Application Layer Security for IoT: The Case Study of a Smart Home

Rasha Ghadeer[1], Ahmad Mahmoud Ahmad [2], Radwan Dandah [3]

[1]PhD Student, Dept. of System and Computer Networks Engineering, Tishreen University, Latakia, Syria.
[2]Assistant Professor, Dept. of System and Computer Networks Engineering, Tishreen University, Latakia, Syria.
[3]Professor, Dept. of System and Computer Networks Engineering, Tishreen University, Latakia, Syria.

Abstract - *The Internet of Things is composed of objects with distinct identities that communicate with one another via the internet. This paved the door for a variety of applications, including home automation, which improves human comfort and security. Security is considered as one of the most important topics that must be taken into consideration. In this paper an application layer security is provided to a smart home system, and that is a set of sensors communicating with the cloud using a constrained application protocol (CoAP). These sensors protected by using two protocols, the first one is Object Security for Constrained RESTful Environments (OSCORE) protocol which provides authenticated encryption for the payload data, while the second one is Ephemeral Diffie-Hellman Over COSE (EDHOC) protocol which provides the symmetric session keys required for OSCORE. We simulate the CoAP protocol without security using the Contiki-NG operating system, then we simulate the EDHOC with OSCORE protocols over CoAP. A comparison between the two experiments was made in terms of memory footprint and energy consumption since these two factors are the most concerning factors in constrained devices in any IoT environment. The results showed that the current implementation gives small overhead compared to other security solutions.*

Key Words: **Smart Home, Security, CoAP, OSCORE, Contiki-NG, Cooja, EDHOC.**

1.INTRODUCTION

In recent years, Internet has grown rapidly and changed human's life by providing better connectivity and communication. Internet technology can be extended to connect objects that are used in day-to-day life. This expansion of internet services is called Internet of Things (IOT)[1]. The term "smart home" refers to a subset of the Internet of Things (IoT) paradigm that combines home automation and security. Homeowners can remotely monitor and control devices in a normal household by connecting them to the Internet. From timer-controlled lamps that can be turned off from anywhere to smart thermostats that can control the temperature in a home while also producing detailed energy usage information. The widespread availability of low-cost smartphones, microcontrollers, and other open-source hardware, as well as the growing use of cloud services, has enabled the development of low-cost smart home security systems. [2].

However, there exist a number of issues with IoT devices, two of them are security and restrictions on resource consumption. Due to the low-resources nature of such devices, complex security protocols are often not viable, and that results in a vast amount of easily hacked targets. Also using old, unencrypted protocols for communication can be a vector of attack as well, leaving the traffic open to examination and manipulation. This is the area we focused on, by implementing a new standard designed to protect a lightweight application protocol commonly used for constrained devices in IoT networks. One way to approach this problem is to use protocols specifically designed for the use case, with energy efficiency in mind while still guaranteeing security [3].

CoAP [4] is a customized web transfer protocol designed for usage with confined nodes and constrained networks (e.g., low-power, lossy). Datagram Transport Layer Security (DTLS) [5] is the only way to enable secure communication for CoAP, according to the CoAP definition. DTLS, in particular, creates a secure channel at the transport layer across unreliable datagram protocols like UDP, and provides hop-by-hop security by encrypting all CoAP communications. Most researches about DTLS protocol show that using DTLS as a security protocol increase the burden on the objects. Thus, it is important to optimize the current implementation of this protocol to have an efficient and reliable IoT devices, or we can search for alternative protocols to achieve security [6].

Building on CoAP protocol, OSCORE protocol offers an end-to-end encrypted security layer while staying compatible with nodes, such as proxies [7]. However, OSCORE has only been standardized very recently (July 2019) and thus, unsurprisingly, there didn't exist implementations on all the operating systems that it uses in IoT objects. Closely related to OSCORE, there exists another new protocol called EDHOC [8]. It is intended to complement OSCORE and extend it with the functionality to allow secure key exchanges.

