Gandhi [11] suffered from the same problem likewise. In addition, their design included a number of smart gateways between the communicating parties, which resulted in significantly increased Round Trip Time (RTT).

III. Problem Identification

The mentioned literature mainly focused on re-using state-of-the-art security protocols in the context of IoT in order to provide more secure establishments. However, many of the implemented protocols were not developed for usage in IoT and were not adapted to the existing constraints. As previously mentioned, many devices in the IoT do not have access to many resources.

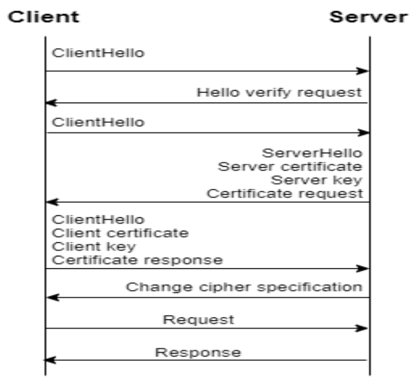


Figure 3: CoAP Request/Response with DTLS

The above figure (Figure 3) demonstrates a simple representation of the message transactions involved in the CoAP with DTLS in a single round trip. As shown in the figure, the DTLS implementation suffers severely from the additional overhead of the extra transactions for the connection establishment. Hence, it would not be practical to implement DTLS in devices with limited resources and this overhead should be considerably reduced.

Moreover, implementing previously shared keys between all the parties for the purpose of end-to-end cryptography could be trivial. Hence, an optimal solution for the security in the application layer of IoT is believed to utilize light-weight cryptography to ensure the integrity of the sent and received messages as well as include implementations that are practical in the real-world IoT scenarios.

IV. Proposed Solution and Novelty

The proposed method confronts the additional overhead problem of DTLS by decreasing the number of messages between the communicating parties and limiting the number of parameters that are shared within each message likewise. Figure 4 illustrates a single round trip between the client and the server based on the proposed method.

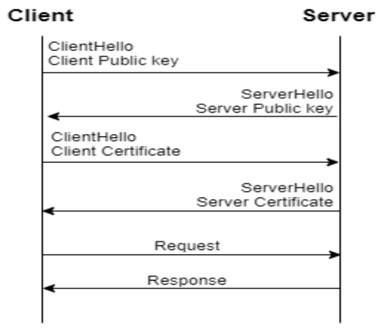


Figure 4: CoAP Request/Response (proposed method)

However, similar to DTLS, the proposed implementation would merely ensure the security of message transportation and not the integrity of the transferred message itself. Hence, due to the importance of utilizing lightweight cryptography in the proposed approach, Elliptic Curve Cryptography (ECC) [12] is implemented to establish a shared key between the two parties at the opposing ends. The complete protocol of the proposed method is as follows:

1) Client A generates public key PK_C and private key SK_C. Then, it sends M1 = (deviceID, PK_C) to server.

2) Server receives and verifies M1 as well as it's the client's device ID. Accordingly, it generate public key PK_S and private key SK_C and sends M2 = ("HelloClient", PK_S) to client.

3) Client receives and verifies M2. Consequently, it generates signature $SIG_C = H(deviceID, PK_C, SK_C)$ and sends M3 = (SIG_C) to server.

4) Server receives M3. The client signiture SIG_C is verified via verify(H(deviceID, PK_C), SIG_C). Then, it calculates $SIG_S = H(deviceID, PK_S, SK_S)$ and sends M4 = (SIG_S) to client. Accordingly, it calculated shared secret key $= PK_C \cdot SK_S$.

5) Client receives M4 and verifies server's signture by verify(H(deviceID, PK_S), SIG_S). Subsequently, it computes shared secret key $ssk= PK_S \cdot SK_C$ and sends request **req** to the server.

6) Server receives request **req**, performs the relative operations and sends corresponding response **res** to client.

7) Client receives and decrypts the response.

V. IMPLEMENTATION AND TESTING

The proposed method was implemented using Python v3.8. The CoAPthon library was used for demonstrating the simple CoAP in this programming language [13]. Accordingly, the tasks of key generation, hashing, signature generation and verification, encryption, and decryption were implemented by utilizing the ECC library. The remainder of the features for this program was written by the author.

In order to demonstrate the proposed method, a CoAP based client (client.py) and a server (server.py) were created. The purpose of these two files was to analyze the resource usage and time consumption of a basic CoAP implementation. The server runs on the CoAP default port 5683 and the code is a basic implementation of the library's documentation. The client receives a method (GET/POST/PUT/DELETE) and an index from the user and sends the corresponding request to the server. The server processes the request based on the designated resource of the index and sends the relative response. This protocol is tested via the following commands:

Python src/server.py

Python src/client.py [request method] [request index]

Where the server should be initiated before the client.

