The Implementation of a Symmetric Lightweight Cryptographic Algorithm called SECURE-BEE Applicable in Communicating Embedded Systems

The Implementation of a Symmetric Lightweight Cryptographic Algorithm called SECURE-BEE Applicable in Communicating Embedded Systems

1st Tobias Glocker
School of Technology and Innovations
University of Vaasa
Vaasa, Finland
tglo@uwasa.fi

2nd Timo Mantere
School of Technology and Innovations
University of Vaasa
Vaasa, Finland
timan@uwasa.fi

***Abstract*—In our modern society, comfort plays an essential role. To increase the comfort, more automation is required. Most of the Automation Systems consist of multiple Embedded Systems, that communicate with each other, either wired or wireless. To ensure confidentiality, a secure communication is required. Since most of the Embedded Systems use Central Processing Units (CPUs) that are more energy efficient and less powerful as Personal Computers (PCs), it is important to use a Lightweight Cryptographic Algorithm (LCA). This paper describes a new implemented LCA, that uses Symmetric Key Cryptography, needs less computation power, and the risk that it can be cracked by a Brute-Force Attack is vanishingly small. The results show the benefits and drawbacks of the developed LCA called SECURE-BEE, in comparison to two other LCAs. It is to mention, that this paper focuses on the implementation of a new LCA. A comparison of the memory usage and the average time for the encryption and decryption has been made only, to show that SECURE-BEE fulfills the encryption and decryption time of LCAs.**

***Index Terms*—Communicating Embedded Systems, Lightweight Cryptographic Algorithm, SECURE-BEE**

I. Introduction

Nowadays, there are many Automation Systems, that are found in smart homes [1] or in modern vehicles [2]. These Automation Systems consist of Embedded Systems, that are connected with each other, either wired [3] or wireless [4]. Besides a reliable communication, it is also important to have a secure communication [5]. In most of the Embedded Systems, the computing power is limited, and therefore it is essential to use LCAs such as the Lightweight Encryption Algorithm (LEA) [6] or the Lightweight Advanced Encryption Standard (Lightweight AES) [7]. All these algorithms need a low computation power and therefore, they are suitable for Embedded Systems. LEA uses ARX operations (modular addition, bitwise rotation and bitwise xor) [8]. However, the encryption and decryption as well as the key schedule follow the same pattern. Although a key size of 128, 192 or 256 bits is considered to be secure, the key could be found using a Brute-Force Attack [9], that counts from zero up to $2^k - 1$, where k is the number of bits used for the key size. This applies also for the Lightweight AES. The advantage of LEA and the Lightweight AES is, that only a 128, 192 or 256 bit key needs to be shared. There are several cryptographic algorithms such as Diffie-Hellman key exchange [10], RSA (Rivest-Shamir-Adleman) [11] and Elliptic-curve Diffie-Hellman (ECDH) [12] that can be used to share a key securely.

In this paper, we have developed a new Symmetric LCA called SECURE-BEE, applicable in Communicating Embedded Systems. In comparison to LEA or the Lightweight AES, our developed LCA is less sensitive to Brute-Force Attacks, but the required information for the encryption and decryption is more than a shared key. Two files need to be exchanged. One file contains the Bit Order Encryption/Decryption, the other file contains the keys for the Periodic Stream Cipher [13]. This can be seen as a drawback. The functionality of the developed LCA called SECURE-BEE is described in Section II. In Section III, the security requirements and considerations for SECURE-BEE are discussed and Section IV contains the performance evaluation of SECURE-BEE. The conclusion can be found in Section V.

II. Functionality of the developed Lightweight Cryptographic Algorithm called SECURE-BEE

The developed symmetric LCA called SECURE-BEE has been designed for CPUs with less computing power, that are mainly used in Embedded Systems. It uses two encryption phases for encrypting a 128 bit message block. In the first phase, there is a Bit Order Encryption while in the second phase, the encrypted message block after the first phase will be additionally encrypted with a Periodic Stream Cipher.