2.THE IMPORTANCE OF THE RESEARCH AND ITS OBJECTIVES

The importance of the research comes through implementing a new standard designed to protect a lightweight application protocol (CoAP) commonly used for constrained devices in IoT Operating system (Contiki-NG). This protocol is globally used in a lot of IoT applications. In addition, this is the first implementation of the EDHOC protocol with the OSCORE protocol built in Contiki-NG OS.

This research aims to provide application layer security to the smart home system designed in cooja simulator in Contiki-NG OS and study the impact of using this type of security on the constrained objects (light bulbs, thermometer, air conditioner ...) in terms of memory footprint and power consumption.

3. RELATED WORK

Different techniques were done in research to achieve security on IOT based Smart Homes. In [9] a brief overview of using voice assistants like Amazon Alexa, Google Home, Apple Siri, or Microsoft Cortana to detect voice commands from a person with speaking disabilities in a much more natural way to control ordinary electrical appliances, and analysing the method of security. While the authors of [10] try to identify the best hash function that can be introduced to the CoAP protocol to improve security without compromising efficiency. The CoAP protocol takes into account three different hash functions: SHA-1, SHA224, and SHA256. The upgraded protocol was tested on a smart home application using the Contiki OS simulation tool, and the findings reveal that SHA 224 is the optimum hash algorithm in terms of performance. In [11], a blockchain-based method to data privacy and security in a smart home is offered. To provide a safe foundation for IoT devices in smart home systems, they propose an authentication strategy that integrates attribute-based access control with smart contracts and edge computing.

On the other hand, the OSCORE protocol was achieved with the EDHOC protocol in different programming languages and systems. In paper [12] the researchers presented the first implementation of the EDHOC and OSCORE protocols using the Rust programming language, as they showed applicability in a real client and resource server scenario on restricted STM32 devices. While in [13], they implement the two protocols designed for embedded devices on GitLab. The new libraries, which implement the mentioned protocols, were written and tested with the assistance of the Continuous Integration pipeline. In [14] describes the design of the uOSCORE and uEDHOC libraries for ordinary microcontrollers, as well as the uOSCORE-TEE and uEDHOC-TEE libraries for microcontrollers that have a Trusted Execution Environment (TEE), such as ARM TrustZone-M microcontrollers.

To the best of our knowledge, no prior studies have thoroughly secured smart home systems using application layer OSCORE protocol, or evaluated OSCORE with EDHOC together in Contiki-NG OS in terms of device memory footprint and energy consumption.

4. Technical background

This part introduces the background principles that will be discussed throughout the article.

4.1. The Constrained Application Protocol:

CoAP is a RESTful application layer protocol especially designed for the IoT domain. It takes into account two sorts of devices: clients and servers, both of which communicate via requests and responses. Sensors and actuators, for example, are stored on the servers. Clients can use the PUT, GET, POST, and DELETE methods to access those resources. A Uniform Resource Identifier (URI) identifies each resource. Each CoAP packet begins with a fixed 4-byte header carrying the method type (PUT, GET, POST, DELETE) or a response code, among other information. The header is followed by an optional token used to correlate requests and responses. The token is followed by optional options that contain additional parameters for the requests/responses. These are followed by an optional payload, prefixed with the payload [4].

4.2. Object Security for Constrained RESTful Environments:

OSCORE [7] uses CBOR Object Signing and Encryption (COSE) [15] to provide application-layer protection for the CoAP protocol. CoAP endpoints can use OSCORE to create an end-to-end security system. As a result, pre-shared keys or keys created using a key exchange protocol such as EDHOC may be used. Converting CoAP or HTTP messages to OSCORE messages then allows for secure communication. HTTP messages are translated to CoAP messages initially in this procedure. The CoAP message is protected using COSE encryption in order to construct an OSCORE message. The OSCORE parameters as well as the message fields of the COSE-Encrypt object are then included in the encrypted message's header fields. OSCORE is used to establish a security context in the implementation employed in this research.

Security Context: A security context must be established before both the sender and the receiver can encrypt and decrypt communications exchanged. This security context corresponds to a set of parameters that are required for cryptographic operations in OSCORE. The security context

consists of three sub contexts: the "common context", the "sender context" and the "recipient context" as shown in figure 1.

