

# IoT Security: Authenticated Lightweight Key Exchange (ALIKE)

Levent Ertaul, Peter Chudinov, and Brian Morales

California State University, East Bay, Hayward, CA, USA

levent.ertaul@csueastbay.edu, pchudinov@horizon.csueastbay.edu, bmorales25@horizon.csueastbay.edu

**Abstract** – IoT security is becoming more and more important as the world relies on computer-based devices. One of the most considerable challenges in today's world is having security and privacy for the Internet of Things (IoT). As of today, there are approximately eight billion IoT devices connected and by the early 2020's it is estimated that there will be 3 to 4 times more IoT devices connected, in which, 25% of cyber-attacks will be targeting IoT devices. Manufacturers are racing to keep up with demands, unfortunately, these devices are equipped with poor security protections creating vulnerabilities. As the power of processors grow and computers become more powerful and efficient the improper security that IoT devices come with will not withstand cyber-attacks. This paper introduces Alike algorithm as a solution to provide lightweight security for IoT devices. In this paper, we will examine the performance analysis of Alike algorithm on a raspberry pi zero. This analysis includes the execution time, memory consumption, CPU utilization, and power utilization.

**Keywords:** Authenticated lightweight key exchange (Alike), IoT security

## 1. INTRODUCTION

The Internet of Things (IoT) devices are currently growing and becoming widespread. An IoT device is a small embedded system that is integrating into our daily lives [1][2]. For example, smart TV's, smart appliances, wearables, smart speakers, etc. are becoming more ubiquitous. These developments are going to have a huge influence on our future and profoundly revamp our environment. Our environment will be heavily influenced by the new cyber-physical world that will result from automated interaction from these devices without human cooperation [3][4]. As we rely on computer-based devices the demand for security and privacy will increase. One of the major mechanisms that is used to solve this issue is cryptography.

The process of converting (encrypting) ordinary messages into indecipherable text and vice versa has been coined cryptography [5]. It is a method of transmitting and storing data in a manner that only the intended recipient or recipients can read and process. In cryptography, key exchange is a process by which cryptographic keys are securely exchanged between two parties and those keys are utilized as a part for some cryptographic algorithm [5][6]. For IoT devices, this has become a major issue because of the inadequate security these devices are equipped with. There are chiefly two different types of encryption methods: symmetric or

asymmetric encryption. Symmetric encryption, also known as secret-key algorithms, commonly require a key to be shared and simultaneously be kept secret within a restricted group [7][8]. Asymmetric encryption, in rudimentary terms, is when the transmitter and receiver hold different keys where at least one is computationally unattainable to derive from the other [9]. Yet, symmetric encryption is widely used today for the reason that they can achieve high-speed or low-cost encryption [7]. Alike algorithm utilizes both: AES (symmetric) and RSA (asymmetric) [10].

In addition, there are other popular key exchange algorithms such as Diffie-Hellman and Elliptic Curve Diffie-Hellman (EC Diffie-Hellman). Though these key exchanges are available, none are lightweight. The necessity of finding a lightweight reliable key exchange algorithm for IoT devices is becoming pervasive especially in the United States.

Recently, the California Legislative branch proposed Senate Bill No. 327. The Bill states, "This bill, beginning on January 1, 2020, would require a manufacturer of a connected device, as those terms are defined, to equip the device with a reasonable security feature or features that are appropriate to the nature and function of the device, appropriate to the information it may collect, contain, or transmit, and designed to protect the device and any information contained therein from unauthorized access, destruction, use, modification, or disclosure, as specified," [11]. IoT security is becoming important in that many cryptography algorithms need to work with the constraints such as, memory and CPU limitations, that IoT devices have. Therefore, there is a search in cryptographic algorithms to find an algorithm that is viable in these constraints. For example, one these algorithms proposed by NIST (National Institute of Standards and Technology) is Alike algorithm [12].

In this paper, we are implementing and analyzing the performance of Alike algorithm, which will be explained in the next section, in an IoT environment. Furthermore, in Section 3 we will discuss the implementation of Alike algorithm in a Raspberry Pi Zero. Section 4 deliberates about the performance and power consumption of Alike algorithm on the Raspberry Pi Zero. Lastly, section 5, we present concluding thoughts on Alike algorithm on the Raspberry Pi Zero.

