International Journal on Advanced Science Engineering Information Technology

Vol.12 (2022) No. 2
ISSN: 2088-5334

Secure Communication Protocol for Arduino-based IoT Using Lightweight Cryptography

Rizki Agus Zandra Kurniawan [a], Sri Wahjuni [a,*], Shelvie Nidya Neyman [a]

[a] *Department of Computer Science, Faculty of Mathematics and Natural Sciences, IPB University, Bogor 16680, Indonesia*
*Corresponding author: *my_juni04@apps.ipb.ac.id*

***Abstract*—We witness massive implementations of the Internet of Things (IoT) in smart homes, smart buildings, smart vehicles, smart wearables as well as Industry 4.0 initiatives. Along with the massive adoption, IoT security has become more important and crucial in this case. Arduino, as IoT hardware platform, also requires enhancements on its security to ensure that data it transmits and receives is secured and has not been tampered in any way. Transmitting of IoT data and telecommand in plaintext is not secure. Securing transmission using traditional block cipher is computationally intensive for embedded-systems with low memory and computing power like Arduino. This research proposes a novel lightweight security communication protocol that is lightweight enough to run on the Arduino platform. The proposed protocol shall be utilizing a lightweight key agreement scheme, the SPECK lightweight block cipher, and BLAKE2s hash function. This protocol is designed to support telemetry and telecommand by using publisher-subscriber, which also is aimed to be extensible but straightforward for future enhancements. This research shows that a secure IoT communication protocol can be designed and implemented on Arduino devices and another IoT platform running Arduino core such as the ESP32. The performance evaluation of this protocol in Arduino Mega shows that the INIT phase's average execution time is 26.83 milliseconds. The key agreement is 13.50 milliseconds, and the encryption-decryption of telemetry and telecommand messages requires 25 milliseconds execution time. The protocol performance evaluation in ESP32 has an average execution time for INIT phase 44.63 milliseconds. The key agreement phase, 13.90 milliseconds, and the encryption and decryption of telemetry and telecommand messages requires an execution time of 17.10 milliseconds.**

***Keywords*— Arduino; blake2s; IoT security; lightweight cryptography; pervasive computing; security protocol for IoT; speck cipher.**

Manuscript received 18 Apr. 2019; revised 13 Jun. 2020; accepted 8 Oct. 2020. Date of publication 30 Apr. 2022.

I. INTRODUCTION

The Internet of Things (IoT) or the Internet of Objects is a network that connects objects in everyday life. The IoT extends the Internet as we know today to a more extensive network with the integration of objects in everyday life through embedded systems. A network of intelligent objects will communicate with each other. At present, IoT applications have reached many aspects of daily life that include the health sector, transportation, utilities, smart homes, and smart appliance. Research by Ahmad and Zafar [1] shows that IoT technology has been applied to healthcare, mostly remote medical monitoring. In this case, smart sensors monitor patients' biomechanical and physiological data such as blood pressure, data heart, body temperature, and blood sugar. The data would then be transmitted via a wireless *body area network* of each patient via a PDA or smartphone, relaying the data to the central server for analytics.

Arduino is very popular for the development and IoT applications. Research in the IoT field that uses Arduino is very ubiquitous, such as [2], which focuses on developing an Arduino based environmental security system. Wahjuni et al. [3] developed a fuzzy inference system to monitor the eel breeding environment and estimate young eels' survival rate. The system is developed using Arduino Uno, and Intel Galileo will then send telemetry data to the server. Wahjuni and Waladi [4] also developed an Arduino-based automatic irrigation system and REST protocol with several input parameters such as soil moisture and pH. The Arduino platform has very limited system resources such as memory or processing power. Therefore, any resources consumption shall be carefully managed.

According to ESET [5], many IoT devices send information in *plaintext*, use inadequate encryption technology, or not use any authentication method. Loi et al. research [6] found that many IoT devices in the consumer market communicate in *plaintext*. In contrast, only one device uses a *secure socket layer protocol,* albeit with low entropy parameters. Besides, five of twenty IoT devices involved in the research were susceptible to replay attacks, while only two were known to resist these attacks.

One of the most important data communication security services is confidentiality. This is achieved by encrypting the data using a cryptographic encryption algorithm with an *encryption key* to prevent the data from being read by any unauthorized parties. Encryption algorithm such as a *block cipher* requires both authorized parties -the sender and the receiver of data- to agree in using the same encryption key in order to exchange encrypted message successfully. Another important security feature is data integrity, which is important to ensure that the transmitted data is not tampered with during transmission.

Lightweight cryptography algorithms are a new class of cryptography algorithms specifically designed to provide an adequate security level and require minimal memory and computing resources. These algorithms are considered more suitable for IoT environments that have very limited computing resources.

Wu and Zhang state [7] that lightweight cryptographic algorithms, when compared to traditional cryptographic algorithms, have three distinctive properties: First, the application of IoT devices generally do not need to encrypt large amounts of data. Second, the attackers cannot record enough encrypted IoT data for cryptanalysis. Therefore, lightweight cryptographic algorithms only need a moderate level of security. Third, light cryptographic algorithms are generally applied to hardware such as 8-bit microcontrollers, so efficiency is critical. Dinu et al. [8] mentioned that important criteria in the use of lightweight cryptography algorithms include code size, RAM required, and computational time needed to encrypt and decrypt data. Research by Loi et al. [6] also mentions that many IoT solutions do not use any form of secure communication protocol in transmitting data from IoT devices to servers or from the server to user applications.

This research aims to design a communication protocol that can primarily provide data confidentiality, data integrity, and authentication services. This protocol should be able to run on IoT devices that have limited computing resources such as the Arduino platform.