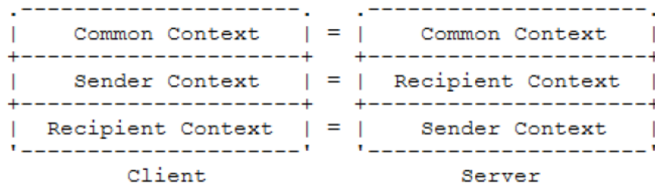

Fig -1: Matching of Contexts, from [7]

The common context is derived first, then is used together with additional data in order to derive the sender and recipient context. Two endpoints that want to communicate over an OSCORE secured channel derive each a sender and a receiver context. In order to receive asymmetric keys, the sender and receiver IDs are used in reverse so that the sender context of the first endpoint is matching the receiver context of the second endpoint and vice versa [7].

4.3. Ephemeral Diffie-Hellman Over COSE:

EDHOC [8] is a Diffie-Hellman key exchange protocol for constructing a shared secret based on an ephemeral key, which provides perfect forward secrecy, identity protection, and mutual authentication. The protocol, which implements the Elliptic Curve Diffie–Hellman (ECDH) algorithm, is detailed in [8] as a work-in-progress IETF draft recently endorsed by the IETF LAKE Working Group. In this protocol, COSE [15], CBOR [16], and CoAP [4] are also used for cryptography, encoding, and transport.

For key generation and security parameter negotiation, EDHOC employs a three-message protocol. The parties exchanging messages are the initiator (I) and responder (R). For each session, they create a new ephemeral ECDH key pair, exchange the public part of their ECDH keys, calculate the shared secret, and generate symmetric application keys. To authenticate communications, raw public keys (RPK) with signature keys or static ECDH keys, as well as public key certificates, can be utilized.

5. EXPERIMNET AND RESULTS

We simulate our experiment on Contiki-NG OS [17]. which is an open-source operating system for IoT, it connects tiny low-power, low-cost microcontrollers to the Internet. Contiki-NG OS provides low-consumption Internet communication and supports many low-power wireless standards. We used the Cooja simulator provided by Contiki. It's a network simulator that allows developers to test their apps on completely simulated devices before deploying them to real hardware.

Our work in this paper is divided into three parts. First, we design a smart home system on Contiki-NG OS. Second, we add application layer security to the proposed system using OSCORE and EDHOC protocols. Third, we evaluate the impact of applying security to the nodes in terms of energy consumption and memory footprint in both plain scenarios without security on one hand, and with applying security on the other hand.

5.1. Designing the smart home system:

For the real-time simulation, the sensor nodes (motes) are implemented in Contiki-NG OS and run in the Cooja simulator as shown in Figure 2, which provides the set of sensors, and a border router (BR).

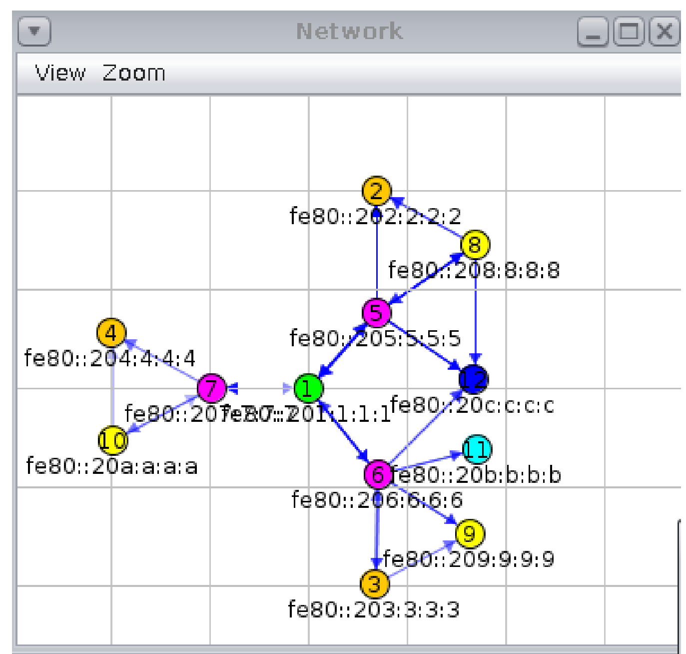

Fig -2: the network simulation of the smart home

On the simulator, each node has TX range of 50m and INT range of 100m where TX range means that the transmitted packet can be received easily by any node which is inside of the range while INT range is the range where the transmission of packets can be heard but cannot receive the transmitted packet. The sensor nodes are actually the end devices which only send data to its neighbor sensors and BR. As shown in Table 1 there are 6 types of smart devices emulated in the smart home as 12 cooja motes distributed in the network.