Accordingly, two python files for the proposed method's server and client were created (secureServer.py and secureClient.py). The server runs on the default port for secure CoAP (5584). However, there are two sockets created for the server. The first is considered with the authentication messages in order to establish the shared key and is connected to port 5588. Accordingly, the latter is the default server for collecting information from resources. Similar to the previous implementation, the client receives the request method and index from the client. Subsequently, a series of messages are sent between the two parties to establish an authenticated connection between the two, as discussed in the prior section. Upon successful authentication from both parties, the client sends the request to the server and receives the corresponding response from the relative resource. It should be noted that the shared secret key is established upon the device ID of the client. Hence, in practical applications, the device ID of each device could be provided by a QRcode and accordingly, it could be added to the list of server's accepted IDs by scanning or evaluating this code. The implementation of the proposed protocol can be tested via the following:

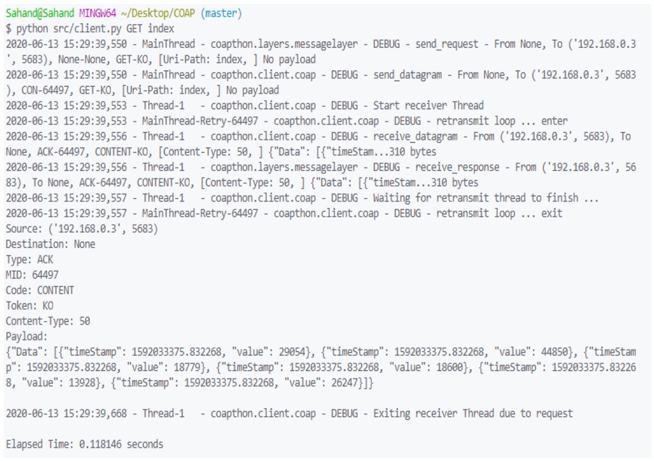
Python src/secureServer.py

Python src/secureClient.py [req method] [req index]

Similar to the previous implementation, the server should be initiated prior to the client's initiation.

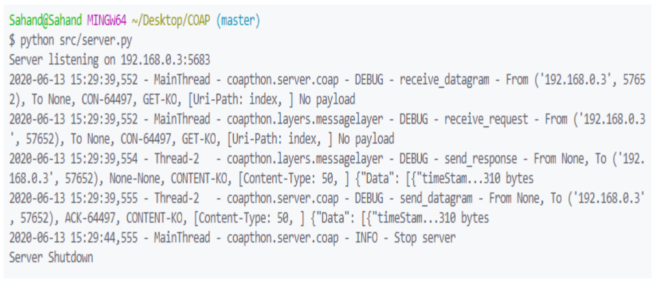
The proposed method was tested on two systems: the platform for the sensor was a desktop computer running 64-bit Windows10, with 32GB of installed RAM and a 2.4GHZ CPU. Visual Studio Code IDE alongside the GitBash terminal were used for developing and testing the code on this platform. Moreover, the client was tested on both a virtual machine with 10GB of RAM and 1.2GHZ CPU with Ubuntu18.04 as its' operating system and the main system itself. The Bash terminal within Visual Studio Code was used to test the code on these platforms likewise. The implemented test consisted of 20 GET /Index from clients of both the local and virtual systems for each implementation. Accordingly, the output results of a single test run are provided respectively below.