II. Material And Method

A. Development Environment

The programming language used for this research is a C++ programming language with the Arduino IDE version 1.8.5 for development on the Arduino IoT device. This research also utilizes open source libraries such as the ArduinoJSON, *ArduinoCryptLibs*, and communication libraries for SIM808 GPRS modules and Wifi modules for ESP32. The protocols server-side is developed using the Python programming language version 3.7.1 with Microsoft Visual Studio Code as IDE. Protocol server-side development also utilizes open source software such as Redis, which serves the *publisher-subscriber* module to support this protocol's functionalities. Redis is used for tracking the protocol communication status within the *protocol state table* and IoT device configuration databases. Open-source libraries being used are Redisworks as the Redis client in Python, Gunicorn as the WSGI HTTP server and Falcon framework, and the implementation of the Speck algorithm in the Python language. We also use Wireshark to analyze raw protocol communication.

B. Test Environment

The Arduino Mega 2560 microcontroller platform used the test equipment, which has an 8-bit processor with a 16MHz clock speed, 8 KB RAM memory, and 256KB flash storage a 4KB EEPROM. The Arduino Mega is connected to the SIM808 GSM / GPRS module. We also use the ESP32 microcontroller with Arduino core, where this platform has 160MHz clock speed, 520KB RAM memory, 2MB embedded flash storage. We also would like to evaluate if the protocol can work in various IoT microcontroller platforms. Therefore, in addition to Arduino Mega, ESP32 is also being used as the test environment.

The research data was taken from the development and implementation of protocols on the Arduino Mega test device with GPRS module and Arduino ESP32 with Wifi. We measure the processing time required at each phase of the protocol. This data collection was carried out on both test devices with a total sample of 30 samples.

C. Assumptions

The assumptions used include: 1) The time of the protocol phase process does not include the time required by network services such as GPRS / Wifi / Internet given the unpredictability of reliability and latency. 2) Public Key Infrastructure and digital certificates are outside the scope of this protocol. 3) Availability of network services where this protocol can be run, such as GSM / GPRS, LoRa, Wireless LAN, Ethernet. is outside this protocol's scope.

D. Research Methodology

This research consists of 5 (five) main parts, namely protocol design, server protocol development, state table protocol development, and publisher-subscriber on the server-side, development of the protocol on the Arduino side as a client. The overall research methodology is shown in Fig 1.

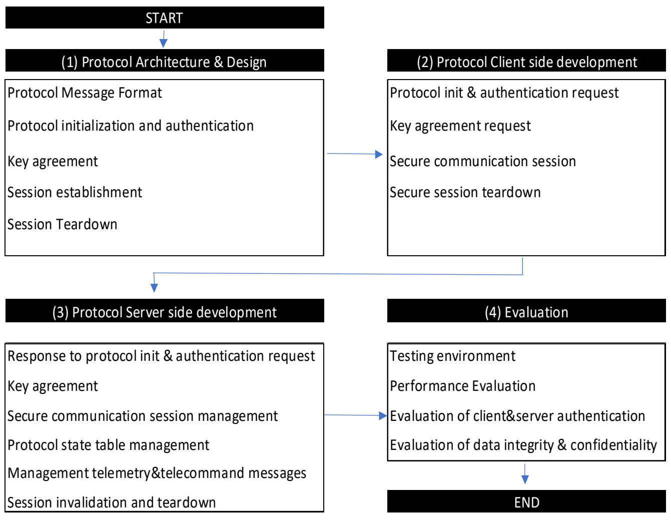

START
(1) Protocol Architecture & Design
Protocol Message Format
Protocol initialization and authentication
Key agreement
Session establishment
Session Teardown
(2) Protocol Client side development
Protocol init & authentication request
Key agreement request
Secure communication session
Secure session teardown
(3) Protocol Server side development
Response to protocol init & authentication request
Key agreement
Secure communication session management
Protocol state table management
Management telemetry&telecommand messages
Session invalidation and teardown
(4) Evaluation
Testing environment
Performance Evaluation
Evaluation of client&server authentication
Evaluation of data integrity & confidentiality
END

Fig. 1 Research Methodology

E. *High-Level Topology*

The protocol is designed to run on top of the HTTP protocol, utilizing any TCP/IP connection, such as using GPRS or Wifi connection. The high-level topology of this protocol is shown in Fig.2.

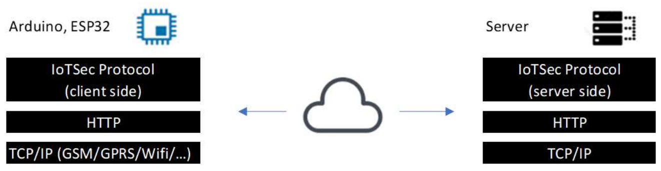

Arduino, ESP32
IoTSec Protocol (client side)
HTTP
TCP/IP (GSM/GPRS/Wifi/...)
Server
IoTSec Protocol (server side)
HTTP
TCP/IP

Fig. 2 Protocol High-Level Topology

F. *Lightweight Cryptography Algorithms in use*

Lightweight cryptography algorithms are designed to run in devices with very limited computing and memory resources. This is ideal for Arduino as IoT hardware, which has such characteristics. There is increasingly more research, such as assessing several lightweight block ciphers' performances for IoT [8] and evaluating lightweight cipher in 8-bit microcontroller [9].