Table -1: Cooja motes descriptions

Mote number	Mote role
1	Border Router
2,3,4	Lightbulb

International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 09 Issue: 03 | Mar 2022 www.irjet.net p-ISSN: 2395-0072

5,6,7	Motion sensor
8,9,10	Thermometer
11	Heater
12	Air Conditionar

A border router is used to connect a regular IP with the RPL 6LoWPAN network. In the Contiki–Cooja simulation environment, a network's border router is located at the network's edge. The BR is also working as a gateway to connect two different networks. In our simulation we should write a command line, for starting connection between BR and the cloud as shown in Figure 3, this command line is:

make TARGET=cooja connect-router-cooja

Fig -3: Command line for starting connection between BR and cloud

After starting the simulation each node follows some steps to get connected with each other. (From Figure 4) at the first step, a set of custom lightweight networking protocol rime stack starts. When one node comes to the stack it automatically gets an abstract address. On the second step, the node gets a mac address and is assigned with a node id. After that, the radio channel and channel band were set. The values are the same for all nodes of the network. Next, the node is assigned with a tentative IPv6 address and at last the node starts transmitting or receiving data.

Fig -4: Mote output

These sensor nodes are able to interface with a cloud application, which has A CoAP server that registers the smart nodes and starts monitoring them through the CoAP observing. Also, there is a command line interface (CLI) that the user can exploit to send request to each node of the network. As shown in figure 5, the cloud shows the 11-motes available in a smart home.

Fig -5: Command line for the HomeIoT-Cloud

5.2. Adding application layer security to the proposed smart home system:

We use the OSCORE version presented in the research study [18], which included an open-source implementation of the OSCORE protocol found on github [19]. In this implementation, there is no algorithm or protocol used to exchange keys between the client and the server. The Master Secret and Master Salt are manually placed in the code, in addition to the Sender ID and Receiver ID from the client side whose values are set opposite to the Sender ID and the Receiver ID of the server side.

On the other hand, we implement EDHOC protocol to generate keys and exchange them between the client and the server to start creating the OSCORE security context. EDHOC protocol generates a Master Secret, Master Salt, Sender ID, and Recipient ID for both the client and server. The Master Secret and Master Salt are shared by both parties, while the Sender ID and Recipient ID are placed inversely between the client and server. This setting is used in OSCORE to derive content encryption keys and decryption keys that are then used to encrypt and decrypt message traffic between the client and the server. In this implementation, CoAP is used for transport [4], COSE for encryption [15], and CBOR for encoding [16].

5.3. Result Analysis:

Our simulation's main purpose is to simulate the CoAP protocol without security and then simulating the EDHOC with OSCORE protocol over CoAP. Comparing the two experiments in terms of memory footprint and energy consumption, because these two characteristics are the most concerning in restricted devices.

5.3.1 Energy consumption:

We use the Energest library [20] to measure energy consumption. Contiki-NG includes the Energest module which can be used to implement lightweight, software-based energy estimation approaches for resource-constrained IoT devices. The Energest module keeps track of how much time a system has spent in different stages. The researcher can estimate the system's energy usage by combining this information with the hardware power consumption model. We categorize energy consumption into four categories:

- tx - the number of ticks the radio was in transmit mode (ENERGEST_TYPE_TRANSMIT)
- rx - the number of ticks the radio was in receive mode (ENERGEST_TYPE_LISTEN)
- cpu - the number of ticks in active mode for the CPU (ENERGEST TYPE CPU)
- cpu idle - the amount of ticks in which the CPU has been idle (ENERGEST TYPE LPM)

Tx and rx, as well as cpu and idle, are mutually exclusive; the system can never be in both modes at the same time. Other combinations are possible, for example, it might be in cpu and tx at the same time. The overall uptime of the system is the sum of cpu and idle. The RTIMER ARCH SECOND constant defines the duration of a timer tick, which is platform-dependent.