## 2. AUTHENTICATED LIGHTWEIGHT KEY

### EXCHANGE (ALIKE) ALGORITHM

Alike algorithm is a lightweight key exchange algorithm that applies RSA with the use of AES encryption [12][13]. Lightweight cryptography is an encryption method that features a small footprint and/or low computational complexity [14]. NIST proposed a six-part standard that specifies lightweight cryptographic algorithms for confidentiality, authentication, identification, non-repudiation, and key exchange to which Alike is the solution [12].

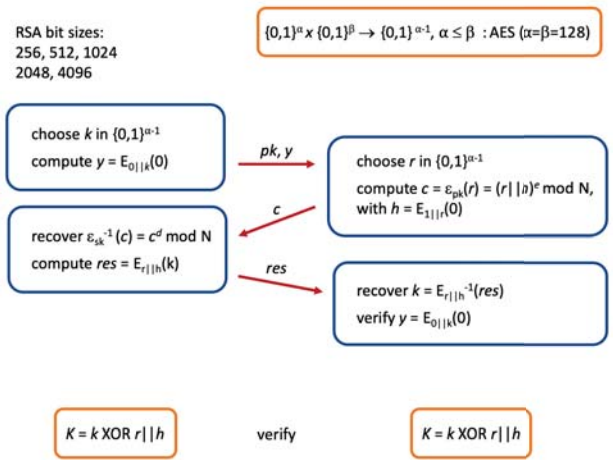
The primitives in the algorithm are: a block-cipher,  $E$ , where  $E$  is  $\{0,1\}^{\alpha} \times \{0,1\}^{\alpha} \rightarrow \{0,1\}^{\alpha}$ ,  $\alpha = 128$  bits for AES and a public key encryption scheme  $\mathcal{R} = \text{RSA}$ .

As shown in Figure 1, first, the algorithm generates a private key,  $sk$ , and a public key,  $pk$ , in the IoT device utilizing RSA key generation algorithm. Afterwards, a number  $k$  is chosen as a primitive, between the bit size of 0 and  $\alpha-1$  - or any number that will not make the difference and append 0's to the beginning ( $0||k$ ), totaling a size of 128 bits. Then encrypt a string of zeros with the result using 128-bit AES and obtain a value  $y$  which will be sent to the computer with the public key,  $pk$ .

The computer receives and saves  $y$ . A random number  $r$  is chosen between the bit size of 0 and  $\alpha-1$ . First, append ones to  $r$ , totaling the size of  $1||r$  to 128 bits, and then use it as key in 128-bit AES, to encrypt string of zeros ( $h = E_{1||r}(0)$ ). Encrypt the result with the public RSA key that was received with  $y$  from the IoT device  $\mathcal{R}_{pk}(r) = (r||h)^e \text{ mod } n = c$ . Note that  $r||h$  has a size of 256 bit. This is important for future AES encryption. The result,  $c$ , is then sent back to the IoT device.

Originally when the IoT device received  $c$ , to recover  $r||h$  the algorithm utilized the prime number  $p$  from the RSA algorithm but instead it was decided to use  $n$ . The modulo  $p$  would recover the original  $r$  disregarding the appended 1's that the  $h$  affixed. Unfortunately, due to the large ratio between  $p$  and  $q$ , the algorithm became computationally challenging and inefficient. Therefore, it was decided to use  $n$  to find the  $r||h$ . Once obtained compute  $res$  by encrypting recovered  $r||h$  with 256-bit AES ( $E_{0||k}(k)$ ). Then send  $res$  to the computer.

The computer recovers  $k$  by decrypting the  $res$  using 256-bit AES with key being  $r||h$  ( $k = E_{r||h}^{-1}(res)$ ). Once  $k$  is recovered, encrypt  $0||k$  with 128-bit AES, and key being a zero 128-bit string ( $E_{0||k}(0) = y$ ) to verify the original  $y$ . Once verified,  $k$  XOR  $r$  to achieve key  $K$ . See figure 1.



**Figure 1:** Alike algorithm. The boxes on the right hand side is the computer, the boxes on the left hand side is the IoT device.

The next section will discuss Alike algorithm implementation on Raspberry Pi Zero to analyze the performance in a resource constraint environment.