Beaulieu et al designed the SPECK cipher algorithm. at the National Security Agency [10] as a general-purpose lightweight block cipher that offers good performance while aiming as the smallest implementation to suit IoT applications. The SPECK cipher has ten variations: the SPECK-32/64 (32-bit block size, 64-bit key, 22 encryption rounds) up to the SPECK-128/256 (128-bit block size, 256-bit key, 34 encryption rounds). The SPECK cipher has since become the subject of research assessing its security and performance. Biryukov et al. [11] concluded that among several algorithms tested. The Speck cipher is one among light cryptographic algorithms that have resistance to *correlation power analysis* attack, in which the attacker exploits the correlation of power leakage with a specific cryptographic function to try to extract the key. Dinur [12] uses *improved differential cryptanalysis* to attack a scaled-down (*reduced round*) SPECK implementation. However, the research concludes that the enhanced attacks do not threaten any member of SPECK's security. Dwivedi et al. [13] published an attack using *differential cryptanalysis* on 12 rounds of SPECK-32 SPECK. This attack requires a large amount of plaintext and encryption operations. Fu et al. [14] also utilized the *differential cryptanalysis* to attack 1-round, 3-rounds, and 5-rounds of SPECK-64, SPECK-96, SPECK-128. No published research so far claims a successful attack on the full 34-rounds of SPECK-128 cipher. Therefore, SPECK cipher can still be considered secure.

In terms of performance, Dinu et al. [8] confirmed that the SPECK algorithm from the NSA was the smallest and fastest cryptographic algorithm on all platforms. Beaulieu et al. [9] conclude that the SPECK cipher is mostly better in term of system resources usage, including flash usage, RAM, and cost (cycle/byte) when compared to the cipher algorithms such as AES, SIMON, HIGHT, IDEA, TWINE, TEA in AVR ATmega128 8-bit microcontroller.

The hash algorithm used is BLAKE2s, which is, according to Aumasson et al. [15]. This algorithm is designed to have a high security level but is quite light, requires little memory and can be run on an 8-bit processor as is commonly used on Arduino. BLAKE2s is part of BLAKE2 hash algorithm family that is targeted on 8-32-bit platform. Jain et al. [16] conclude that BLAKE2 hash algorithm family is secure and fast, with the performance 2.01x compared to latest standard SHA3-256. Luykx et al. [17] conclude that currently, there is no generic attack on any modes that BLAKE2 uses.

G. *Protocol Design*

This protocol is designed as a data communication protocol that provides an authentication mechanism for both the server and the client by using a *nonce*-based challenge-response authentication method at the initiation phases. This protocol will perform a key agreement function by agreeing to an ephemeral session key, a secret key that is only used during this session. An *ephemeral session key* is derived based on parameters communicated by the server and client. After the communication session is securely established (*session established* state), the client and server can communicate with the traffic encrypted with SPECK lightweight block cipher being used. In this secure communication session, the client can send telemetry data and receive telecommand commands to a Publisher-Subscriber module of the protocol at the server-side, which the Publisher-Subscriber module is based on Redis server. The server will manage the session securely until the session is terminated (session teardown).

The protocol flow design can be described as follows:

1) INIT phase: In the protocol initiation phase, as shown in Fig.3, Arduino-based IoT devices that act as clients will send INIT messages to the server. The client-side protocol sends INIT messages containing the protocol version and supported SPECK key length parameters, as well as random client (client nonce) and key hash values from the nonce client with keys stored permanently on Arduino.

The IoT client will use the *ephemeral client ID* for the initialization message. *Ephemeral client* ID is a temporary ID that helps to conceal the true client ID of this device during each protocol initialization handshake. This temporary ID is calculated based on a BLAKE2 hash of the real client ID and a saved initialization vector. Both are stored in EEPROM. The real ClientID itself is never transmitted.

The saved initialization vector is always changed upon every successful protocol handshake. This results in the next protocol handshake will use different *ephemeral ClientID.*

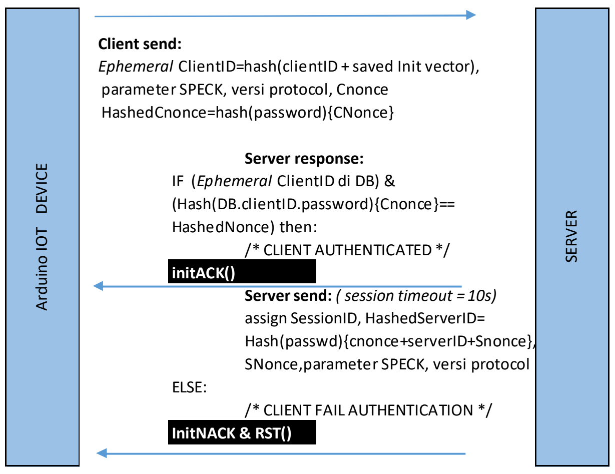

Arduino IOT DEVICE
SERVER
Client send:
Ephemeral ClientID=hash(clientID + saved Init vector),
parameter SPECK, versi protocol, Cnonce
HashedCnonce=hash(password){CNonce}
Server response:
IF (Ephemeral ClientID di DB) &
(Hash(DB.clientID.password){Cnonce}==
HashedNonce) then:
/* CLIENT AUTHENTICATED */
initACK()
Server send: (session timeout = 10s)
assign SessionID, HashedServerID=
Hash(passwd){cnonce+serverID+Snonce},
SNonce,parameter SPECK, versi protocol
ELSE:
/* CLIENT FAIL AUTHENTICATION */
InitNACK & RST()

Fig. 3 Protocol Initialization & Client Auth Phase

Client ID acts as the primary key to differentiate the identity of entities such as one of the IoT devices that the server manages. Bin-Rabiah et al. [18] seem to transmit the Client ID in plaintext so that when the server receives it, the server will be easier to find data related to the ClientID in the database. By transmitting ClientID in plaintext, it is more likely for potential attackers to selectively record communication activities. It is easy to find out the specific ClientID that their communication activities will follow. This protocol proposes to improve the approach by having the ClientID is further disguised by using an *ephemeral ClientID* that is derived from the real ClientID and a saved initialization vector obtained from the previous successful, secure session as shown in Fig.3. Therefore, *ephemeral ClientID* of one session to another will appear to be different despite it comes from the same real ClientID.

