Performance Analysis of Lightweight Cryptographic Algorithms in IoT

Hazel Leonard and Dana Estra
Tufts University
hazel.leonard@tufts.edu dana.estra@tufts.edu

Abstract

In the past few years, the market for Internet of Things devices has expanded significantly. With the growing ubiquity of these devices and their continuous sending and receiving of personal data, it is extremely important that these devices utilize proper encryption to keep data secure. Studies have shown that the vast majority of IoT fails to strongly encrypt data, reason being that the small computers lack the resources required to perform standard encryption algorithms. In this paper, we analyze the performance of two novel lightweight algorithms, TinyJAMBU and Elephant, across message size and gcc compiler optimization. These algorithms are designed to perform well in resource-constrained environments such as IoT devices. We also compare these algorithms against the standard non-lightweight AES. Our study finds that AES performs the best; however, this may be due to AES' highly optimized software. Among the lightweight algorithms, we find that the 128-bit key size version of TinyJAMBU has the best performance, however the 256-bit key size version may work well if message size is not expected to be much larger than 32KB. We also find that compiler optimization has most likely an insignificant effect on performance.

1 Introduction

The Internet of Things (IoT) is a network of devices that wirelessly communicate with each other through the internet. These devices typically perform some special function, such as measuring a person's heartrate or sensing the temperature of a room, and then send this data to the cloud or to another device to be processed. Importantly, this data is transferred without being instigated by human interaction. Many IoT devices are designed to facilitate our lives in some way; for example, smart home security cameras might send video to a server, where an application analyzes it to determine if a package has arrived, in which case a message is sent to the user's smart phone. IoT's capability to make processes more efficient accounts for its projected expansion to 43 billion devices by 2023 [6].

At the heart of IoT is the sending of data between devices. The massive amounts of data, including highly personal data, that are being sent and received by IoT makes it imperative that this information is properly secured against security attacks. Even when an IoT device does not transmit personal data, it is still critical that it remain secure against attacks. This is necessary to defend against the large number of attackers that aim to gain control over many IoT devices for the purpose of carrying out a larger scale operation. "Bot Nets" are known for having been used for email spam delivery, distributed denial-of-service (DDoS) attacks, password cracking, key logging, and cryptocurrency mining [3].

Despite the importance of securing IoT, the vast majority of IoT devices remain inadequately secured. Security vulnerabilities found in IoT include the lack of enforcement of strong passwords, the lack of a privilege hierarchy for reading and writing data, and weak encryption of transmitted data [3]. As of 2014, 70% of IoT failed to encrypt local and remote traffic communications, remaining a worrying trend since [3]. This paper focuses on addressing these encryption concerns.

Manufacturers struggle to encrypt communication while not sacrificing performance, given that IoT devices tend to have much lower computation power and available memory relative to other types of computers. As a result of the struggle to implement strong encryption in IoT, an area of research has recently developed for designing lightweight cryptographic algorithms [8]. According to the National Institute of Standards and Technology (NIST), lightweight cryptographic algorithms should exhibit the same level of security as what is expected from a non-lightweight algorithm [9]. Lightweight algorithms have been released for implementation in both software and hardware.

2 Related Work

Past research has been dedicated to evaluating the performance of standard (non-lightweight) encryption mechanisms in IoT environments. For example, Anaya et al., 2020 studied the performance of AES, RSA, ChaCha20, and Twofish, finding ChaCha20 to be the most efficient for IoT [2]. A smaller amount of research has focused specifically on lightweight cryptography in IoT. Lightweight algorithms, including the symmetric key algorithms LEA and Speck, were studied in Chan, 2021 [5]. Additionally, Naru et al., 2017 provides a meta-analysis of recent research into lightweight algorithms in IoT devices [8].