Figures 5 and 6 demonstrate a simple request and response transaction between CoAP-based server and client. In this case, both the server and the client reside on the desktop system. That is, the client considers server as localhost.



```
Sahand@Sahand MINGW64 ~/Desktop/COAP (master)
$ python src/client.py GET index
2020-06-13 15:29:39,550 - MainThread - coapthon.layers.messagelayer - DEBUG - send_request - From None, To ('192.168.0.3
', 5683), None-None, GET-KO, [Uri-Path: index, ] No payload
2020-06-13 15:29:39,550 - MainThread - coapthon.client.coap - DEBUG - send_datagram - From None, To ('192.168.0.3', 5683
), CON-64497, GET-KO, [Uri-Path: index, ] No payload
2020-06-13 15:29:39,553 - Thread-1   - coapthon.client.coap - DEBUG - Start receiver Thread
2020-06-13 15:29:39,553 - MainThread-Retry-64497 - coapthon.client.coap - DEBUG - retransmit loop ... enter
2020-06-13 15:29:39,556 - Thread-1   - coapthon.client.coap - DEBUG - receive_datagram - From ('192.168.0.3', 5683), To
None, ACK-64497, CONTENT-KO, [Content-Type: 50, ] {"Data": [{"timeStam...310 bytes
2020-06-13 15:29:39,556 - Thread-1   - coapthon.layers.messagelayer - DEBUG - receive_response - From ('192.168.0.3', 56
83), To None, ACK-64497, CONTENT-KO, [Content-Type: 50, ] {"Data": [{"timeStam...310 bytes
2020-06-13 15:29:39,557 - Thread-1   - coapthon.client.coap - DEBUG - Waiting for retransmit thread to finish ...
2020-06-13 15:29:39,557 - MainThread-Retry-64497 - coapthon.client.coap - DEBUG - retransmit loop ... exit
Source: ('192.168.0.3', 5683)
Destination: None
Type: ACK
MID: 64497
Code: CONTENT
Token: KO
Content-Type: 50
Payload:
{"Data": [{"timeStamp": 1592033375.832268, "value": 29054}, {"timeStamp": 1592033375.832268, "value": 44850}, {"timeStam
p": 1592033375.832268, "value": 18779}, {"timeStamp": 1592033375.832268, "value": 18600}, {"timeStamp": 1592033375.83226
8, "value": 13928}, {"timeStamp": 1592033375.832268, "value": 26247}]}

2020-06-13 15:29:39,668 - Thread-1   - coapthon.client.coap - DEBUG - Exiting receiver Thread due to request

Elapsed Time: 0.118146 seconds
```

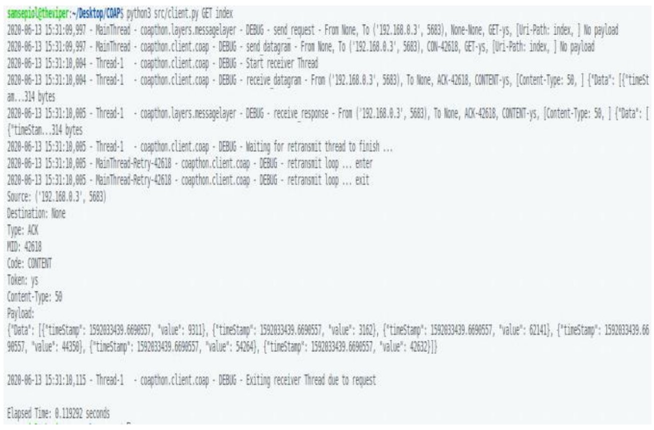
Figure 5: CoAP client request (Local system)



```
Sahand@Sahand MINGW64 ~/Desktop/COAP (master)
$ python src/server.py
Server listening on 192.168.0.3:5683
2020-06-13 15:29:39,552 - MainThread - coapthon.server.coap - DEBUG - receive_datagram - From ('192.168.0.3', 5765
2), To None, CON-64497, GET-KO, [Uri-Path: index, ] No payload
2020-06-13 15:29:39,552 - MainThread - coapthon.layers.messagelayer - DEBUG - receive_request - From ('192.168.0.3
', 57652), To None, CON-64497, GET-KO, [Uri-Path: index, ] No payload
2020-06-13 15:29:39,554 - Thread-2   - coapthon.layers.messagelayer - DEBUG - send_response - From None, To ('192.
168.0.3', 57652), None-None, CONTENT-KO, [Content-Type: 50, ] {"Data": [{"timeStam...310 bytes
2020-06-13 15:29:39,555 - Thread-2   - coapthon.server.coap - DEBUG - send_datagram - From None, To ('192.168.0.3'
, 57652), ACK-64497, CONTENT-KO, [Content-Type: 50, ] {"Data": [{"timeStam...310 bytes
2020-06-13 15:29:44,555 - MainThread - coapthon.server.coap - INFO - Stop server
Server Shutdown
```

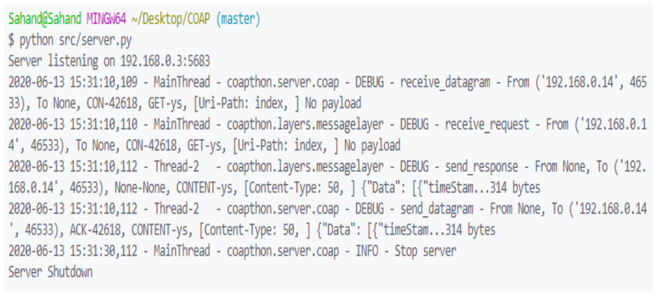
Figure 6: CoAP server response to local client