Simple-energest is a system service included with Contiki-NG. We add this line to every sensor's makefile to activate the simple-energest service:

MODULES += os/services/simple-energest

Once the Simple Energest service is enabled, it will print a summary message once per minute. An example message taken from mote output on our emulated BR node shown in Figure 6.

```
30:01.292  ID:1  [INFO: Energest  ] Total time  :   1966081
30:01.299  ID:1  [INFO: Energest  ] CPU         :    131364/   1966081 (66 permil)
30:01.306  ID:1  [INFO: Energest  ] LPM         :   1834717/   1966081 (933 permil)
30:01.312  ID:1  [INFO: Energest  ] Deep LPM    :         0/   1966081 (0 permil)
30:01.319  ID:1  [INFO: Energest  ] Radio Tx    :     37844/   1966081 (19 permil)
30:01.327  ID:1  [INFO: Energest  ] Radio Rx    :   1927740/   1966081 (980 permil)
30:01.334  ID:1  [INFO: Energest  ] Radio total :   1965584/   1966081 (999 permil)
```

Fig -6: From Mote output, BR Energest output in one minute

To compute the average current consumption (in milliamperes, mA) we can use this equation

state_avg_current_mA = (ticks * current_mA) / (RTIMER_ARCH_SECOND * period_sec) = (ticks * current_mA) / period_ticks

Then we can compute charge consumption (in millicoulumbs, mC) :

state_charge_mC = (ticks * current_mA) / RTIMER_ARCH_SECOND

where:

- ticks - the number of ticks spent in a state
- RTIMER_ARCH_SECOND - the number of ticks per second
- period_sec - the duration of the accounting period in seconds
- period_ticks - the duration of the accounting period in period_ticks
- current_mA - the current consumption in that state in mA
- voltage - the voltage provided by the system to the component (radio or CPU)

Finally, to compute the energy consumption (in millijoules, mJ), multiply the power with the duration in seconds or multiply the charge with the voltage of the system:

state_energy_mJ = state_charge_mC * voltage

After running the simulation for the smart home system described in section 5.1 for 1 hour, we save the mote

output in a file. This file contains the output of the simple Energest every minute for all the 12 nodes. Then depending on the script in [21], which uses the upper equations we compute the energy consumption in both scenarios, one without security and the second one when applying security. The results shown in Chart 1.

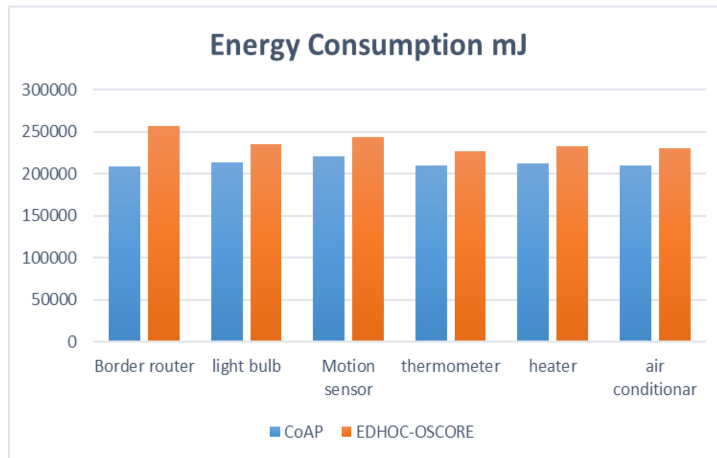

Chart -1: Energy consumption for nodes in 3600 second

We can see from the Chart 1 that using EDHOC and OSCORE protocols results in energy consumption about 10% higher than in COAP. If we compare these results with reference studies [18] and [22], which study the impact of using DTLS as security protocol on the energy usage of the node. It results that using DTLS protocol consumes energy about 50% higher than in COAP. Our experimental results show that OSCORE displays moderately better performance than DTLS in important metric like energy usage.