3 Project Description

In contrast with past research, we aim to study the performance of a collection of novel lightweight algorithms chosen as finalists in NIST's lightweight cryptography competition during March of 2021. The contest was launched to distinguish lightweight cryptographic algorithms best-suited for environments with constrained resources, such as IoT devices. For an algorithm to reach the final round, it must reach the same level of security as what is expected for non-lightweight algorithms. We intended to compare the software implementations of several of these

lightweight algorithms to each other and to the non-lightweight AES. Additionally, our goal was to study the performance of these algorithms in relation to the size of the message being encrypted and in relation to the GCC optimization level used to compile the implementations. Being that these are novel algorithms, they have not been thoroughly optimized like other more established algorithms have been, therefore it is worthwhile to examine if compiler optimizations can improve their performance.

Ten finalists were selected from the original 57 submissions to the NIST contest. Of these ten algorithms, we chose two to test: TinyJAMBU and Elephant. We provide an overview of these algorithms, as well as AES, below:

- **TinyJAMBU** A symmetric-key block cipher algorithm. It offers implementations with 128-bit, 192-bit, and 256-bit key sizes, all of which we test in this study.
- **Elephant** A symmetric-key block cipher algorithm. It offers multiple implementations with different permutation sizes. For this study we used the 200-bit permutation size, which the developers of Elephant report to be optimized for software.
- **AES** The standard symmetric-key block cipher algorithm set by NIST. For this study we analyzed the 128-bit key size version.

We selected TinyJAMBU because of its low implementation cost even among lightweight algorithms. JAMBU, the encryption algorithm that serves as the basis for TinyJAMBU, was "the smallest block cipher authenticated encryption mode in the CAESAR competition, and it was selected to the Third Round of the competition"[10].

The CAESAR competition aims to select the strongest authenticated ciphers, which both ensure data's confidentiality and authenticity. Authenticated ciphers achieve this by establishing a message encryption scheme along with a message authentication code that is used to determine a message has not been tampered with and has come from the sender who claims to have transmitted it. TinyJAMBU is an even smaller version of JAMBU, with a state size two-thirds that of JAMBU, a message block size half that of JAMBU, and better authentication security than JAMBU[10].

Along with TinyJAMBU, we selected the Elephant encryption algorithm because of its general similarities to TinyJAMBU but also due to smaller differences in its implementation. Both are authenticated, nonce-based, encrypt-then-MAC, symmetric-key block encryption schemes. Both require a key, nonce, associated data and plaintext as inputs to their encryption and decryption functions; however, their approaches to using these components differ.

3.1 Elephant Algorithm Details

Elephant is a parallelizable scheme that guarantees the integrity of ciphertext. It uses a cryptographic permutation to encrypt, which means the characters in the plaintext message are rearranged. The 80-round Spongent permutation developed previously by researchers Bogdanov et al. is used for the 160-bit and

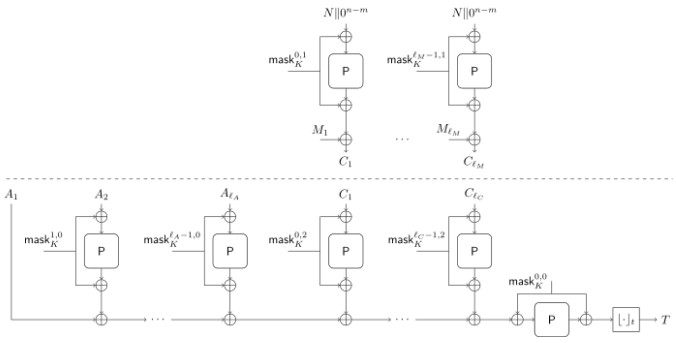

Figure 1: **Depiction of Elephant.**

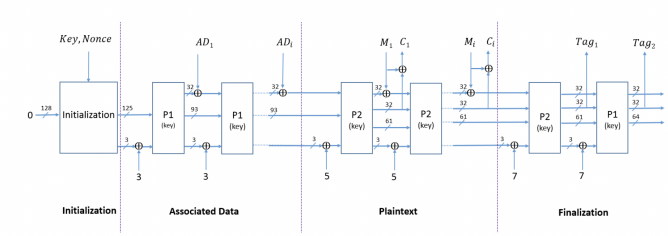

Figure 2: **Depiction of TinyJAMBU.**

176-bit versions, and the 18-round Keccak permutation developed by the Bertoni et al.team is used for the 200-bit version[4].