Subsequently, Figures 7 and 8 demonstrate the connection and message transportation between a CoAP residing on the virtual machine as a remote client and a local server.



```
samsepiol@theviper:~/Desktop/COAP$ python3 src/client.py GET index
2020-06-13 15:31:09,997 - MainThread - coapthon.layers.messagelayer - DEBUG - send_request - From None, To ('192.168.0.3', 5683), None-None, GET-ys, [Uri-Path: index, ] No payload
2020-06-13 15:31:09,997 - MainThread - coapthon.client.coap - DEBUG - send_datagram - From None, To ('192.168.0.3', 5683), CON-42618, GET-ys, [Uri-Path: index, ] No payload
2020-06-13 15:31:10,004 - Thread-1   - coapthon.client.coap - DEBUG - Start receiver Thread
2020-06-13 15:31:10,004 - Thread-1   - coapthon.client.coap - DEBUG - receive_datagram - From ('192.168.0.3', 5683), To None, ACK-42618, CONTENT-ys, [Content-Type: 50, ] {"Data": [{"timeSt
am...314 bytes
2020-06-13 15:31:10,005 - Thread-1   - coapthon.layers.messagelayer - DEBUG - receive_response - From ('192.168.0.3', 5683), To None, ACK-42618, CONTENT-ys, [Content-Type: 50, ] {"Data": [
{"timeStam...314 bytes
2020-06-13 15:31:10,005 - Thread-1   - coapthon.client.coap - DEBUG - Waiting for retransmit thread to finish ...
2020-06-13 15:31:10,005 - MainThread-Retry-42618 - coapthon.client.coap - DEBUG - retransmit loop ... enter
2020-06-13 15:31:10,005 - MainThread-Retry-42618 - coapthon.client.coap - DEBUG - retransmit loop ... exit
Source: ('192.168.0.3', 5683)
Destination: None
Type: ACK
MID: 42618
Code: CONTENT
Token: ys
Content-Type: 50
Payload:
{"Data": [{"timeStamp": 1592033439.6690557, "value": 9311}, {"timeStamp": 1592033439.6690557, "value": 3162}, {"timeStamp": 1592033439.6690557, "value": 62141}, {"timeStamp": 1592033439.66
90557, "value": 44350}, {"timeStamp": 1592033439.6690557, "value": 54264}, {"timeStamp": 1592033439.6690557, "value": 42632}]}

2020-06-13 15:31:10,115 - Thread-1   - coapthon.client.coap - DEBUG - Exiting receiver Thread due to request

Elapsed Time: 0.119292 seconds
```

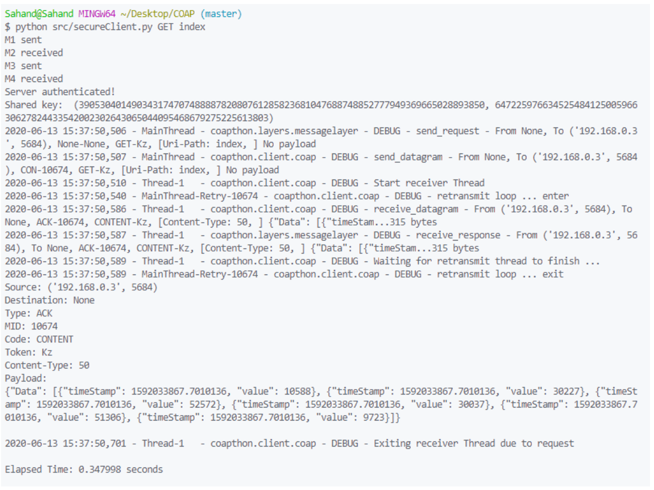
Figure 7: CoAP client request (Virtual system)



```
Sahand@Sahand MINGW64 ~/Desktop/COAP (master)
$ python src/server.py
Server listening on 192.168.0.3:5683
2020-06-13 15:31:10,109 - MainThread - coapthon.server.coap - DEBUG - receive_datagram - From ('192.168.0.14', 465
33), To None, CON-42618, GET-ys, [Uri-Path: index, ] No payload
2020-06-13 15:31:10,110 - MainThread - coapthon.layers.messagelayer - DEBUG - receive_request - From ('192.168.0.1
4', 46533), To None, CON-42618, GET-ys, [Uri-Path: index, ] No payload
2020-06-13 15:31:10,112 - Thread-2   - coapthon.layers.messagelayer - DEBUG - send_response - From None, To ('192.
168.0.14', 46533), None-None, CONTENT-ys, [Content-Type: 50, ] {"Data": [{"timeStam...314 bytes
2020-06-13 15:31:10,112 - Thread-2   - coapthon.server.coap - DEBUG - send_datagram - From None, To ('192.168.0.14
', 46533), ACK-42618, CONTENT-ys, [Content-Type: 50, ] {"Data": [{"timeStam...314 bytes
2020-06-13 15:31:30,112 - MainThread - coapthon.server.coap - INFO - Stop server
Server Shutdown
```