There are two files, the Bit Order Encryption/Decryption File and the Key File. The Bit Order Encryption/Decryption File contains the information to which new position every bit in the message block is moved. Fig. 1 shows the contents of the Bit Order Encryption/Decryption File. Each line holds seven values, that are separated by commas. The first three values

are determining the new position (row, column and starting bit position) of the bit(s), that is/are being moved. The last four values determine the starting position and the number of bits, that are moved to their new positions.

```
// BEGIN BLOCK ENCRYPTION/DECRYPTION (n0)
// WITH 34 LINES (Bit Order Instructions)
[00,34]
// re,ce,posE,rm,cm,posM,numOfBits
// n0(0,0)
3,3,4,0,0,4,4
2,3,0,0,0,0,4
    .
    .
    .
// n0(3,3)
0,3,0,3,3,4,4
1,3,6,3,3,0,2
3,1,6,3,3,2,2
// BEGIN BLOCK ENCRYPTION/DECRYPTION (n1)
// WITH 34 LINES (Bit Order Instructions)
[01,34]
// re,ce,posE,rm,cm,posM,numOfBits
// n1(0,0)
3,2,6,0,0,0,2
1,3,0,0,0,2,6
    .
    .
    .
// n1(3,3)
2,2,7,3,3,0,1
2,3,4,3,3,1,4
1,1,5,3,3,5,3
    .
    .
    .
```

Fig. 1. Contents of the Bit Order Encryption/Decryption File.

The following two matrices are used to illustrate, how the Bit Order Encryption for the first byte with the value 72 ('H') works. Each message block is represented by a 4 x 4 matrix, where each element has a size of one byte. The matrix P is plaintext matrix and the matrix C is the ciphertext matrix. By default, the first byte (element p_{00}) of the first message block is encrypted with the Bit Order Instructions of n0(0,0). According to the instructions of the Encryption/Decryption File, the four most significant bits (Bit_4 to Bit_7) of element p_{00} are copied to the most significant bits of element c_{33} and the four least significant bits (Bit_0 to Bit_3) of element p_{00} are copied to the least significant bits of element c_{23}.

$$P_{4\times4} = \begin{bmatrix} 01001000 & p_{01} & p_{02} & p_{03} \\ p_{10} & p_{11} & p_{12} & p_{13} \\ p_{20} & p_{21} & p_{22} & p_{23} \\ p_{30} & p_{31} & p_{32} & p_{33} \end{bmatrix}$$

$$C_{4\times4} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & 00001000 \\ c_{30} & c_{31} & c_{32} & 01000000 \end{bmatrix}$$

In the Bit Order Encryption/Decryption File, there are Encryption/Decryption Instructions for n message blocks. If the message to be encrypted needs to be split into m message blocks, where m is bigger than n (amount of blocks containing the Bit Order Instructions), then the message block n+1 is encrypted with the same instructions as the first message block (m0). The message block n+2 is encrypted with the same instructions as the second message block (m1). This is the case, when the Bit Order Encryption is done periodically. In other words, the cipher acts as a periodic cipher. However, it is also possible to encrypt each message block with any of the nth Bit Order Instruction blocks. After the Bit Order Encryption each byte of each message block will be encrypted using a Periodic Stream Cipher, that xor's each byte with an eight bit key stored in the Key File (see Fig. 2). The second encryption is applied to increase the security.

```
// 48 Keys
[048]
010 020 100 255 080 050 130 148
040 046 096 200 220 158 153 207
050 043 178 184 067 033 012 009
094 120 156 076 044 021 199 219
099 180 231 089 051 064 252 142
053 243 091 020 251 083 016 171
```

Fig. 2. Contents of the Key File.

When there are not enough bytes to fill the last message block, then padding bytes need to be added. The receiver must know the number of padding bytes. Therefore, the number of padding bytes must be also transmitted.

