Lightweight Cryptography

By Muhammad Shahir Rahman, Sathwik Karnik, Sumiyajav Sarangerel

TABLE OF CONTENTS

1 INTRODUCTION

Low-power devices are everywhere in our daily lives — from healthcare gadgets to digital assistants to home devices. Since these devices tend to have our precious private information, it is crucial to provide a reasonable amount of security given the low power environments these devices work in. On these devices, standard encryption schemes do not work very well for all the applications due to possible limitations in both the hardware (i.e., memory) and software (i.e., processor speed). For example, if there is a video stream or a large stream of data which needs to be secured in a short amount of time, low-power devices will have trouble since they do not have as much computing power as smartphones and laptops. In a low-power environment, tradeoffs are made which leads to a lower security as the power becomes more limited. For instance, smaller key sizes are preferred in such an environment but this may reduce the level of security provided to the system. Thus, the goal of lightweight cryptography is to use less computing power, less memory or less power while providing some sense of security.

From a software point of view, lightweight devices may be bound by memory size, processor speed and latency. In terms of hardware, lightweight devices may be bound by area, throughput and power consumption. While working in these environments, light-weight cryptographic algorithms that are able to provide a decent amount of security in various types of applications are needed. Figure 1 shows examples of devices of various computational ability.

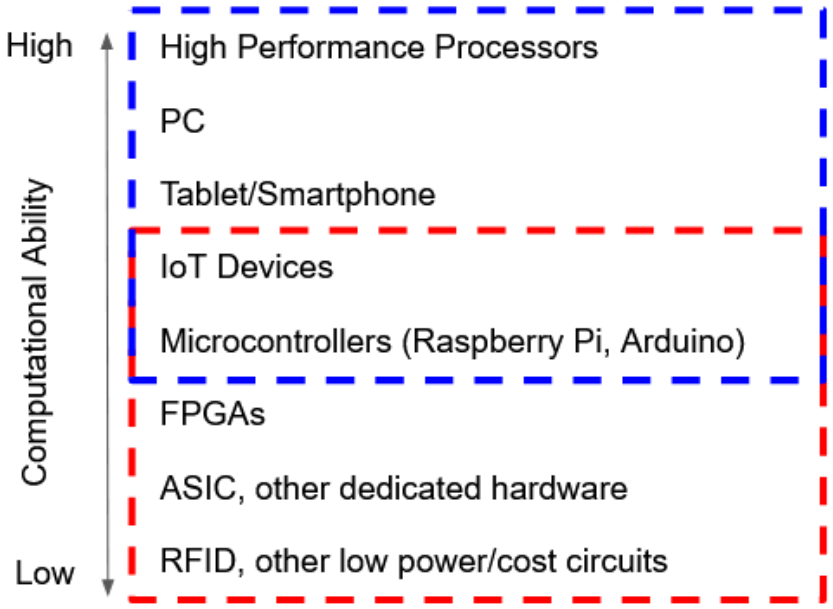

Figure 1: Range of computational abilities over various hardware devices. The red dotted box denotes the "lightweight" devices, while the blue dotted box denotes the non-lightweight devices. [2]

As there are many devices and various different software and hardware constraints across various low-compute devices, there is a need to standardize software and hardware in lightweight cryptography. Noticing this need for standard, the National Institute of Standards and Technology (NIST) — a government agency responsible for measuring scientific advancement and industrial innovation — created a way to benchmark and standardize lightweight cryptographic algorithms that are developed for low-compute environments and started accepting algorithms according to certain requirements [10]. Following this vision, NIST announced that it is accepting algorithms to be considered for lightweight cryptography standards. NIST received 56 algorithms in total and after two rounds, in 2021, ten algorithms (Ascon, Elephant, GIFT-COFB, etc.) were finalized. It is worth noting that the 5th Lightweight Cryptography Workshop is being held virtually from May 9th to May 11st [7].

In a 2015 NIST workshop on lightweight cryptography, Dan Shumow, a researcher in the Microsoft Research Security and Cryptography Group, claimed with some limited evidence that current cryptographic standards and algorithms are sufficient enough for software applications on hardware with the computation power at or above that of a microcontroller [2]. He started his presentation by talking about what devices exactly qualify as low-power and these devices are already powerful enough in terms of software aspects. These devices are possibly limited by their hardware and new standards should apply to hardware only.

In this project, we sought to evaluate his claims by measuring the throughput of various functions (such as hashing, encryption, and decryption) across a variety of standard and lightweight cryptographic algorithms on both a Raspberry Pi and an Arduino Teensy 3.2. After experimenting, we concluded that Arduino and Raspberry Pi do not need lightweight cryptography for many standard applications but would benefit from lightweight cryptography in certain use cases, such as video recording. We then discuss the need for standards in lightweight cryptography, as well as the different areas for optimization in both hardware and software.

2 BACKGROUND

In this section, we consider standard encryption algorithms and hash functions, as well as several candidates for NIST's lightweight cryptography challenge. We will discuss each of these algorithms with descriptions of each algorithm, as well as their use cases.

The table below describes the most commonly used encryption algorithms and hash functions. Encryption algorithms like AES and ChaCha have multiple rounds of operations and provide a very high level of security. ChaCha is more suitable in mobile devices and more CPU friendly. Hash functions are an essential part of randomization in cryptography and the most popular hash functions are SHA-3 and SHA-2. Even though SHA-3 is more secure, it is not widely used due to its slow performance in software [1].