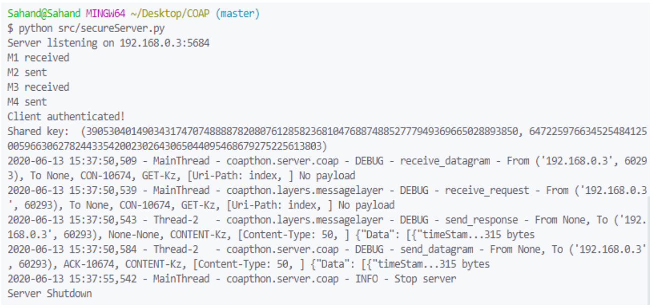
Figure 8: CoAP server response to remote client

Based on the above figures, it can be observed that the basic implementation of the CoAP is accurate and efficient. Hence, this would be a sufficient foundation for testing and run-time comparison to the proposed method.



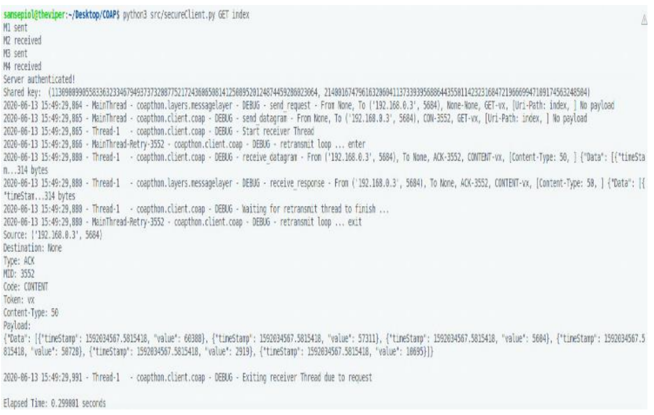
```
Sahand@Sahand MINGW64 ~/Desktop/COAP (master)
$ python src/secureClient.py GET index
M1 sent
M2 received
M3 sent
M4 received
Server authenticated!
Shared key:  (39053040149034317470748888782080761285823681047688748852777949369665028893850, 647225976634525484125005966
30627824433542002302643065044095468679275225613803)
2020-06-13 15:37:50,506 - MainThread - coapthon.layers.messagelayer - DEBUG - send_request - From None, To ('192.168.0.3
', 5684), None-None, GET-Kz, [Uri-Path: index, ] No payload
2020-06-13 15:37:50,507 - MainThread - coapthon.client.coap - DEBUG - send_datagram - From None, To ('192.168.0.3', 5684
), CON-10674, GET-Kz, [Uri-Path: index, ] No payload
2020-06-13 15:37:50,510 - Thread-1   - coapthon.client.coap - DEBUG - Start receiver Thread
2020-06-13 15:37:50,540 - MainThread-Retry-10674 - coapthon.client.coap - DEBUG - retransmit loop ... enter
2020-06-13 15:37:50,586 - Thread-1   - coapthon.client.coap - DEBUG - receive_datagram - From ('192.168.0.3', 5684), To
None, ACK-10674, CONTENT-Kz, [Content-Type: 50, ] {"Data": [{"timeStam...315 bytes
2020-06-13 15:37:50,587 - Thread-1   - coapthon.layers.messagelayer - DEBUG - receive_response - From ('192.168.0.3', 56
84), To None, ACK-10674, CONTENT-Kz, [Content-Type: 50, ] {"Data": [{"timeStam...315 bytes
2020-06-13 15:37:50,589 - Thread-1   - coapthon.client.coap - DEBUG - Waiting for retransmit thread to finish ...
2020-06-13 15:37:50,589 - MainThread-Retry-10674 - coapthon.client.coap - DEBUG - retransmit loop ... exit
Source: ('192.168.0.3', 5684)
Destination: None
Type: ACK
MID: 10674
Code: CONTENT
Token: Kz
Content-Type: 50
Payload:
{"Data": [{"timeStamp": 1592033867.7010136, "value": 10588}, {"timeStamp": 1592033867.7010136, "value": 30227}, {"timeSt
amp": 1592033867.7010136, "value": 52572}, {"timeStamp": 1592033867.7010136, "value": 30037}, {"timeStamp": 1592033867.7
010136, "value": 51306}, {"timeStamp": 1592033867.7010136, "value": 9723}]}

2020-06-13 15:37:50,701 - Thread-1   - coapthon.client.coap - DEBUG - Exiting receiver Thread due to request

Elapsed Time: 0.347998 seconds
```