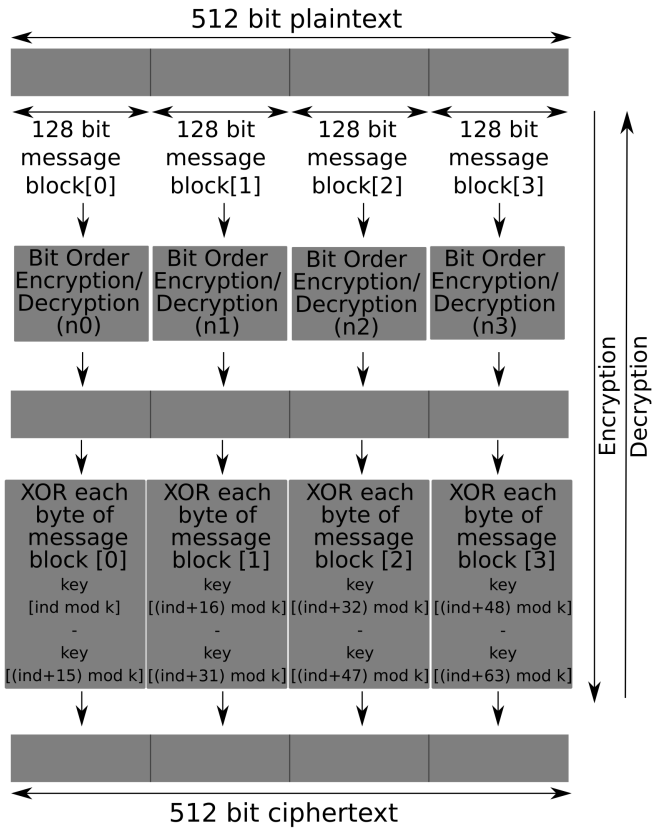

Fig. 3. Bit Order Encryption and Decryption Process.

Fig. 3 illustrates the whole encryption and decryption process of a 512 bit plaintext in detail. First, the 512 bit plaintext is split into four 128 bit message blocks. Each message block is represented by a 4 x 4 matrix, where the size of each element is one byte, as emphasized before. After this step, each message block is encrypted according to its preassigned Bit Order Instructions. As mentioned before, the Bit Order Encryption/Decryption File specifies to which new position each bit of the message block is moved. Then follows the second encryption with the Periodic Stream Cipher, where each message block is additionally encrypted by using an xor operation with an individual key. The variable ind is the key start index, that determines the position in the Key File, from where the first key is read. There is also a variable k, that determines the number of keys in the Key File. In the decryption process, first the Periodic Stream Cipher (xor operation) is applied, before the plaintext with the Bit Order Decryption is generated.

The following pseudocode shows the function implementations for the Bit Order Encryption and Decryption. In this pseudocode, arrays are passed by reference.

```
// global variables
re, ce, posE, rm, cm, posM, numOfBits
bitMaskArray ← {0x00, 0x01, 0x03, 0x07, 0x0F, 0x1F,
                0x3F, 0x7F, 0xFF}
```

```
function INITARRAYS(blockNr)
    numOfLines ← readFromBitOrderFile(blockNr)
    re ← new_array(numOfLines)
    ce ← new_array(numOfLines)
    posE ← new_array(numOfLines)
    rm ← new_array(numOfLines)
    cm ← new_array(numOfLines)
    posM ← new_array(numOfLines)
    numOfBits ← new_array(numOfLines)
    // initialize re, ce, posE, rm, cm, posM and numOfBits
    // with the values of the Bit Order File for the
    // corresponding message block number
    .
    .
    .
end function
```

```
function DOBITORDERENCRYPTION(mBlock(4)(4))
    tempArray ← new_array(4)(4)
    for i ← 0 to (numOfLines − 1) do
        tempArray[re[i]][ce[i]] ←
        (((mBlock[rm[i]][cm[i]] >> posM[i])
        & bitMaskArray[numOfBits[i]]) << posE[i])
    end for
    mBlock ← tempArray
end function
```

```
function DOBITORDERDECRYPTION(mBlock(4)(4))
    tempArray ← new_array(4)(4)
    for i ← 0 to (numOfLines − 1) do
        tempArray[rm[i]][cm[i]] ←
        (((mBlock[re[i]][ce[i]] >> posE[i])
        & bitMaskArray[numOfBits[i]]) << posM[i])
    end for
    mBlock ← tempArray
end function
```

```
function DOXORENDECRYPTION(mBlock(4)(4), ind)
    // XOR Encryption/Decryptions is
    // implemented here!
    .
    .
    .
end function
```

```
// In this example, the message to send consists of only one
// message block! Two padding bytes are added!
function MAIN
    // encryption of message block 0
    messageBlock0 ← {{'H','O','W',' '},
                     {'D','O',' ','Y'},
                     {'O','U',' ','D'},
                     {'O','?','X','Z'}}
    INITARRAYS(0)
    DOBITORDERENCRYPTION(messageBlock0)
    DOXORENDECRYPTION(messageBlock0, 0)
    // send encrypted message to receiver
    // first parameter: message to transmit
    // second parameter: Bit Order Instr. block number
    // third parameter: size of the message in bytes
    // fourth parameter: number of padding bytes
    // fifth parameter: key start index (xor encryption)
    // sixth parameter: receiver identification number
    SEND(messageBlock0, 0, 16, 2, 0, 1)
end function
```