The result of these permutations is then masked using linear-feedback shift registers (LFSRs) whose starting state is generated in an approach described in previous work by Granger et al [4]. LFSRs consist of a register whose contents are shifted at set intervals: when one bit is outputted, the LFSR uses a linear function to add a new bit to the register based on the register's current state. Next, a series of XOR operations and shifts are performed using the key, nonce, permuted text, and LFSR. This permutation is only conducted in the forward direction, and a separate but similar algorithm is used for decryption.

Elephant offers permutation sizes of 160, 176, and 200 bits, with the 160-bit version "Dumbo" the primary submission. The 200-bit instance is especially optimal for software, making it especially relevant to our project [4]. However, one aspect of Elephant to note is that "if the nonce is reused for two different evaluations of [the encrypt function], security is void[4]." Despite this, the fact that Elephant is parallelizable makes it unique among other permutation-based encryption modes.

3.2 TinyJAMBU Algorithm Details

On the other hand, TinyJAMBU's cipher is based on a keyed permutation, which uses a key to generate a unique permutation for each input. The permutation is accomplished through multiple rounds, one of them using a nonlinear feedback shift register (NFSR). NFSRs may be better at preventing cryptanalysis compared to LFSRs, which Elephant uses. The cipher also uses Framebits, which are three bits set to a different constant depending on the phase of encryption or decryption [10].

To encrypt, the state must first be initialized through being randomized by the keyed permutation. Then, the Framebits for the nonce stage are XORed with the state, the state is updated

Algorithm	Block size	Key size	Nonce size	State size	Optimized?
TinyJAMBU 128	32 bits	128 bit	96 bits	128 bits	Yes
TinyJAMBU 192	32 bits	192 bit	96 bits	128 bits	Yes
TinyJAMBU 256	32 bits	256 bit	96 bits	128 bits	Yes
Elephant 200	200 bits	128 bits	96 bits	200 bits	No
AES	128 bits	128 bits	No nonce	128 bits	Yes

Table 1: **Parameters, Block, and State Size**

using a different keyed permutation, and then 32 nonce bits are XORed with the state [10]. The associated data, which provides context for the encrypted information and ensures that it is not used in an unintended way, is processed using the same sequence. The plaintext is then encrypted using similar steps: the Framebits for the plaintext stage are XORed with the state, the state is updated with the keyed permutation, 32 bits of plaintext are XORed with the state, and plaintext is XORed with another part of the state to produce 32 bits of ciphertext. Finally, the authentication tag is generated.

To decrypt, the state initialization and associated data are processed in the same way as in encryption. 32 bits of plaintext are then obtained by XORing the ciphertext and 32 state bits before XORing the generated plaintext with different state bits. Finally, an authentication tag is generated and compared to the tag received. Verification fails if the tags are different. TinyJAMBU offers 128-bit, 192-bit, and 256-bit key sizes. Unlike Elephant, TinyJAMBU actually provides better authentication security than JAMBU when nonce is reused[10].

3.3 AES Algorithm Details

We chose to compare these lightweight algorithms to AES, the standard encryption algorithm first built by the U.S. government to encrypt sensitive information but now applied to the transfer of all types of data over the internet due to its speed and security. AES, similar to TinyJAMBU, is a symmetric block cipher that offers key sizes of 128-bit, 192-bit, and 256-bits.

AES is processed in bytes, and its state is recorded in a two-dimensional array of bytes on which encryption or decryption is carried out. To encrypt, input is copied to the state array, which is transferred in a number of rounds according to the key size. Each round consists of substituting bytes, shifting the last three rows of the state by different amounts, modifying columns via multiplication, and then finally applying a "round key" to each column using an XOR operation[1]. These "round keys" are generated by applying AES' key expansion routine to the cipher key provided as input to the function. To decrypt, the cipher transformations can be "inverted and implemented in reverse order[1]".