### 3. RASPBERRY PI ZERO IMPLEMENTATION OF ALIKE

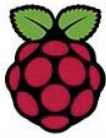
It is decided to implement Alike algorithm on a Raspberry Pi Zero development kit (devkit) shown in figure 2. The Raspberry Pi Zero Wireless comes with 802.11n Wireless LAN, Bluetooth, and BCM2835 [15]. This contains an ARM1176JZFS with floating point, running at 1 GHz, and a video core 4 GPU. [16] We burned the Raspbian image to the SD card, enabled *ssh*, and added network info [17][18].



**Figure 2:** A picture of Raspberry Pi Zero

Raspberry pi Zero with ARM1176JZFS is considered the target platform, which is specifically designed for these environments and supports the specifications mentioned in Table 1.

Table 1 – Raspberry Pi Zero devkit Specification

	Raspberry Pi Zero Wireless
SoC/CPU	BCM2835 1GHz ARM11
GPU	none
RAM	512 MB
Storage	MicroSD
USB	1 Micro USB socket
Ethernet	0
Wi-Fi	802.11n Wireless LAN
Bluetooth	Bluetooth 4.0
Video output	Mini-HDMI
Audio output	none
GPIO	40 (unpopulated)
Size	65 mm x 30 mm x 5 mm
Price	\$10

The Raspberry Pi Zero features many I/O pinouts for component interfacing, however for this study these pins are not used. The Raspberry Pi Zero is assembled with a small microSD card slot. The python program developed for this study uses direct output transmitted over *SSH* with reporting results and debug messages [21]. The Raspberry Pi Zero Wireless was connected via Wi-Fi to a 2015 MacBook pro with Retina Display laptop running MacOS X Mojave. RSA algorithm is considered a secure algorithm due to its factorization properties [19]. Hence, we believe the security of the algorithm is still formidable.

The Alike algorithm was implemented utilizing the latest edition of Python 3.7.2 with the Raspbian operating system installed on the Raspberry Pi Zero Wireless [18][20]. Raspberry Pi was run headless (a.i. without a monitor) and was fully configured and controlled via *SSH*. File transfer (code has been initially written and tested on the aforementioned Mac OS machine) was accomplished via the use of *sftp* utility [21]. In order to get the most adequate results, Raspberry Pi's uptime was kept as low as possible and it was rebooted after every 5 runs of the program.

As shown in figures 3 and 5, the RSA and AES encryption we employed were elicited from the PyCrypto library, which was the only non-standard library we selected [22]. Random values came from Python's standard library - *random* module. The AES algorithm is administered in ECB mode [22]. To apply RSA, we were required to adjust the size of the message to fit under PyCrypto's standards, which enforced us to apply a padding algorithm [23]. Specifying byte order was important because different systems have different ways of reading payloads and we want to make sure our code is reusable on different types of computers.

```
k = random.getrandbits(K_SIZE)
# using 'big' byteorder
k_bytes = k.to_bytes(k_byte_size, BYTEORDER)

zeroes_with_k = bytearray(alpha_byte_size - k_byte_size)
zeroes_with_k.extend(k_bytes)

k_aes = AES.new(bytes(zeroes_with_k))
y = k_aes.encrypt(bytes(alpha_byte_size))
```

Figure 3: AES implementation

We used relatively small size  $r$  (16 bit) and  $k$  (32 bit) but their size should not affect the performance of the program. As you can see from the figure 4, generating zeroes was done with plain *bytes* method, included in Python's standard library.

```
ALPHA_SIZE = 128
K_SIZE = 32
R_SIZE = 16
N_SIZE = 1024
BYTEORDER = 'big'
CTR_SIZE = ALPHA_SIZE
```

Figure 4: Test case constants

Generating ones, however, required a little trick - *byte array* method with '\xFF'. See figure 5.

```
r = random.getrandbits(R_SIZE)
r_bytes = r.to_bytes(r_byte_size, BYTEORDER)
ones_with_r = bytearray(b'\xFF' * (alpha_byte_size - r_byte_size))
ones_with_r.extend(r_bytes)

h_aes = AES.new(bytes(ones_with_r))
h = h_aes.encrypt(bytes(alpha_byte_size))

r_h = r_bytes + h
c = rsa_pk.encrypt(r_h, 32)
```

Figure 5:  $0//r$  generation and RSA encryption

#### 4. ALIKE ALGORITHM PERFORMANCE ANALYSIS

Execution time for Alike algorithm is tested on Raspberry Pi Zero and results are assessed. Performance of the implementation was calculated using *line\_profiler* and *memory\_profiler* in python [24][25]. Private and public keys are generated using python's RSA algorithm with exception for 256 and 512-bit keys. Key size under 1024 bits is considered unsafe by PyCrypto developers, so an additional Ruby script was introduced in order to generate small RSA keypairs [26][27]. Results are evaluated below.