2) Server Authentication and Key Agreement phase: In this phase, the protocol on the server-side has successfully verified the client's identity. The server will send a response to the client by assigning a session ID that will be used as a reference for further communication, as shown in Fig.4. This session ID is temporary and only used until both sides to complete the handshaking process. Therefore, it is given a short expiry time. At this phase, the protocol state table still in the INIT phase. The temporary sessionID with short expiry is meant to prevent a potential INIT flood *Denial of Service* attack. INIT flood is possible if too many INIT requests are not completed and stay coming, which shall fill up the server's protocol state table until the system fails. By this mechanism, those INIT requests which fail to complete protocol handshaking before timeout shall be flushed out from the IoTSec protocol state table as a strategy to prevent the INIT flood.

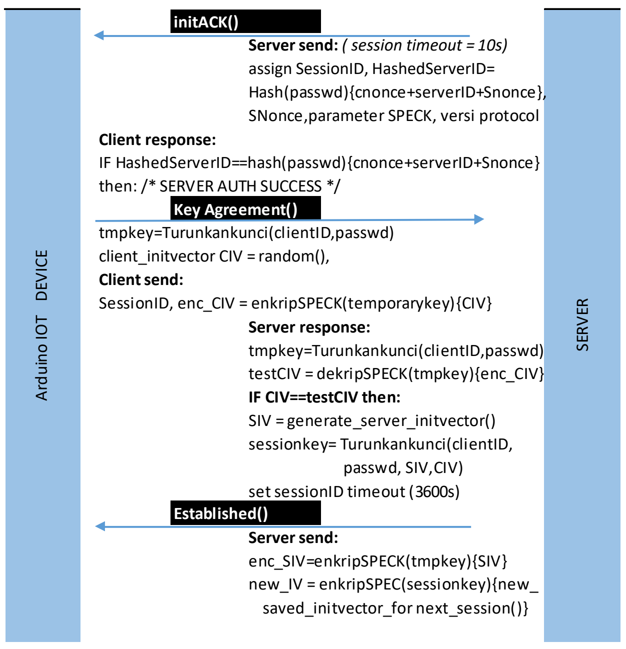

Arduino IOT DEVICE
SERVER
initACK()
Server send: (session timeout = 10s)
assign SessionID, HashedServerID=
Hash(passwd){cnonce+serverID+Snonce},
SNonce,parameter SPECK, versi protocol
Client response:
IF HashedServerID==hash(passwd){cnonce+serverID+Snonce}
then: /* SERVER AUTH SUCCESS */
Key Agreement()
tmpkey=Turunkankunci(clientID,passwd)
client_initvector CIV = random(),
Client send:
SessionID, enc_CIV = enkripSPECK(temporarykey){CIV}
Server response:
tmpkey=Turunkankunci(clientID,passwd)
testCIV = dekripSPECK(tmpkey){enc_CIV}
IF CIV==testCIV then:
SIV = generate_server_initvector()
sessionkey= Turunkankunci(clientID,
passwd, SIV,CIV)
set sessionID timeout (3600s)
Established()
Server send:
enc_SIV=enkripSPECK(tmpkey){SIV}
new_IV = enkripSPEC(sessionkey){new_
saved_initvector_for next_session()}

Fig. 4 Server Authentication and Key Agreement phase

The server will send a keyed hash value of the serverID so that the client can validate the server's validity. The client will then verify the identity of the server, as shown in Fig4. Upon successful verification, the client will initiate a key agreement process by sending a random client initialization vector (CIV) encrypted using the SPECK algorithm with a temporary key derived from the ClientID and password. The server also sends a server initialization vector (SIV) with a similar method. After that, the server and client can proceed to the *key-agreement* process by deriving the *session key* from the keyed-hash of ClientID, ServerID, CIV, SIV, which at this stage both the IoT client and the server have them all. The keyed-hash process will use the BLAKE2s algorithm that uses the password for that IoT device as the key parameter for the keyed-hash process.

The security design of this process lies that the password, ClientID, and ServerID are never transmitted, while CIV and SIV are transmitted in encrypted form with SPECK.

3) Protocol Session Established and Encrypted Communication phase: In this phase, the protocol reaches a state where a secure communication session can happen using the previously agreed *session key* as described in Fig. 5. The client can send encrypted telemetry data, and simultaneously receive telecommand commands on the same transmission. The encryption algorithm used is the SPECK lightweight block cipher. The secure communication session can continue to use the same *session key* until the protocol timed-out state is reached in the server's protocol state table. The flow is shown in Fig.5.