3.4 Experimental Design

Despite its benefits, AES is not built specifically for lightweight cryptography, so we were interested in learning how it performs on IoT devices. Taking inspiration from IoT Design Framework: Model of End-to-End Encryption with Protocols (MEEEP) paper [5], we chose to measure the encrypt and decrypt time of these algorithms while differentiating our work by not using protocols. All encryption algorithms were tested using their software implementations. We chose to test TinyJAMBU's 128-bit, 192-bit, and 256-bit versions to investigate how an increase in key size affects encryption and decryption time. We then tested Elephant's 200-bit version, as it is optimal for software use, and AES' 128-bit version. This version of AES is the commonly used and contains both a 128-bit key size and 128-bit block size. Along with these different versions of the algorithms, we swept message sizes of 32, 320, 3,200, 32,000, and 320,000 bytes with messages containing random characters. The length of associated data was increased according to message size. We also swept compiler optimizations, using the GCC compiler and compiling separate executables with optimization flags O0, O1, O2, and O3.

3.5 Code Details

To time the lightweight algorithms, we downloaded the optimized version of each's C source code (if available) from the NIST website. We used the gettimeofday() function to save the time of day before encryption and after decryption for each algorithm. We performed encryptions and decryptions in a loop, so we obtained the total encrypt and decrypt time for a number of iterations—in this case 50—then divided by the number of iterations to obtain the average time for a single encrypt-decrypt cycle in microseconds (μs).

Both the TinyJAMBU and Elephant implementations took the same arguments to their encrypt and decrypt functions: a pointer to the location to store ciphertext, the length of ciphertext, a pointer to plaintext, the length of plaintext, a pointer to associated data, the length of the associated data, a pointer to the secret message number, public message number, and secret key number. The public message number is used as the nonce, and though the secret key number is passed into each encryption and decryption function, it is not actually used to encrypt or decrypt in either implementation. We used the default lengths of each of these parameters in bytes provided by each algorithm's api.h file and initialized the parameters by populating them with unsigned bytes containing arbitrary values.

We then tested the Crypto++ C++ library's implementation of AES. Using a similar structure to our tests for TinyJAMBU and Elephant, we obtained the time of day before encryption and after decryption for 50 loop iterations and divided by the number of loop iterations. We used the default AES key size and block size, both 128 bits.

3.6 Hardware

As previous studies have done, we modeled the resource constraints of an IoT device using a Raspberry Pi [5]. We ran our experiments on a Raspberry Pi 3 Model B+. It has a 1.4GHz 64-bit quad-core processor and 1 GB of memory.

4 Results and Analysis

4.1 Algorithm Performance across Message Size

Figure 3 compares message size vs data processing time for the 5 algorithms: AES, Elephant, TinyJAMBU128, TinyJAMBU192, and TinyJAMBU256. Above all, AES performs the best. However, it is important to take into account the fact that the implementation for AES that was tested was from the Crypto++ library, and therefore was a more optimized version than the

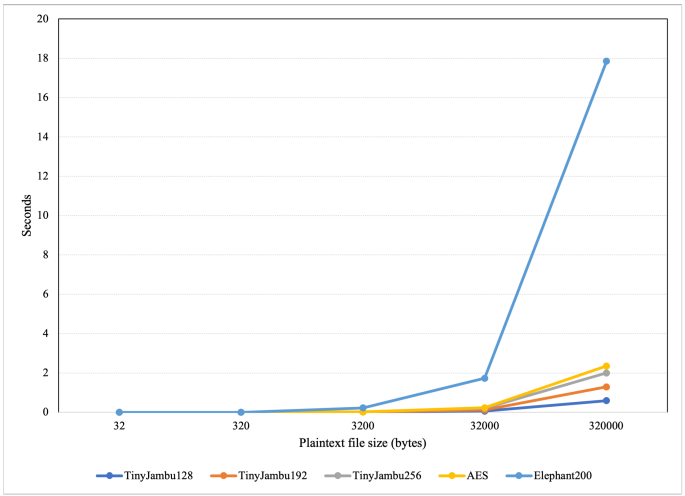

Figure 3: **Encrypt & Decrypt Time (s) Across Message Size (bytes)** *This figure compares the time it takes to encrypt and decrypt a message across message size for the five algorithms. The default gcc optimization level (O0) is used.*