Table 2: Total Execution time

RSA (bits)	Time (secs)	Clock (peak %)	RAM (MB)
256	0.025598	62.26	1.0656
512	0.0306904	68.98	1.0948
1024	0.245425	76.02	1.1034
2048	0.3790166	86.68	1.165
4096	0.9041591	93.8	1.1356

From Table 2 we can see that, as the RSA key size increases, the execution time increases exponentially. Since *memory\_profiler* takes both base Python and executable code in account, we used an empty “Hello World” program to determine the baseline memory consumption. In an average of 5 trials it turned out to be 25.617 MB on our Raspberry Pi Zero Wireless. Then, we subtracted the averaged baseline value from values we have got with *memory\_profiler*.

We have also used *line\_profiler* to measure peak CPU usage. As it turns out RSA decryption is the “heaviest” process in code due to arithmetic involved. Increasing the RSA key size increases the execution and memory/CPU usage accordingly. Interestingly, increasing RSA from 2048 to 4098 almost triples the execution time.

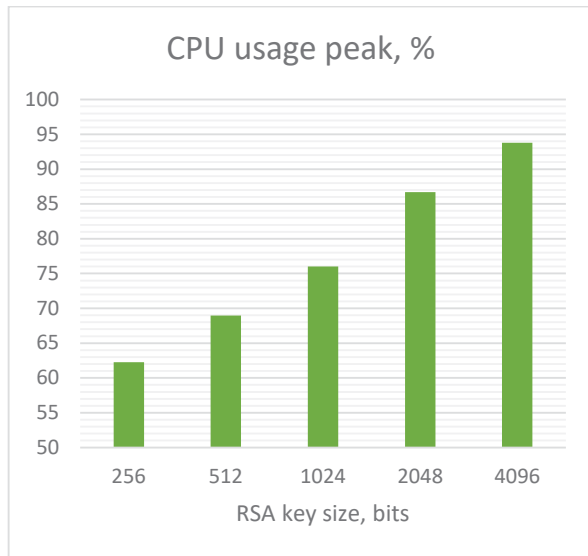


Figure 6: Peak CPU load.

As we can see in figure 6, peak CPU usage grows proportionally to the key size used.

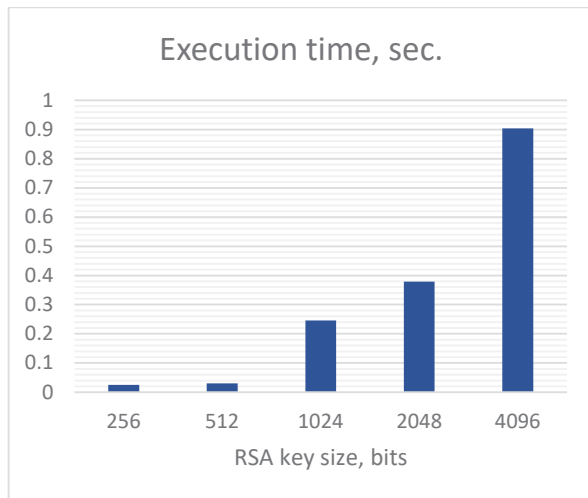


Figure 7: Execution Time.

Figure 7 suggests that execution time exponentially increases when larger key sizes are introduced: with 256 and 512-bit

key execution time is less than 0.031 seconds. When 1024-bit encryption only takes around 0.25 seconds to process, 4096-bit encryption takes 0.9 seconds.

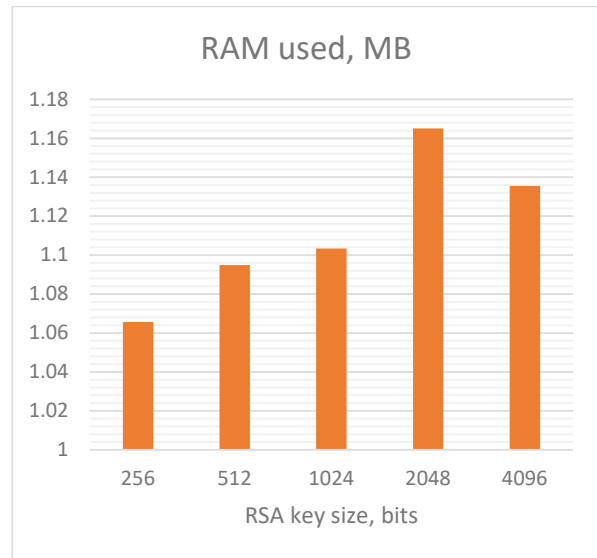


Figure 8: RAM usage in megabytes.