5.3.2 Memory footprint:

We use the following command to measure the memory [23]:

size <binary file name>. elf

We measure the RAM using the bss and data.

RAM = data + bss

We measure the ROM using the text and data.

ROM = data + text

The bss represents the uninitialized variables storage, data represents the initialized variable area and the text represents the code and constants.

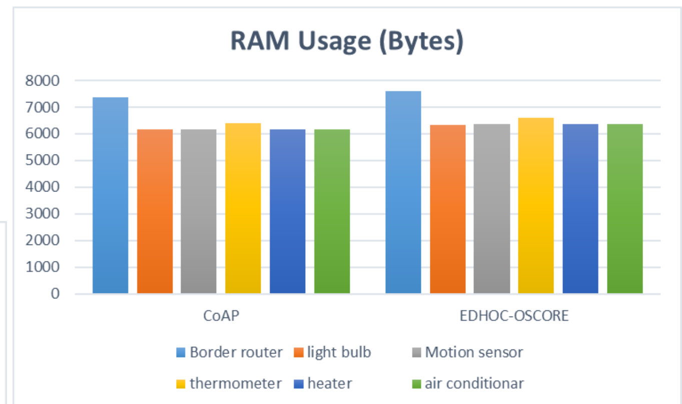

Chart -2: Memory Footprint (RAM Usage)

We can see in Chart 2 and Chart 3 that OSCORE only uses 3% more RAM and 14% more ROM than CoAP. Comparing with [18] , DTLS protocol uses 17% more RAM and 27% more ROM than CoAP. So, while OSCORE uses more memory compared to CoAP, but it outperforms DTLS protocol in conserving nodes resources.

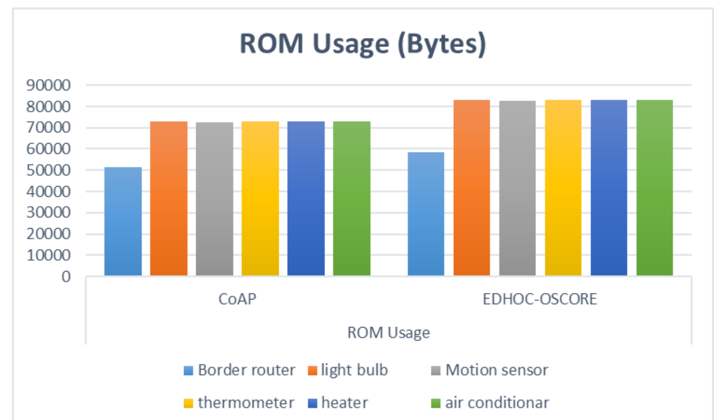

Chart -3: Memory Footprint (ROM Usage)

6. CONCLUSIONS

IoT security is gaining a lot of attention these days from both academia and industry. Due to significant energy consumption and computational overhead, existing security solutions are not always ideal for IoT. We previously developed an application layer security solution that secures the CoAP protocol, which overcomes these issues. A smart home was used as an example case study to discuss the concept. The results showed that applying security creates a small overhead, the energy usage overhead is about 10%, the RAM overhead is 3 % and the ROM overhead is about 14% higher than in COAP, which is still acceptable for the majority of IoT applications, and it is less than when we use DTLS protocol to secure CoAP protocol.

Our implementation is able to run on limited RAM sizes, such as the one in the IoT devices with limited resources. Our evaluation demonstrates the applicability of the EDHOC implementation as an effective way for establishing an end-to-end application layer security context including resource constrained embedded devices.

ACKNOWLEDGEMENT

The authors wish to acknowledge the Faculty of Information Engineering at Tishreen University for their support of this research.

REFERENCES

[1] S. Li, L. Da Xu, and S. Zhao, “The internet of things: a survey,” Inf. Syst. Front., 2015, doi:10.1007/s10796-014-9492-7.

[2] Hoque, Mohammad Asadul, and Chad Davidson. "Design and Implementation of an IoT-Based Smart Home Security System." Int. J. Networked Distributed Comput. 7.2 (2019): 85-92.