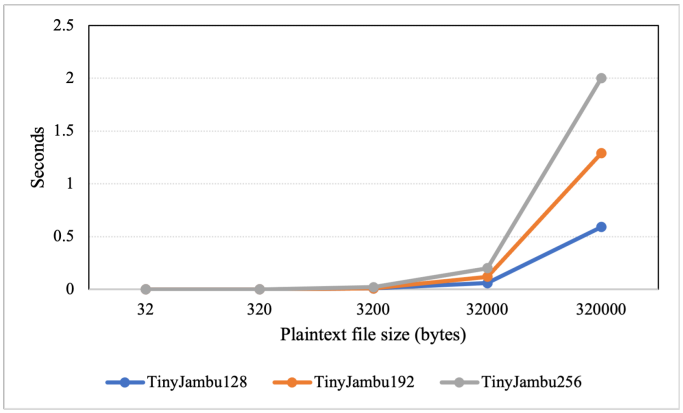

Figure 4: **Encrypt & Decrypt Time (s) of TinyJAMBU Compared with Key Size and Message Size** *This figure shows the encrypt and decrypt time across the sweep of message sizes for the TinyJAMBU algorithms, each having a different key length: 128 bits, 192 bits, and 256 bits.*

implementations of its lightweight competitors which were obtained from the NIST competition.

When comparing only the lightweight algorithms, we find that the TinyJAMBU versions all perform better than Elephant. We speculate this to be the result of several factors.

First, the Elephant code provided only a reference optimization, while TinyJAMBU provided an optimized version. This is likely to be the main discrepancy between the two algorithms, as both are generally similar as a whole and are both lightweight cryptographic algorithms, so such a large difference in encryption and decryption time is not expected. However, Elephant's unique implementation features could have also contributed to these results. Compared to TinyJAMBU's 128-bit state size, Elephant has a larger state size of 200 bits as well as a much larger block size of 200 bits in contrast with TinyJAMBU's 32-bit blocks. Elephant also has a more complex permutation scheme, leading to its higher latency when compared to TinyJAMBU and AES. TinyJAMBU's encryption scheme consists mostly of XORs and bitshifts completed in-function, while Elephant's utilizes masking heavily, calls memset() and memcpy() consistently throughout encrypt functions, and calls a separate function to XOR blocks. These details may add up to raise Elephant's encryption and decryption time.

4.2 TinyJAMBU Performance across Key Size and Message Size

Figure 4 compares the performance of the three TinyJAMBU implementations across message size. The three implementations vary in key size: 128 bits, 192 bits, and 256 bits. With a larger key size, typically security is improved. It is also the case that as key size increased, so did the time it took to encrypt and decrypt the messages. However, the difference in performance between the key sizes only became pronounced at the 320,000 message size, at which point the difference in time between 128 bits and 256 bits was almost 1.5 seconds.

Although TinyJAMBU-128 completes the encryption and decryption the fastest, there are applications where a more secure version such as the 192-bit or 256-bit might be appropriate, such as in cases where the message size is expected to be smaller.

4.3 Algorithm Performance across Compiler Optimization Level: Elephant

Surprisingly, the time needed to complete an encrypt-decrypt cycle increases across all algorithms as the level of compiler optimization increases. Figure 5 compares the time to encrypt and decrypt messages from 32 bytes to 320,000 bytes using the O1 optimizations of each algorithm tested, while Figure 6 compares encryption and decryption time when programs are compiled with the O3 optimization.