III. Security Requirements and Considerations for SECURE-BEE

SECURE-BEE needs two files for the encryption and decryption. Both files, the Bit Order File and the Key File must be shared between the communicating devices. Therefore, the file exchange should be done securely by using the Secure Shell (SSH) [14] or the Secure File Transfer Protocol (SFTP) [15]. Furthermore, the Bit Order File and the Key File should not be stored in plaintext. There are software tools [16] that can encrypt files on an Embedded Linux System. When running an Embedded Linux Operating System, it is highly recommended to install a firewall. The firewall should be configured so, that only the necessary ports are open. Furthermore, in this application it is not a good idea, to run the SSH or SFTP client(s) and server all the time. Crontab [17] could be used to run the SSH or SFTP client(s)/server only at certain times. Alternatively, pam_time [18] could be used. When using iptables, the time at which certain ports should be open can be configured [19]. It is also a good idea, to implement port knocking [20] or to disable the external access to SSH entirely and mandate the use of a Virtual Private Network (VPN).

IV. Performance Evaluation, Benefits and Drawbacks of SECURE-BEE and Lightweight AES

The performance of SECURE-BEE was tested on a Raspberry Pi 3 Model B [21]. It is an Embedded System with a 1.2 GHz Quad Core CPU (64 bits) and with a Random Access Memory (RAM) of 1 GB. As described in Section II, SECURE-BEE has been implemented for less powerful CPUs, because it does not contain complex computations, that require a lot of computing power. Raspian has been used as an Operating System. To get an accurate result of the encryption plus decryption time, several measurements have been taken, and the average time has been computed. Furthermore, the average encryption plus decryption time of Lightweight AES [22] and LEA has been measured, too. It is to mention, that the average encryption plus decryption time of Lightweight AES has been measured for two different modes, the Counter (CTR) Mode and the Cipher Block Chaining (CBC) Mode. The average encryption plus decryption times of SECURE-BEE and Lightweight AES are shown in Table I. In comparison to the Least Significant Bit (LSB) Steganography [23], where only three out of 24 bits (one pixel) can be used for the encryption, SECURE-BEE has a much lower overhead. For a 512 bit plaintext, encrypted using LSB Steganography, 4104 (roundup(512 / 3) * 24) bits need to be transmitted. When encrypting a 512 bit plaintext with SECURE-BEE, only 512 bits need to be transmitted. In other words, an encrypted 512 bit plaintext is about eight times shorter when using SECURE-BEE instead of LSB Steganography. The less bits have to be sent, the more energy can be saved. For longer plaintexts, SECURE-BEE is even more energy efficient than LSB Steganography, because less bits need to be transmitted. Therefore, the ciphertext can be transmitted on a lower bit rate. According to [24], a lower bit rate results in a lower energy consumption.

TABLE I
Average Encryption plus Decryption Time of SECURE-BEE, Lightweight AES and LEA

Algorithm	**Average Time (Encryption + Decryption)**
SECURE-BEE (512 bit plain-/ciphertext)	621 μs
Lightweight AES (CBC Mode) (512 bit plain-/ciphertext) (key size is 128 bits)	726 μs
Lightweight AES (CTR Mode) (512 bit plain-/ciphertext) (key size is 128 bits)	140 μs
LEA (512 bit plain-/ciphertext) (key size is 128 bits)	26 μs

Table II shows the memory consumption of the algorithm programs in the Random Access Memory (RAM) and the code size on disk. In the RAM, the VmRSS and the VmPeak memory consumption have been measured. VmRss is the resident set size and VmPeak is the peak virtual memory size. It can be clearly seen, that SECURE-BEE has the lowest memory consumption. In terms of the code size on disk for SECURE-BEE it is to mention, that the size of the Bit Order Encryption/Decryption File and Key File has not been taken into account.