Figure 9: Proposed method's client request (Local System)



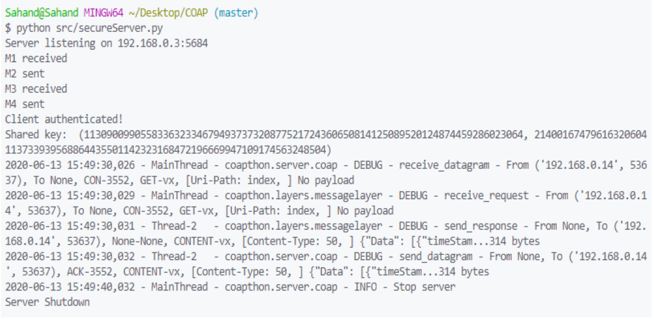
```
Sahand@Sahand MINGW64 ~/Desktop/COAP (master)
$ python src/secureServer.py
Server listening on 192.168.0.3:5684
M1 received
M2 sent
M3 received
M4 sent
Client authenticated!
Shared key:  (39053040149034317470748888782080761285823681047688748852777949369665028893850, 647225976634525484125
00596630627824433542002302643065044095468679275225613803)
2020-06-13 15:37:50,509 - MainThread - coapthon.server.coap - DEBUG - receive_datagram - From ('192.168.0.3', 6029
3), To None, CON-10674, GET-Kz, [Uri-Path: index, ] No payload
2020-06-13 15:37:50,539 - MainThread - coapthon.layers.messagelayer - DEBUG - receive_request - From ('192.168.0.3
', 60293), To None, CON-10674, GET-Kz, [Uri-Path: index, ] No payload
2020-06-13 15:37:50,543 - Thread-2   - coapthon.layers.messagelayer - DEBUG - send_response - From None, To ('192.
168.0.3', 60293), None-None, CONTENT-Kz, [Content-Type: 50, ] {"Data": [{"timeStam...315 bytes
2020-06-13 15:37:50,584 - Thread-2   - coapthon.server.coap - DEBUG - send_datagram - From None, To ('192.168.0.3'
, 60293), ACK-10674, CONTENT-Kz, [Content-Type: 50, ] {"Data": [{"timeStam...315 bytes
2020-06-13 15:37:55,542 - MainThread - coapthon.server.coap - INFO - Stop server
Server Shutdown
```

Figure 10: Proposed method's response to local client



```
sansepiol@theviper:~/Desktop/COAP$ python3 src/secureClient.py GET index
M1 sent
M2 received
M3 sent
M4 received
Server authenticated!
Shared key:  (113090099055833632334679493737320877521724360650814125089520124874459286023064, 21400167479616320604113733939568864435501142323168472196669947109174563248504)
2020-06-13 15:49:29,864 - MainThread - coapthon.layers.messagelayer - DEBUG - send_request - From None, To ('192.168.0.3', 5684), None-None, GET-vx, [Uri-Path: index, ] No payload
2020-06-13 15:49:29,865 - MainThread - coapthon.client.coap - DEBUG - send_datagram - From None, To ('192.168.0.3', 5684), CON-3552, GET-vx, [Uri-Path: index, ] No payload
2020-06-13 15:49:29,865 - Thread-1   - coapthon.client.coap - DEBUG - Start receiver Thread
2020-06-13 15:49:29,866 - MainThread-Retry-3552 - coapthon.client.coap - DEBUG - retransmit loop ... enter
2020-06-13 15:49:29,888 - Thread-1   - coapthon.client.coap - DEBUG - receive_datagram - From ('192.168.0.3', 5684), To None, ACK-3552, CONTENT-vx, [Content-Type: 50, ] {"Data": [{"timeSta
m...314 bytes
2020-06-13 15:49:29,888 - Thread-1   - coapthon.layers.messagelayer - DEBUG - receive_response - From ('192.168.0.3', 5684), To None, ACK-3552, CONTENT-vx, [Content-Type: 50, ] {"Data": [{
"timeStam...314 bytes
2020-06-13 15:49:29,888 - Thread-1   - coapthon.client.coap - DEBUG - Waiting for retransmit thread to finish ...
2020-06-13 15:49:29,888 - MainThread-Retry-3552 - coapthon.client.coap - DEBUG - retransmit loop ... exit
Source: ('192.168.0.3', 5684)
Destination: None
Type: ACK
MID: 3552
Code: CONTENT
Token: vx
Content-Type: 50
Payload:
{"Data": [{"timeStamp": 1592034567.5815418, "value": 60388}, {"timeStamp": 1592034567.5815418, "value": 57311}, {"timeStamp": 1592034567.5815418, "value": 5604}, {"timeStamp": 1592034567.5
815418, "value": 50728}, {"timeStamp": 1592034567.5815418, "value": 2919}, {"timeStamp": 1592034567.5815418, "value": 10695}]}

2020-06-13 15:49:29,991 - Thread-1   - coapthon.client.coap - DEBUG - Exiting receiver Thread due to request

Elapsed Time: 0.299001 seconds
```