The most obvious takeaway from comparing the two graphs is that Elephant gained about a second of latency from this increased compiler optimization. These results are especially interesting considering that the version of Elephant used in our experiments was not originally optimized by its creators. Given this, we expected to see a large decrease in Elephant's encrypt and decrypt time with higher levels of compiler optimization. The increased time suggests that the additional features of GCC's higher optimization levels do not mesh well with the code of Elephant. According to GNU, the -O1 optimization option is meant to "reduce code size and execution time, without performing any optimizations that take a great deal of compilation time"[7], while the -O2 option claims to further boost performance by activating "nearly all supported optimizations that do not involve a space-speed tradeoff". Finally, the -O3 option turns on slightly more optimization flags. Encrypt-decrypt time also increased from the -O1 version to the -O2 version, showing that this shift is not restricted to the -O3.

4.4 Algorithm Performance across Compiler Optimization Level: TinyJAMBU and AES

The trend seen in Elephant is repeated in the other algorithms.

Figure 7 and 8 compare the time to encrypt and decrypt mes-

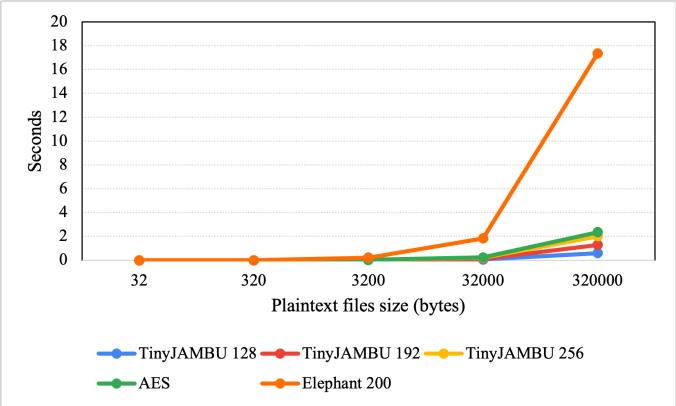

Figure 5: **Encrypt & Decrypt Time (s) for O1 Optimization** *This figure shows the encrypt and decrypt time across the entire sweep of message lengths for all algorithms tested.*

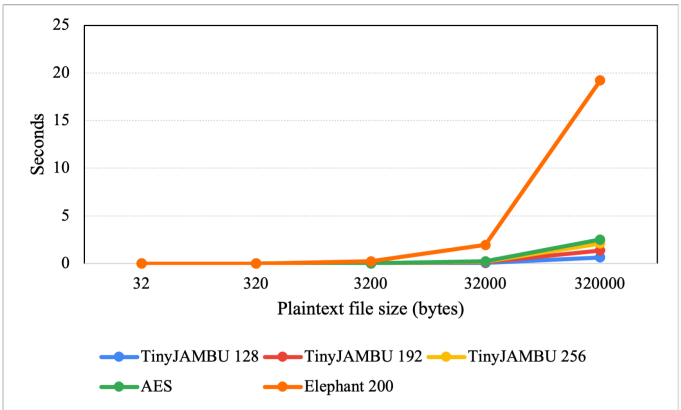

Figure 6: **Encrypt & Decrypt Time (s) for O3 Optimization** *This figure shows the encrypt and decrypt time across the entire sweep of message lengths for all algorithms tested.*

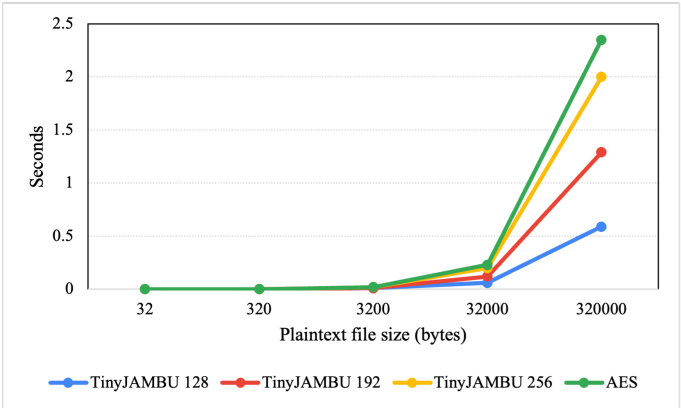

Figure 7: **Encrypt & Decrypt Time (s) for O1 Optimization** *This figure shows the encrypt and decrypt time across the entire sweep of message lengths for the TinyJAMBU algorithms and AES.*

sages from 32 bytes to 320,000 bytes using the TinyJAMBU algorithms and AES. With the removal of Elephant, it can be seen that these algorithms as a group produce similar encryption and decryption times in each optimization. However, like Elephant, the time of an encrypt-decrypt cycle increases with compiiler optimization.