TABLE II
Memory Consumption of the Algorithm Programs in RAM and Code Size on Disk

Algorithm	**RAM usage VmRSS**	**RAM usage VmPeak**	**Code Size on Disk**
SECURE-BEE (512 bit plain-/ciphertext)	312 kB	1888 kB	16.384 kB
Lightweight AES (CBC Mode) (512 bit plain-/ciphertext) (key size is 128 bits)	336 kB	1892 kB	20.480 kB
Lightweight AES (CTR Mode) (512 bit plain-/ciphertext) (key size is 128 bits)	348 kB	1892 kB	20.480 kB
LEA (512 bit plain-/ciphertext) (key size is 128 bits)	364 kB	1888 kB	16.384 kB

Table III presents the benefits and drawbacks of SECURE-BEE, Lightweight AES and LEA. The main advantage of SECURE-BEE is the Double Encryption (Bit Order Encryption and Periodic Stream Cipher Encryption) which makes this algorithm very secure. It is faster than Lightweight AES used in CBC Mode. However, two files for the Encryption/Decryption need to be shared between the Communicating Devices. The Bit Order Encryption/Decryption File is used for the Bit Order Encryption/Decryption, while the Key File is used from the Periodic Stream Cipher. Furthermore, it needs to be emphasized, that generating the Bit Order Encryption/Decryption File is complicated. In comparison to SECURE-BEE, Lightweight AES and LEA require only one initial key. Lightweight AES operating in CTR Mode and LEA

TABLE III
BENEFITS AND DRAWBACKS OF SECURE-BEE, LIGHTWEIGHT AES AND LEA

Algorithm	Benefits and Drawbacks
SECURE-BEE	+ Very Secure! Double Encryption enhances the security! + Faster than Lightweight AES used in CBC Mode! + Requires the lowest amount of RAM (VmRSS)! - Generation of the Bit Order Encryption/Decryption File is complicated! - Two files for the Encryption/Decryption need to be shared between the Communicating Devices!
Lightweight AES	+ Requires one initial key! + Faster than SECURE-BEE when used in CTR Mode! - Can be cracked with a Brute-Force Attack!
LEA	+ Requires one initial key! + Faster than SECURE-BEE and Lightweight AES (in CBC and CTR Mode)! - Can be cracked with a Brute-Force Attack!

have a lower encryption/decryption time than SECURE-BEE. Since AES and LEA require one initial key to perform their Encryption/Decryption, they can be cracked with a Brute-Force Attack. However, the time to crack the AES or LEA Cipher will take quite long, because long key lengths are used.

V. CONCLUSION

In this paper, we have described and evaluated our new developed LCA called SECURE-BEE. SECURE-BEE does not contain complex computations, and therefore it is suitable for devices equipped with less powerful CPUs. Two files, the Bit Order Encryption/Decryption File as well as the Key File, must be shared securely between the communicating devices. In the encryption process, there is first the Bit Order Encryption, before each byte of each message block is encrypted with an individual key, stored in the Key File. The decryption process is done in reverse order. Since SECURE-BEE uses the Bit Order Encryption and a Periodic Stream Cipher, it is not so vulnerable against Brute-Force Attacks. However, generating the Bit Order Encryption/Decryption File is complicated. In comparison to LSB Steganography, with SECURE-BEE the length of the encrypted message is much shorter. The encryption/decryption time of SECURE-BEE is shorter than the encryption/decryption time of Lightweight AES (CBC Mode), but longer than the encryption/decryption time of Lightweight AES (CTR Mode) and LEA. Furthermore, SECURE-BEE has the lowest memory consumption in the RAM.

REFERENCES

[1] S. K. Vishwakarma, P. Upadhyaya, B. Kumari and A. K. Mishra, "Smart Energy Efficient Home Automation System Using IoT", 2019 @IEEE, DOI: 10.1109/IoT-SIU.2019.8777607

[2] ResearchGate, "Network vehicle architecture (from ("Automotive Systems Research Group", 2008))." Accessed: May 22, 2023. [Online]. Available: https://www.researchgate.net/figure/Network-vehicle-architecture-from-Automotive-Systems-Research-Group-2008_fig5_264234515

[3] A. He, "Basic Electronics: Wired Communication Protocols in Embedded Design." Accessed: May 22, 2023. [Online]. Available: https://www.seeedstudio.com/blog/2019/07/03/basic-electronics-wired-communication-protocols-in-embedded-design/

[4] Atria University, "What is Wireless Communication? ThinkBig." Accessed: May 22, 2023. [Online]. Available: https://www.atriauniversity.edu.in/types-and-advantages-of-wireless-communication/

[5] Tutorialspoint.com, "Network Security – Overview." Accessed: May 22, 2023. [Online]. Available: https://www.tutorialspoint.com/network_security/network_security_overview.htm