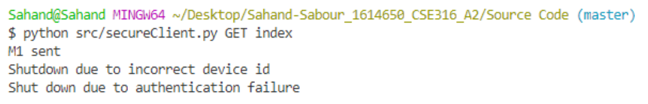
Figure 11: Proposed method's remote client request



```
Sahand@Sahand MINGW64 ~/Desktop/COAP (master)
$ python src/secureServer.py
Server listening on 192.168.0.3:5684
M1 received
M2 sent
M3 received
M4 sent
Client authenticated!
Shared key:  (113090099055833632334679493737320877521724360650814125089520124874459286023064, 21400167479616320604
113733939568864435501142323168472196669947109174563248504)
2020-06-13 15:49:30,026 - MainThread - coapthon.server.coap - DEBUG - receive_datagram - From ('192.168.0.14', 536
37), To None, CON-3552, GET-vx, [Uri-Path: index, ] No payload
2020-06-13 15:49:30,029 - MainThread - coapthon.layers.messagelayer - DEBUG - receive_request - From ('192.168.0.1
4', 53637), To None, CON-3552, GET-vx, [Uri-Path: index, ] No payload
2020-06-13 15:49:30,031 - Thread-2   - coapthon.layers.messagelayer - DEBUG - send_response - From None, To ('192.
168.0.14', 53637), None-None, CONTENT-vx, [Content-Type: 50, ] {"Data": [{"timeStam...314 bytes
2020-06-13 15:49:30,032 - Thread-2   - coapthon.server.coap - DEBUG - send_datagram - From None, To ('192.168.0.14
', 53637), ACK-3552, CONTENT-vx, [Content-Type: 50, ] {"Data": [{"timeStam...314 bytes
2020-06-13 15:49:40,032 - MainThread - coapthon.server.coap - INFO - Stop server
Server Shutdown
```

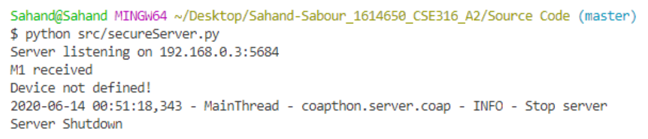
Figure 12: Proposed method's response to remote client

Similarly, Figures 9-12 demonstrate the implementation testing of the proposed method via both local and remote clients. As shown in the figures, the communicating parties are able to accurately exchange the required messages, securely authenticate the credibility of the opposing party, and successfully establish a shared secret key. In addition, upon obtaining a shared key, the clients were enabled to submit a given to the server and accordingly, receive the corresponding response.



```
Sahand@Sahand MINGW64 ~/Desktop/Sahand-Sabour_1614650_CSE316_A2/Source Code (master)
$ python src/secureClient.py GET index
M1 sent
Shutdown due to incorrect device id
Shut down due to authentication failure
```

Figure 13: Invalid Device ID (Client side)



```
Sahand@Sahand MINGW64 ~/Desktop/Sahand-Sabour_1614650_CSE316_A2/Source Code (master)
$ python src/secureServer.py
Server listening on 192.168.0.3:5684
M1 received
Device not defined!
2020-06-14 00:51:18,343 - MainThread - coapthon.server.coap - INFO - Stop server
Server Shutdown
```

Figure 14: Invalid Device ID (Server side)

Furthermore, in the case that the client provides an invalid device ID the communication is terminated. That is, if the device ID supplied by the client is not recognized by the server, the protocol would not continue as the client is not authenticated. The obtained results for the complete testing phase is summarized and provided in the Table 1.