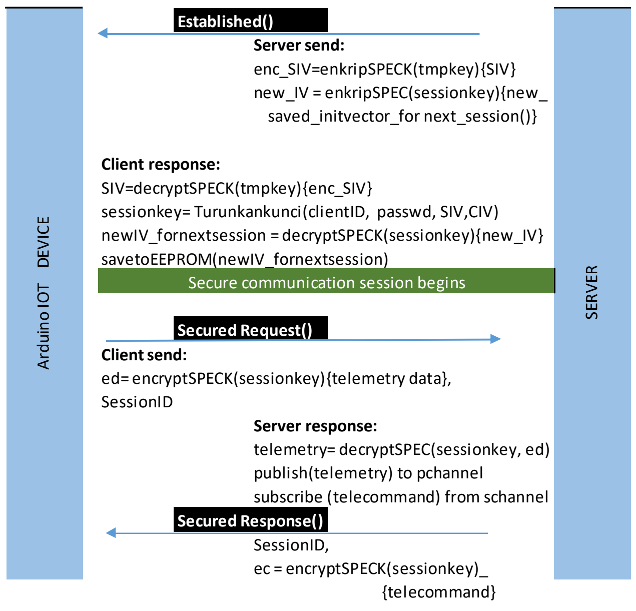

Arduino IOT DEVICE
SERVER
Established()
Server send:
enc_SIV=enkripSPECK(tmpkey){SIV}
new_IV = enkripSPEC(sessionkey){new_
saved_initvector_for next_session()}
Client response:
SIV=decryptSPECK(tmpkey){enc_SIV}
sessionkey= Turunkankunci(clientID, passwd, SIV,CIV)
newIV_fornextsession = decryptSPECK(sessionkey){new_IV}
savetoEEPROM(newIV_fornextsession)
Secure communication session begins
Secured Request()
Client send:
ed= encryptSPECK(sessionkey){telemetry data},
SessionID
Server response:
telemetry= decryptSPEC(sessionkey, ed)
publish(telemetry) to pchannel
subscribe (telecommand) from schannel
Secured Response()
SessionID,
ec = encryptSPECK(sessionkey)_
{telecommand}

Fig. 5 Protocol Session Established & Encrypted Communication

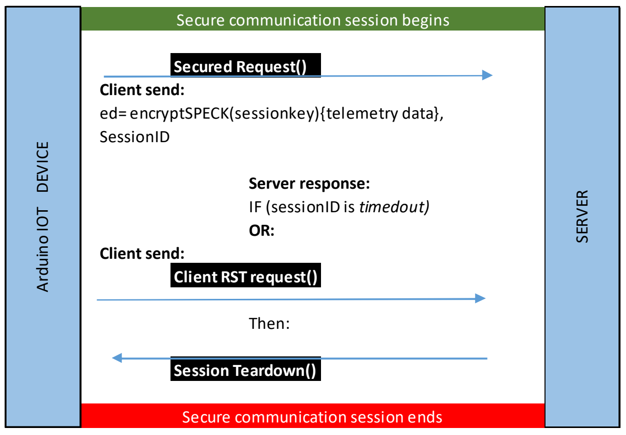

Arduino IOT DEVICE
SERVER
Secure communication session begins
Secured Request()
Client send:
ed= encryptSPECK(sessionkey){telemetry data},
SessionID
Server response:
IF (sessionID is timedout)
OR:
Client send:
Client RST request()
Then:
Session Teardown()
Secure communication session ends

Fig. 6 Protocol Session Teardown

4) *Session teardown phase*: When the session timed-out state is reached, then the server will send a connection termination message when it receives a transmission from the corresponding IoT client. The same thing happens if the client intentionally sends a connection termination request (Fig.6).

III. RESULTS AND DISCUSSION

In the test environment, we take 30 test samples of protocol execution time measurement on both test environment: the Arduino Mega and ESP32 microcontroller. The test results obtained are as follows:

A. *Protocol Performance Evaluation*

The results of the protocol performance measurements as shown in Fig.7, the *INIT* phase averaged 44.63 milliseconds at ESP32 and 26.83 milliseconds on Arduino Mega for the protocol initialization phase as described in Figure 3, which in this phase the Arduino client sends access requests while sending credential to the server for the verification process.

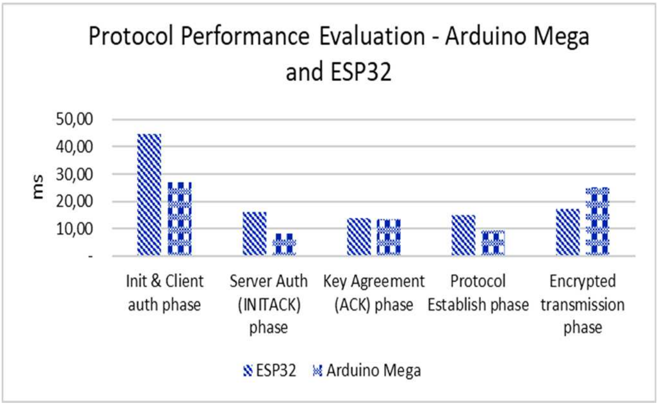


Fig. 7 Protocol Performance Evaluation on each phase

The performance evaluation of the *INIT-ACK* phase, the Arduino ESP32 client on average, takes 16.17 milliseconds to verify the validity of the server. In comparison, Arduino Mega requires an average of 8.27 milliseconds for the same process.

The *Key Agreement* phase (as described in Figure 4) on average, takes 13.90 milliseconds for ESP32 and 13.50 milliseconds for Arduino Mega. The *Protocol-Established* phase (as illustrated in Figure 5) on average, takes 15.30 milliseconds for ESP32 and 9.50 milliseconds for Arduino Mega.

After all the handshaking processes are completed, now the protocol reaches a *Protocol-established* state. The client and server can communicate securely in a process as depicted in Figure 8, using a *session* key that is valid only for this encrypted communication session. The average processing of each encrypted message is 17.10 milliseconds on ESP32 and 25 milliseconds on Arduino Mega for each telemetry encrypted message. The Arduino Mega or ESP32 as IoT client can communicate securely with the server using the same *session key* until the session is marked as expired on the server protocol state table. As the session expires, the protocol will enter the session teardown phase, as described in Figure 6. The client will verify whether the information to terminate the session is valid from the server by checking the results of the hash locking the session key against the current session ID. This is a strategy to mitigate a potential *false reset* attack. The *false reset* attack referred to here is a potential attempt to fool an IoT client to think the server has sent a protocol session termination message by sending a bogus message.