From figure 8, the use of memory varies within 50 kilobytes with 2048-bit encryption surprisingly using the most - 1.165 MB - which is not critical. We suspect that the reason behind 2048-bit encryption being the heaviest on RAM is that despite the key being bigger, the primes randomly generated were smaller. Additionally, it’s possible that the values we are using in  $\alpha$  in the encryption for 4096 is significantly smaller than the  $\alpha$  being used in 2048 since its being randomly generated. Hence, that is why we conjecture that 2048 utilizes a considerable amount of RAM.

In order to estimate power consumption, we will use a simple formula that takes voltage (5.1V), current (1A) and amount of clock cycles and returns joules. The voltage and amperage were taken from the power adapter that came with Raspberry Pi, and clock cycles are calculated by multiplying execution time by processors clock rate, which in case of our Raspberry Pi Zero Wireless is 1GHz.

$$Cycle\ count = \frac{execution\ time}{clock\ rate} \quad (1)$$

Table 3: Average clock cycle count by RSA size

RSA (bits)	Clock cycles
256	25,598,000
512	30,690,400
1024	245,424,509
2048	379,016,638
4096	904,159,069



Using the given equations (1) we estimate the power of Alike algorithm on Raspberry Pi [28].

$$V_{cc} = 3.3V$$

$$I = 0.12A$$

$$Power = 0.12A * 3.3V = 0.396 \text{ Watts} \quad (1)$$

Multiplying the following power consumption with the execution time, average power consumption can be calculated.

$$\begin{aligned} \text{Power consumption in Joules} \\ = 0.396W * \text{Execution time} \end{aligned}$$

$$\begin{aligned} \text{Average power consumption at 4096 bit RSA:} \\ 0.3580452 \text{ Joules} \end{aligned}$$

Knowing that an average AA battery yields around 12960 Joules [29], this algorithm will consume 1425.6 joules per hour. We can conclude that an average AA battery will only last about 9 hours, which is quite insufficient for an IoT device assuming that Alike algorithm is running continuously. On the contrary, Alike algorithm will not be running continuously so the power consumption could be fine.

## 5. CONCLUSION

In this study we have evaluated the performance of Alike algorithm on a Raspberry pi Zero with ARM1176JZFS processor with respect to execution time, CPU usage, memory and power consumption. These factors play a significant role in implementing Alike algorithm because of the constrained resources in IoT devices. Evidently, RSA key generation requires more memory and time due to the mathematical complexity. Fortunately, key generation is done once on the computer end of this algorithm, hence IoT devices shouldn't not have an issue with memory consumption. Considering for IoT devices Alike algorithm can be implemented on Raspberry Pi Zero applications with moderate processing time with peak memory and CPU utilization.

Favorably, Raspberry Pi Zero has sufficient amount of memory and CPU that Alike algorithm is viable. In the case of IoT devices we conclude that our implementation of Alike algorithm will most likely be staggering for the IoT device if applying RSA keys greater than or equal to 1024. In addition, the power the algorithm utilizes is ideal for IoT devices. From our analysis, if we were to run Alike algorithm continuously the algorithm will drain a AA battery very quickly but, fortunately, Alike will not be running continuously. The 256 and 512 RSA key size will make Alike algorithm lightweight and feasible.

## REFERENCES

[1] G. Saldamli, L. Ertaul and B. Kodirangaiah. "Post-Quantum Cryptography on IoT: Merkle's Tree Authentication". Int'l Conference Wireless Networks. 2018, 40, 286.

[2] A. Zanelle, N. Bui, A. Castellani, L. Vangelista, M. Zorzi. "Internet of Things for Smart Cities". IEEE Internet of Things Journal, Vol. 1, February 2014

[3] H. Petersen, E. Baccelli, and M. Wählisch. "Interoperable Services on Constrained Devices in the Internet of Things". In W3C, editor, W3C Workshop on the Web of Things, Berlin, Germany, June 2014.