Method	Min	Average	Max
CoAP	0.117325 s	0.119034 s	0.1226845 s
CoAPS [14]	0.368749 s	0.381574 s	0.4264857 s
Proposed	0.298961 s	0.336894 s	0.3489935 s

Table 1: Runtime comparison chart for GET /index

Based on obtained results, it can be identified that the proposed method has an increased run-time and overhead compared to the basic implementation of the CoAP. However, additional overhead is expected when security measures are implemented. Comparison between the proposed method's and CoAPS' runtime analysis clearly indicates enhancements in the overall overhead and transportation time. Furthermore, as the proposed approach contains lightweight cryptography that is achievable by common IoT constraints, its implementation is believed to be practical in real-world scenarios.

VI. Conclusions

In this paper, the Internet of Things (IoT) and its security protocols were briefly introduced. Moreover, Constrained Application Protocol (CoAP), which is a lightweight protocol for the application layer of IoT, was described and its security issues were analyzed. Furthermore, the state-of-the-art approaches and proposed methods were thoroughly investigated and their shortcomings were highlighted likewise. Subsequently, a novel approach for confronting the mentioned issues was proposed and demonstrated in detail. In addition, relative implementation and experimental results of the proposed method was provided. Based on the obtained results, it could be concluded that the proposed method would a practical approach due to its lightweight cryptography and overall overhead comparative enhancement.

References

[1] S. Panchiwala and M. Shah, “A Comprehensive Study on Critical Security Issues and Challenges of the IoT World”, Journal of Data, Information and Management, April 2020.

[2] S. Mijovic, E. Shehu, and C.Burrati, “Comparing Application Layer Protocols for the Internet of Things via Experimentation”, 2016 IEEE 2nd International Forum on Research and Technologies, 2016.

[3] C. Bormann , A .Castellani, and Z. Shelby,“CoAP: An Application Protocol for Billions of Tiny Internet Nodes”, IEEE Internet Computing, vol.16, no.2, pp. 62-67, April 2012.

[4] V. Karagiannis, P. Chatzimisios, and F. Vazquez-Gallego, “A survey on application layer protocols for the Internet of Things”, Transaction on Iot and Cloud Computing, January 2015.

[5] N. Naik, “Choice of Effective Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP and HTTP”, 2017 IEEE International Systems Engineering Symposium (ISSE), Vienna, 2018, pp. 1-7.

[6] T.A. Alghamdi, A. Lasebae, and M. Aiash, “Security Analysis of the Constrained Application Protocol in the Internet of Things”, Second International Conference on Future Generation Communication Technology (FGCT), 2013, pp. 163-168.

[7] G. Zhao, X. Si, J. Wang, and T. Hu, “A novel mutual authentication scheme for Internet of Things”, International Conference on Modelling, Identification and Control (ICMIC), pp. 563-566, 2011.

[8] R. Mahmoud, T. Yousuf, F. Aloul, and I. ZualKernan, “Internet of Things (IoT) Secuirty: Current Status, Challenges and Prospective Measuers”, 10th International Conference for Internet Technology and Secured Transactions (ICITST), London, 2015, pp. 336-341

[9] Q. Wen, X. Dong, and R. Zhang, “Application of dynamic variable cipher security certificate in internet of things”, International Conference on Cloud Computing and Intelligent Systems (CCIS), pp. 1062-1066, 2012.

[10] S. Raza, H. Shafagh, K. Hewage, R. Hummen, and T. Voigt, “Lithe: Lightweight secure CoAP for the Internet of things,” IEEE Sensors J., vol. 13, no. 10, pp. 3711–3720, 2013.

[11] P.M. Kumar and U.D. Gandhi, “Enhanced DTLS with CoAP-based authentication scheme for the internet of things in healthcare application”, J SuperComput 76, pp. 3963-3983, 2020.

[12] G. Seroussi, “Elliptic Curve Cryptography”, 1999 Information Theory and Networking Workshop, Metsovo, Greecem 1999, pp. 41-46.

[13] G.Tanganelli, C. Vallati, and E.Mingozzi, "CoAPthon: Easy Development of CoAP-based IoT Applications with Python", IEEE World Forum on Internet of Things, 2015.

[14] JayNoblez (2020) DTLSCoAp (Version 0.4) [Source Code]. https://github.com/JayNoblez/DTLSCoAP