This growth in latency occurs throughout all of the algorithms, which indicates it may be common to the type of symmetric-key, cryptographic algorithms we analyze in this paper. Two factors that could contribute to this finding are a lack of variables for the compiler to optimize and the large amount of data structures that take up cache memory.

Compilers can best simplify code containing variables whose values are guaranteed not to change through the run of a program. Unfortunately, the value of almost every variable in each of the algorithms is constantly changing through XOR operations or masks. Each of the algorithms also contains many data structures, which take up cache memory when accessed regardless of optimizations. Finally, the just under 100 different optimizations contained in -O3 may simply have interacted in a way that decreased performance for the algorithms. Given the difficulty of applying compiler optimizations successfully to these cryptographic algorithms, the algorithms are likely better suited to being optimized by programmers themselves instead of through compilers. Ultimately, however, these algorithms are intended to be implemented in hardware, so focusing on reducing encrypt-decrypt time in software may not be a primary concern for implementers.

4.5 TinyJAMBU Performance across Key Size and Compiler Optimization Level

Figure 9 compares the three TinyJAMBU implementations, the 128-bit, 192-bit, and 256-bit key versions, across four optimization levels offered by the GCC compiler. O0 is the default level, while O3 is the most aggressive optimization level. Evidently, all of the algorithms' performance worsen as level O3. TinyJAMBU-128 and TinyJAMBU-256 perform best at levels O0 through O2 while TinyJAMBU-192 performs best at level O2. The only TinyJAMBU algorithm that benefited from a higher level compiler optimization than the default was TinyJAMBU-192. However, it is important to keep in mind that a program-level optimization effort would very likely have a much greater impact on performance than optimization efforts by the compiler.

5 Conclusion

In this paper, we studied the time it took to encrypt and decrypt a message on an IoT device using AES versus two novel lightweight encryption algorithms: TinyJAMBU and Elephant. Our current findings show that AES has the best performance. However, this ignores the fact that AES has been fine tuned to run as efficiently as possible, meanwhile TinyJAMBU and Elephant are still in their beginning phases of optimization. It is fair to believe that TinyJAMBU and Elephant both have room for improvement in terms of optimization, especially Elephant, which only supported a reference implementation and no optimized implementation.

When looking only at the lightweight algorithms, TinyJAMBU exhibits much faster performance than Elephant. As stated earlier, this could be because of Elephant's lack of an

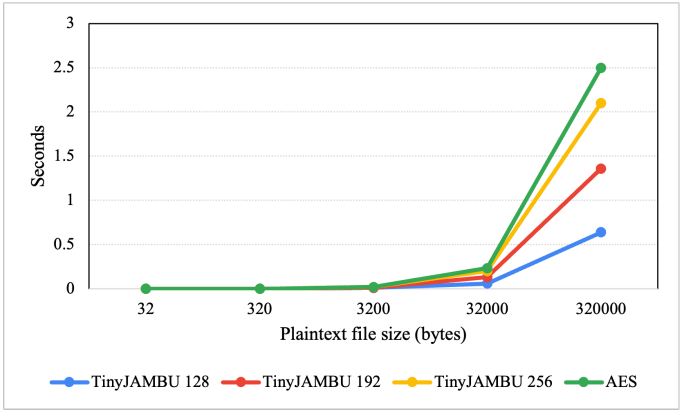

Figure 8: **Encrypt & Decrypt Time (s) for O3 Optimization** *This figure shows the encrypt and decrypt time across the entire sweep of message lengths for the TinyJAMBU algorithms and AES.*