[4] Ericsson, "More than 50 billion devices," Ericsson White Paper, Tech. Rep., 2011.

[5] S. Kumari. "A Research Paper on Cryptography Encryption and Compression Techniques". International Journal Of Engineering and Computer Science, Vol 6, Issue 4, April 2017

[6] J. Gaba, N. Rani, M. Kumar. "A Review Based Study of Key Exchange Algorithms". International Journal of Recent Trends in Mathematics & Computing, Vol 1, Issue 1, October 2012

[7] A. Mouloudi. "NEW SYMMETRIC ENCRYPTION SYSTEM BASED ON EVOLUTIONARY ALGORITHM". International Journal of Computer Science & Information Technology (IJCSIT) Vol 7, No 6, December 2015

[8] S. Pavithra, E. Ramadevi. "Study and Performance Analysis of Cryptography Algorithms". International Journal of Advance Research in Computer Engineering & Technology Vol 1, Issue 5, July 2012

[9] G. J. Simmons. "Symmetric and Asymmetric Encryption" (PDF). Computing Surveys, Vol 11, No. 4. December 1979.

[10] F. Shao, Z. Chang, Y. Zhang. "AES Encryption Algorithm Based on the High-Performance Computing of GPU". Second International Conference on Communication Software and Networks, February 2010

[11] N.p n.p N.d. (2018) "Senate Bill No. 327"

[12] K. A. McKay, L. Bassham, M. S. Turan, N. Mouha. (2017), NISTIR 8114, Report on Lightweight Cryptography. <https://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf>

[13] S. Agaglate. "ALIKE: Authenticated Lightweight Key Exchange PowerPoint" GEMALTO Security Labs. N.p. n.d.

[14] O. Toshiko, "Lightweight Cryptography Applicable to Various Iot Devices". 2017

[15] T. Klosowki. "The Raspberry Pi Zero W Adds Wi-fi and Bluetooth to the zero, Cost \$10". Article. <https://lifelifehacker.com/the-raspberry-pi-zero-wireless-adds-wi-fi-and-bluetooth-1792789503>

[16] "RPi Zero Hardware General Specifications" N.p. n.d. Web. <https://raspberrypi-projects.com/pi/pihardware/raspberry-pi-zero/raspberry-pi-zero-hardware-general-specifications>

- [17] N.p. n.p. N.d. “*Raspbian Installer*”. [Online] Available: <https://raspbian.org/RaspbianInstaller>
- [18] Allen, Mitch. (N/A) “*Headless Pi Zero Wifi Setup (Windows)*”. [Online] Available: <https://desertbot.io/blog/headless-pi-zero-w-wifi-setup-windows>
- [19] M. Preetha, M. Nithya (2013) “*A Study and Performance Analysis of RSA Algorithm*” IJCSMC, Vol. 2, Issue. 6, pg. 126-139
- [20] N.p. (2018) “*Python 3.7.2*” [Online] Available: <https://www.python.org/downloads/release/python-372/>
- [21] N.p. n.p. N.d. “*SFTP - Raspberry Pi Documentation*” [Online] Available: <https://www.raspberrypi.org/documentation/remote-access/ssh/sftp.md>
- [22] “*PyCrypto Documentation*”. [Online] Available: <https://pypi.org/project/pycrypto/>
- [23] N.p. (2018) Python Crypto: Using AES - 128 in ECB mode. [Online] Available: <https://techtutorialsx.com/2018/04/09/python-pycrypto-using-aes-128-in-ecb-mode/>
- [24] “*Line\_Profiler 2.1.2*” [Online] Available: [https://pypi.org/project/line\\_profiler/](https://pypi.org/project/line_profiler/)
- [25] N.p. (2018) “*Memory-Profiler 0.55.0*” [Online] Available: <https://pypi.org/project/memory-profiler/>
- [26] N.p. “*Welcome to open SSL*” [Online] Available: <https://www.openssl.org/>
- [27] J. Britt, Neurogami. “*Open SSL*” [Online] Available: <http://ruby-doc.org/stdlib-2.0.0/libdoc/openssl/rdoc/OpenSSL.html>
- [28] K. Nisimova. “*Energy of a 1.5 V Battery*”. 2001
- [29] B. Gebeau. “*How Many AA Batteries Would it Take to Power a Human?*” N.d.