INIT & client identity verification phases, server identity verification, key agreement, and establish protocol are referred to as the overhead phase of this protocol. The phases are the preparation phases of the protocol before actual secure communication begins. After the overhead phases are completed, the next phase is the encrypted communication phase, where the secured data transmission happens. Performance measurements are shown in Fig.8.

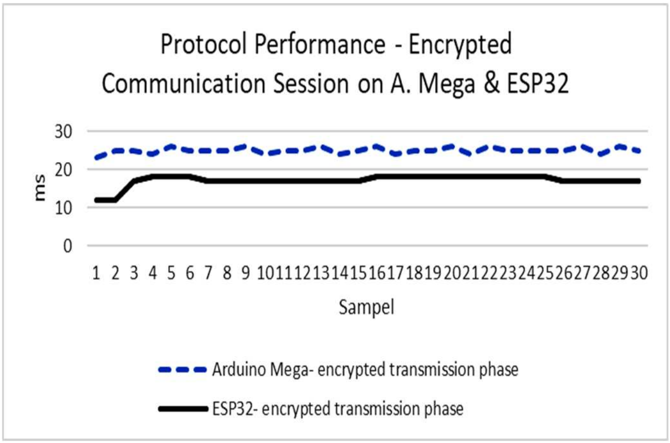


Fig. 8 Protocol communication phase on Arduino Mega and ESP32

It shall be noted that the overhead time for INIT is the largest among the other overhead phases. In this case, the INIT process is carried out the first time and only once when the Arduino IoT client connects to the network. After that, the protocol phases will gradually proceed until the protocol is established to complete all the overhead phases. After all overhead phases are completed, now the protocol state is *established*, in which the client and server can communicate data securely protected using SPECK encryption that uses the *ephemeral session key* as the encryption key. The secure data communication transmissions -both telemetry and telecommand- may happen many times during the *protocol established* state until the protocol timed-out state is reached.

The research data reveal that ESP32 outperforms Arduino Mega for secure data communication in the *protocol-established* phase, the most repetitive phase in this protocol lifecycle. This is consistent with the fact that ESP32 has better processing and memory resources. However, in the *overhead* phases, Arduino Mega outperforms the ESP32 despite having smaller computing resources. This may be an anomaly that may become a potential subject-of-interest for future research.

Suppose a communication network disruption during the session protocol state is still valid (not timed-out). In that case, the protocol shall only need to continue from the last protocol phase when the communication is disrupted. However, in the event of a power failure that causes the IoT device as this client to reboot, then the process will start from INIT again because the current protocol state is reset.

B. *Protocol Security Services Evaluation*

This section focuses on evaluating the proposed protocol's security services: client and server authentication services, data integrity services, and data communication confidentiality services. The test scenario simulates a man-in-the-middle-attack attack, as illustrated in Figure 9.

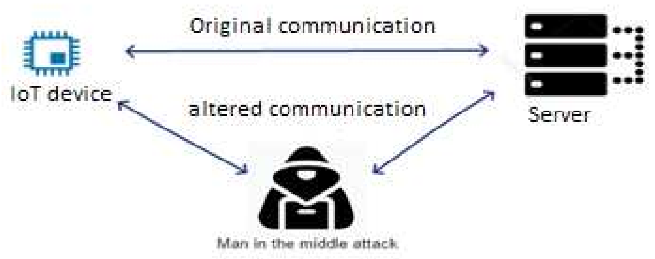
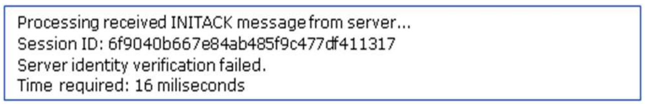


Fig. 9 Man-in-the-middle attack illustration

1) Evaluation of Client-Server Authentication: Client and server authentication process are carried out before the key agreement process is completed, as mentioned in the INIT and INIT-ACK phases, as illustrated in Figures 3 and 4.

In this test, a simulated attacker tries to take over the connection and pretend as if the attacker were the server. The simulated attacker will intercept the normal authentication process and send the parameters required for client-server authentication, such as the server-*nonce* and keyed hash of server-*nonce* required by the client to validate the identity of the server. The results of testing this protocol on the server authentication process, as previously illustrated in Figure 4, the client was able to detect this and display the *Server identity verification failed* message, as shown in Figure 10.

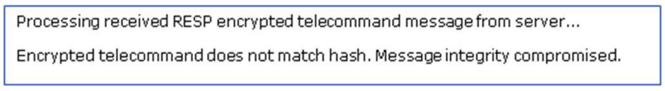


```
Processing received INITACK message from server...
Session ID: 6f9040b667e84ab485f9c477df411317
Server identity verification failed.
Time required: 16 miliseconds
```

Fig. 10 Evaluation of Protocol Client-Server Authentication

2) Evaluation of Protocol Data Integrity: The data transmission process of telemetry and telecommand is secured by SPECK encryption, which uses the session key, as shown in Figure 4. The integrity of the encrypted telemetry message is protected by using the BLAKE2s hash algorithm.

The process of testing data integrity, as described in Figure 11, is performed by simulating an attacker that captures the data communication by an IoT client with the server. The simulated attack tries to tamper with the encrypted message before sending it to the server to simulate man-in-the-middle-attack. The result shows that the protocol can detect that a change has occurred so that its integrity cannot be guaranteed.