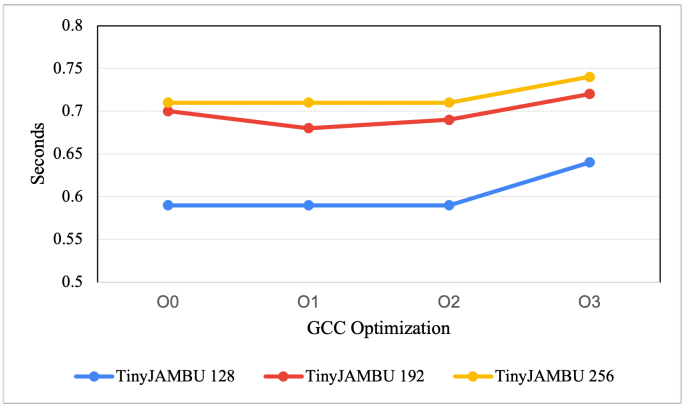

Figure 9: **Encrypt & Decrypt Time (s) of TinyJAMBU Compared with Key Size and Compiler Optimization** *This figure shows the encrypt and decrypt time across the sweep of compiler optimizations for the TinyJAMBU algorithms, each having a different key length: 128 bits, 192 bits, and 256 bits. A message size of 320,000 bytes (320KB) is used.*

optimized implementation; however, this could also be because of an inherent difference in the algorithms. For example, Elephant has a larger state size and block size than TinyJambu. We also find that among the TinyJambu key sizes that are supported, the smallest key size (128-bit) exhibits the fastest performance; However, up until a very large message size (between 32,000 320,000 bytes) the difference between 128-bit and 256-bit remains small. Therefore, if an application desired higher security and did not expect to be sending and receiving messages larger than 32,000 bytes, TinyJAMBU-256 could be an advantageous choice.

Lastly, we conclude that compiler optimizations have a very small impact on the algorithms' performance, and in some cases worsened performance. Therefore, we conclude that to optimize these algorithms, it would need to be done at the programmer level, rather than the compiler level.

5.1 Future Work

Future work could include analyzing the performance of the remaining finalists of the NIST Lightweight Cryptography competition. Hopefully as more work is completed studying the performance of these new algorithms, it will drive others find ways to optimize them and improve their performance to be equal or greater to that of more established algorithms. Additionally, future work can analyze the performance of lightweight cryptography on various hardware configurations to represent different IoT devices. This could reveal that certain algorithms are optimized for particular systems. Lastly, this paper only studied the software implementations of lightweight cryptographic algorithms. Future work could study the hardware implementations as well, which are expected to be more efficient and more secure.

References

[1] Specification for the advanced encryption standard (aes). Technical Report 197, Federal Information Processing Standards, November 2001.

[2] E. Anaya, J. Patel, P. Shah, V. Shah, and Y. Cheng. *A Performance Study on Cryptographic Algorithms for IoT Devices*, page 159–161. Association for Computing Machinery, New York, NY, USA, 2020.

[3] E. Bertino and N. Islam. Botnets and internet of things security. *Computer*, 50(2):76–79, 2017.

[4] T. Beyne, H. L. Chen, C. Dobraunig, and B. Mennink. Elephant v2. Technical report, Radboud University, May 2021.

[5] H. Chan. IoT design framework: Model of end-to-end encryption with protocols (MEEEP). 2021.

[6] F. Dahlqvist, M. Patel, A. Rajko, and J. Shulman. Growing opportunities in the internet of things. `https://www.mckinsey.com/industries/private-equity-and-principal-investors/our-insights/growing-opportunities-in-the-internet-of-things`.

[7] GCC, the GNU Compiler Collection. Options that control optimization. `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[8] E. R. Naru, H. Saini, and M. Sharma. A recent review on lightweight cryptography in iot. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pages 887–890, 2017.

[9] M. S. Turan. Lightweight crypto, heavyweight protection. `https://www.nist.gov/blogs/taking-measure/lightweight-crypto-heavyweight-protection`.

[10] H. Wu and T. Huang. TinyJAMBU: A family of lightweight authenticated encryption algorithms (version 2). Technical report, Nanyang Technological University, May 2021.