[6] Wikipedia, "LEA (cipher)," Jan. 3, 2023. Accessed: May 22, 2023. [Online]. Available: https://en.wikipedia.org/wiki/LEA_(cipher)

[7] R. S. Salman, A. K. Farhan and A. Shakir, "Lightweight Modifications in the Advanced Encryption Standard (AES) for IoT Applications: A Comparative Survey" in CSASE, 2022 @IEEE, pp. 325-330, DOI: 10.1109/CSASE51777.2022.9759828

[8] A. D. Dwivedi and G. Srivastava, "Differential Cryptanalysis of Round-Reduced LEA" in IEEE Access, 2018, vol. 6, DOI: 10.1109/ACCESS.2018.2881130

[9] T. Gautam and A. Jain, "Analysis of Brute Force Attack using TG – Dataset" in SAI Intelligent Systems Conference, 2015 @IEEE, pp. 984-988, DOI: 10.1109/IntelliSys.2015.7361263

[10] Wikipedia, "Diffie–Hellman key exchange," May 22, 2023. Accessed: May 23, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange

[11] Wikipedia, "RSA (cryptosystem)," May 18, 2023. Accessed: May 24, 2023. [Online]. Available: https://en.wikipedia.org/wiki/RSA_(cryptosystem)

[12] S. Mehrotra and Prof. A. K. Agrawal, "Application of Elliptic Curve Cryptography in Pretty Good Privacy (PGP)" in ICCCA, 2016 @IEEE, pp. 924-929, DOI: 10.1109/CCAA.2016.7813870

[13] Wikipedia, "Stream cipher," Mar. 9, 2023. Accessed: June 13, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Stream_cipher

[14] R. K. Megalingam, S. Tantravahi, H. S. S. K. Tammana, N. Thokala, H. S. R. Puram and N. Samudrala, "Robot Operating System Integrated robot control through Secure Shell (SSH)" in RDCAPE, 2019 @IEEE, pp. 569-573, DOI: 10.1109/RDCAPE47089.2019.8979113

[15] A. S. Gillis, "Secure File Transfer Protocol (SSH File Transfer Protocol)," Oct. 22, 2022. Accessed: May 24, 2023. [Online]. Available: https://www.techtarget.com/searchcontentmanagement/definition/Secure-File-Transfer-Protocol-SSH-File-Transfer-Protocol

[16] M. D. Okoi, "10 Best Linux File and Disk Encryption Tools," Apr. 19, 2023. Accessed: May 24, 2023. [Online]. Available: https://www.tecmint.com/file-and-disk-encryption-tools-for-linux/

[17] Admin's Choice, "Crontab – Quick Reference," 2022. Accessed: May 24, 2023. [Online]. Available: https://www.adminschoice.com/crontab-quick-reference

[18] A. Le Morvan, "PAM Authentication Modules," July 25, 2022. Accessed: May 24, 2023. [Online]. Available: https://docs.rockylinux.org/guides/security/pam/

[19] G. Smith, "16 iptables tips and tricks for sysadmins. Iptables provides powerful capabilities to control traffic coming in and out of your system," Oct. 1, 2018. Accessed: Aug. 3, 2023. [Online]. Available: https://opensource.com/article/18/10/iptables-tips-and-tricks

[20] R. deGraaf, J. Aycock and M. Jacobson, Jr, "Improved Port Knocking with Strong Authentication" in ACSAC, 2005 @IEEE, DOI: 10.1109/CSAC.2005.32

[21] Raspberry Pi, "Raspberry Pi 3 Model B, Single-board computer with wireless LAN and Bluetooth connectivity." Accessed: June 3, 2023. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-3-model-b/

[22] GitHub, "Tiny-AES-c," Jan. 6, 2021. Accessed: June 3, 2023. [Online]. Available: https://github.com/kokke/tiny-AES-c/blob/master/aes.c

[23] M. Nosrati, R. Karimi and M. Hariri, "An introduction to steganography methods" in WAP journal, 2011, vol. 1, no. 3, pp. 191-195. ISSN: 2222-2510

[24] M. A. Hoque, M. Siekkinen and J. K. Nurminen, "Energy Efficient Multimedia Streaming to Mobile Devices – A Survey" in IEEE Communications Surveys & Tutorials, 2014, vol. 16, no. 1, pp. 579-597.