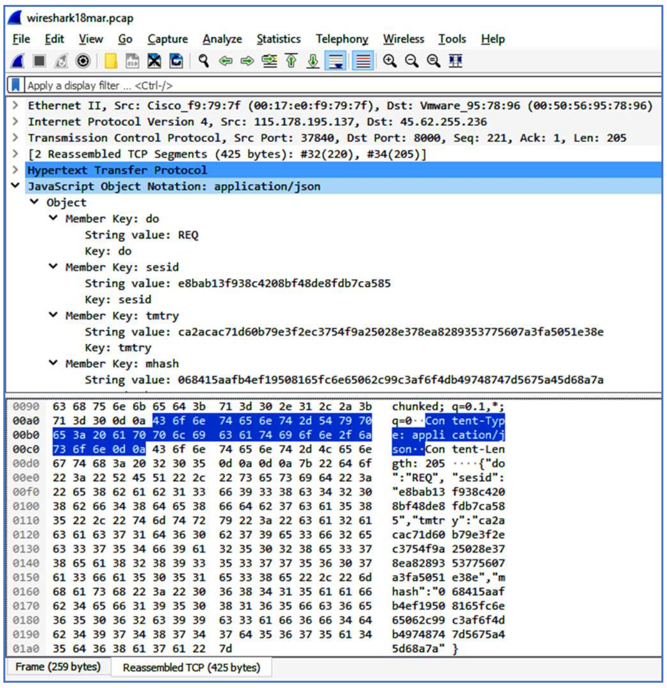


```
Processing received RESP encrypted telecommand message from server...

Encrypted telecommand does not match hash. Message integrity compromised.
```

Fig. 11 Evaluation of Protocol Data Integrity

3) Evaluation of Protocol Data Confidentiality: The data confidentiality testing process is carried out by using the Wireshark tool to record and open encrypted communication transmissions in the established communication protocol phase. When carried out packet analysis using the Wireshark tool, the test results indicate that the data transmission is in an encrypted state, as illustrated in Figure 12.


wireshark18mar.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
> Ethernet II, Src: Cisco_f9:79:7f (00:17:e0:f9:79:7f), Dst: Vmware_95:78:96 (00:50:56:95:78:96)
> Internet Protocol Version 4, Src: 115.178.195.137, Dst: 45.62.255.236
> Transmission Control Protocol, Src Port: 37840, Dst Port: 8000, Seq: 221, Ack: 1, Len: 205
> [2 Reassembled TCP Segments (425 bytes): #32(220), #34(205)]
> Hypertext Transfer Protocol
v JavaScript Object Notation: application/json
  v Object
    v Member Key: do
        String value: REQ
        Key: do
    v Member Key: sesid
        String value: e8bab13f938c4208bf48de8fdb7ca585
        Key: sesid
    v Member Key: tmtry
        String value: ca2acac71d60b79e3f2ec3754f9a25028e378ea8289353775607a3fa5051e38e
        Key: tmtry
    v Member Key: mhash
        String value: 068415aafb4ef19508165fc6e65062c99c3af6f4db49748747d5675a45d68a7a

0090  63 68 75 6e 6b 65 64 3b  71 3d 30 2e 31 2c 2a 3b   chunked; q=0.1,*;
00a0  71 3d 30 0d 0a 43 6f 6e  74 65 6e 74 2d 54 79 70   q=0··Con tent-Typ
00b0  65 3a 20 61 70 70 6c 69  63 61 74 69 6f 6e 2f 6a   e: appli cation/j
00c0  73 6f 6e 0d 0a 43 6f 6e  74 65 6e 74 2d 4c 65 6e   son··Con tent-Len
00d0  67 74 68 3a 20 32 30 35  0d 0a 0d 0a 7b 22 64 6f   gth: 205 ····{"do
00e0  22 3a 22 52 45 51 22 2c  22 73 65 73 69 64 22 3a   ":"REQ", "sesid":
00f0  22 65 38 62 61 62 31 33  66 39 33 38 63 34 32 30   "e8bab13 f938c420
0100  38 62 66 34 38 64 65 38  66 64 62 37 63 61 35 38   8bf48de8 fdb7ca58
0110  35 22 2c 22 74 6d 74 72  79 22 3a 22 63 61 32 61   5","tmtr y":"ca2a
0120  63 61 63 37 31 64 36 30  62 37 39 65 33 66 32 65   cac71d60 b79e3f2e
0130  63 33 37 35 34 66 39 61  32 35 30 32 38 65 33 37   c3754f9a 25028e37
0140  38 65 61 38 32 38 39 33  35 33 37 37 35 36 30 37   8ea82893 53775607
0150  61 33 66 61 35 30 35 31  65 33 38 65 22 2c 22 6d   a3fa5051 e38e","m
0160  68 61 73 68 22 3a 22 30  36 38 34 31 35 61 61 66   hash":"0 68415aaf
0170  62 34 65 66 31 39 35 30  38 31 36 35 66 63 36 65   b4ef1950 8165fc6e
0180  36 35 30 36 32 63 39 39  63 33 61 66 36 66 34 64   65062c99 c3af6f4d
0190  62 34 39 37 34 38 37 34  37 64 35 36 37 35 61 34   b4974874 7d5675a4
01a0  35 64 36 38 61 37 61 22  7d                        5d68a7a" }