[3] Von Raumer, Marco. "Continuous integration of embedded security software." (2020).

[4] Shelby, Z., Hartke, K., & Bormann, C. (2014). The constrained application protocol (coap)(rfc 7252). Jun-2014 Available online. http://www. rfc-editor. org/info/rfc7252.

[5] E. Rescorla, N. Modadugu, Datagram Transport Layer Security Version 1.2, RFC, RFC Editor, Fremont, CA, USA, 2012, doi: 10.17487/RFC6347 .

[6] Alamri, Mohammed Hassan. "Securing the Constrained Application Protocol (CoAP) for the Internet of Things (IoT)." (2017).

[7] F. Palombini G. Selander, J. Mattsson and L. Seitz. Object Security for Constrained RESTful Environments (OSCORE). RFC 8613, July 2019. 20, 21

[8] G. Selander, J. Mattsson, and F. Palombini, “Ephemeral diffie-hellman over cose (edhoc),” Internet Requests for Comments, RFC Editor, RFC draft-ietf-lake-edhoc-05 (work in progress), February 2021. [Online].Available: https://datatracker.ietf.org/doc/draft-ietf-lake-edhoc

[9] Ranjan, Mr Vivek, and Mr Harsh Vardhan Mishra. "HOME AUTOMATION AND SECURITY USING INTERNET OF THINGS."

[10] Halabi, Dana, Salam Hamdan, and Sufyan Almajali. "Enhance the security in smart home applications based on IOT-CoAP protocol." 2018 Sixth International Conference on Digital Information, Networking, and Wireless Communications (DINWC). IEEE, 2018.

[11] Qashlan, A., Nanda, P., He, X., & Mohanty, M. (2021). Privacy-preserving mechanism in smart home using blockchain. IEEE Access, 9, 103651-103669.

[12] Disch, M. Lightweight Application Layer Protection for Embedded Devices with a Safe Programming Language. Diss. Ph. D. dissertation, Software Engineering Group Department of Informatics, Switzerland, 2020.

[13] von Raumer, M. Continuous integration of embedded security software. (2020). Master thesis, University of Fribourg (Switzerland).

[14] Hristozov, S.; Huber, M.; Xu, L.; Fietz, J.; Liess, M.; & Sigl, G. The Cost of OSCORE and EDHOC for Constrained Devices. In Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy, (2021, April) , (pp. 245-250).

[15] J. Schaad, “Cbor object signing and encryption (cose),” Internet Requests for Comments, RFC Editor, RFC 8152, July 2017. [Online]. Available: https://tools.ietf.org/html/rfc7252

[16] C. Bormann and P. E. Hoffman, “Concise binary object representation (cbor),” Internet Requests for Comments, RFC Editor, RFC 7049, October 2013. [Online]. Available: https://tools.ietf.org/html/rfc7049

[17] CONTIKI-NG, 10Jun. 2022. <https://www.contiki-ng.org/>.

[18] Gunnarsson, M.; Brorsson, J.; Palombini, F.; Seitz, L.; & Tiloca, M. Evaluating the performance of the OSCORE security protocol in constrained IoT environments. 2021 Internet of Things, volume 13, 100333. ISSN 2542-6605.

[19] OSCORE implementation in Contiki-NG , 10Jun. 2022.<https://github.com/Gunzter/contiki-ng/tree/master> .

[20] Tutorial: Energy monitoring - The Energest module https://github.com/contiki-ng/contiki-ng/wiki/Tutorial:-Energy-monitoring

[21] Instrumenting Contiki NG applications with energy usage estimation < https://github.com/contiki-ng/contiki-ng/wiki/Instrumenting-Contiki-NG-applications-with-energy-usage-estimation>

[22] Kantharajan, Karnarjun, and Sahar Shirafkan. "Efficient Security Protocol for RESTful IoT devices." (2020).

[23] Tutorial: RAM and ROM usage, from Contiki-NG wiki, https://github.com/contiki-ng/contiki-ng/wiki/Tutorial:-RAM-and-ROM-usage