Frame (259 bytes)   Reassembled TCP (425 bytes)
```

Fig. 12 Evaluation of Protocol Data Confidentiality

IV. Conclusions

This research shows that a secure IoT communication protocol can be designed and implemented in resource-constrained IoT devices such as the Arduino. Performance evaluation of this IOTSec protocol in Arduino Mega concludes that the INIT phase's average execution time is 26.83 milliseconds, the *key-agreement* phase is 13.50 milliseconds and encrypted message processing during the *protocol established* phase requires 25 milliseconds. Performance evaluation of this IOTSec protocol performance in ESP32 for INIT phase is 44.63 milliseconds, the *key-agreement* phase is 13.90 milliseconds, and encrypted message processing during *protocol established* phase requires 17.10 milliseconds.

This research also proves that the protocol can provide client authentication service to the server and vice versa. This is accomplished securely that the *server-ID, Client-ID,* and *password* is never transmitted in any way because all the verification processes are carried out by a *challenge-response* mechanism utilizing a keyed-hash function. This protocol currently utilizes the BLAKE2s keyed-hash algorithm, as well as the lightweight block cipher SPECK algorithm with 256-bits key size and 128-bit block size of data. This protocol currently supports 128-bit (16 bytes) per message telemetry and telecommand messages.

The research shows that ESP32 outperforms Arduino Mega for the secure data communication phase, which is consistent with the fact that ESP32 has better processing and memory resources. However, for the *overhead* phases, Arduino Mega outperforms the ESP32 despite having smaller resources. This might be an anomaly, which may be an interesting subject for future research.

This protocol can be further developed to add new security services. The addition of this new security service is adapted to the challenges of developing new security threats that continue to emerge from time to time, so the IoT security protocol shall evolve to deal with the new threats.

References

[1] Ahmad J, Zafar F, "Review of body area network technology & wireless medical monitoring." *International Journal of Information and Communication Technology. 2(2).*, 2012.

[2] Yadav G, Devi HMS., "Arduino based Security System – An Application of IOT", International Journal of Engineering Trends and Technology (IJETT) – Special Issue. pp. 209–212. 2017.

[3] Wahjuni S, Maarik A, Budiardi T. "The Fuzzy Inference System for Intelligent Water Quality Monitoring System to Optimize Eel Fish Farming", *Proceeding of The International Symposium on Electronics and Smart Devices. Bandung (ID)*, 2016.

[4] Wahjuni S, Waladi A. "Komiot: Exploring Rest Protocol for IoT Server of The Automatic Control System for Production Land Irrigation.", *Proceedings of The 4th International Seminar on Sciences "Sciences for Green Development" pp.71-81.,* 2017

[5] (2018) ESET We Live Security Website [Online]. Available: https://www.welivesecurity.com/2018/03/02/start-analyzing-security-iot-devices/

[6] Loi F, Sivanathan A, Gharakheili HH, Radford A, Sivaraman, V. "Systematically evaluating security and privacy for consumer IoT devices", *In Proceedings of the 2017 Workshop on Internet of Things Security and Privacy (pp. 1-6)*, 2017.

[7] Wu W, Zhang L. "LBlock: a lightweight block cipher". *International Conference on Applied Cryptography and Network Security (pp. 327-344). Berlin(DE): Springer*, 2011.

[8] Dinu D, Le Corre Y, Khovratovich D, Perrin L, Großschädl J, Biryukov A, "Triathlon of lightweight block ciphers for the internet of things*", Journal of Cryptographic Engineering. pp.1-20.*, 2015.

[9] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B. and Wingers, L. "The SIMON and SPECK block ciphers on AVR 8-bit microcontrollers". *In International Workshop on Lightweight Cryptography for Security and Privacy (pp. 3-20). Springer, Cham.*, 2014

[10] Beaulieu R., Treatman-Clark S, Shors D, Weeks B, Smith J, Wingers L., "The SIMON and SPECK lightweight block ciphers". *52nd ACM/EDAC/IEEE Design Automation Conference (DAC) (pp. 1-6).*, 2015.

[11] Biryukov A, Dinu D, Großschädl J., "Correlation power analysis of lightweight block ciphers: from theory to practice"., *In International Conference on Applied Cryptography and Network Security. (pp. 537-557)*, 2016.

[12] Dinur, I. "Improved differential cryptanalysis of round-reduced speck". *In International Conference on Selected Areas in Cryptography (pp. 147-164). Springer, Cham.* 2014

[13] Dwivedi, A.D., Morawiecki, P. and Srivastava, G. "Differential cryptanalysis of round-reduced SPECK suitable for internet of things devices". *IEEE Access, 7, pp.16476-16486.*, 2019

[14] Fu, K., Wang, M., Guo, Y., Sun, S. and Hu, L. "MILP-based automatic search algorithms for differential and linear trails for SPECK". *In International Conference on Fast Software Encryption (pp. 268-288). Springer, Berlin, Heidelberg.* 2016, March.

[15] Aumasson, J.P., Neves, S., Wilcox-O'Hearn, Z. and Winnerlein, C., "BLAKE2: simpler, smaller, fast as MD5". *In International Conference on Applied Cryptography and Network Security (pp. 119-135). Springer, Berlin, Heidelberg.* 2013, June

[16] Jain, A.K., Jones, R. and Joshi, P.. "Survey of Cryptographic Hashing Algorithms for Message Signing". *Int. J. Comput. Sci. Technol, 8, pp.18-22.* , 2017

[17] Luykx, A., Mennink, B. and Neves, S. "Security analysis of BLAKE2's modes of operation." *IACR Transactions on Symmetric Cryptology, pp.158-176.`*, 2016

[18] Bin-Rabiah A, Ramakrishnan KK, Liri E, Kar K. "A Lightweight Authentication and Key Exchange Protocol for IoT". *Workshop on Decentralized IoT Security and Standards (DISS). San Diego(US)*, 2018.