Algorithm	Description
AES	Block cipher encryption algorithm which is a substitution-permutation network and its secure level is highly dependent on its key size and number of rounds
ChaCha	Stream cipher encryption algorithm whose round operations only consist of modular addition, rotation and xor
Speck	Block cipher encryption algorithm which has been optimized for performance in software implementations and works in the same way as ChaCha
SHA-2	Family of hash functions including SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256 and is built using the Merkle-Damgard construction
SHA-3	Hash function which is built by using "sponge construction". Sponge construction is a function which takes a bit stream of any length and outputs a bit stream of any desired length

Table 1: Examples of standard cryptographic algorithms [17, 18, 19]

Even though these algorithms work great in high-power devices (e.g., portable computers, smartphones), there are certain prohibitive constraints in low-power environments, such as memory limitation, timing, power consumption, and gate equivalents (i.e., hardware area). For instance, ChaCha and SHA-2 make extensive use of modular additions, which is not the best choice for lightweight hardware implementations.

As part of the NIST cryptographic competition, there were 10 finalist cryptographic algorithms that NIST recognized. Table 2 briefly describes each algorithm, its purpose, and what software or hardware aspects it seeks to optimize.

Algorithm	Description	Characteristics
Romulus	Tweakable block cipher	High efficiency on short messages, Less area
PHOTON-Beetle	Authenticated encryption and Sponge-based mode hashing	Flexible, High efficiency on short messages
ISAP	Sponge-based authenticated encryption	Robust against side channel attacks
Grain	Stream Cipher	Good throughput, low energy consumption, flexible

Dumbo	Permutation-based authenticated encryption	Has some issues on security, easy to implement, can be parallelized
TinyJAMBU	Block cipher	Less area, can be computed in parallel
Schwaemm	Sponge-based cipher	Small state size, fast and can be boosted with parallelism
Ascon	Block cipher	Good speed, Less area, High efficiency on short messages, Robust
GIFT-COFB	Block cipher	High efficiency on short messages, small state size, fast
Xoodyak	Hashing, Encryption, MAC computation and Authenticated encryption	Robust against side channel attacks, small state size, flexible, secure against multi-target attacks

Table 2: Examples of lightweight cryptographic algorithms. [16]

In this project, we examined both the standard non lightweight and lightweight cryptographic algorithms. We will next discuss how we tested these algorithms across two microcontrollers — a Raspberry Pi and Arduino Teensy.

3 METHODS

3.1 MICROCONTROLLER CHOICE

For our experiments we decided to test standard and lightweight cryptography algorithms on two popular microcontrollers: a Raspberry Pi Zero W and an Arduino Teensy 3.2. We chose these two systems because Arduino and Raspberry Pi's are widely used and the two we selected are at physically small design points, and thus lightweight cryptography may be more applicable there.

The Raspberry Pi Zero W contains a 1 GHz single core ARMv6 CPU (BCM2835), 512MB of RAM 802.11n WiFi and Bluetooth 4.0. Although it is physically very small, it is very much a capable machine [12]. The Arduino Teensy 3.2 features a 72 MHz Cortex-M4 CPU, 64 KB of RAM and 256 KB of flash memory [13]. Although the Teensy is only a bit smaller physically than the Raspberry Pi Zero W, it is not as powerful, and therefore should, in theory, be more suited to utilizing lightweight cryptography algorithms.

3.2 Cryptography Libraries

There were multiple options for various cryptographic libraries we could use. Raspberry Pi's have the capability to run the Python programming language, and thus can utilize the high-level "cryptography" library that is easy and flexible to use. However, since we are considering lightweight it is more appropriate to select optimized implementations written in lower level languages such as C/C++ and Assembly. To this end we found Rhys Weatherly's implementation of cryptographic libraries for the Arduino. This included standard, reduced memory, and lightweight implementations, and a variety of routines included AEAD, hashing, authentication and public key cryptography. For this paper, we concentrated on AEAD and hashing as we could collect more data and compare more implementations in those categories.

However, this library was not adapted for the Raspberry Pi. Since Arduino code is simply C++ with a few additional methods and classes written, we decided to try to adapt the Arduino code to execute on the Raspberry Pi. To this end we found and utilized the piduino C++ library which allows usage of the digital IO on the Raspberry Pi in the same way as the Arduino, and also defines the utility functions for things like measuring elapsed time and printing outputs that the Arduino has defined. Thus using the piduino library we could compile cryptographic tests that utilize the Arduino library on the Raspberry Pi itself, and execute them with no/minimal overhead.

3.3 Metrics Collected

The main metrics we tried to collect are timing results. These include average time to encrypt and decrypt (per byte) and key setup times for the AEAD algorithms. For the hashing algorithms, this includes hashing throughput (per byte). We explored collecting data on other metrics such as temperature, CPU utilization and memory utilization. This data collection was only possible on the Raspberry Pi which runs a Linux environment, so it was possible to run other processes in the background to collect data for these metrics.

However, we quickly found out that these metrics either had data that was not meaningful or was difficult to collect. For example, with temperature, the Raspberry Pi was able to measure the overall CPU temperature up to one decimal point in Celsius. The purpose was to test how much stress a specific cryptographic workload had on the system. However, with successive runs, the temperature would just keep increasing, and eventually reach a peak level when the rate of heat dissipation (through the heat sink) matched the rate of heat generation. Thus, there was no guarantee that the relative increases in temperature between successive runs would be comparable, and the temperature was highly related to the up time of the processor.

For CPU utilization and memory utilization, we realized that in order to properly measure this, we would need to measure CPU utilization and memory usage for every single individual subroutine. For example, it didn't make sense to measure peak memory usage for key setup and encryption combined as they both are different routines with drastically different runtimes and function in most algorithms. Thus to measure this we either needed to create dozens of separate

executables and profile each individual, or use an actual profiler toolset (perf, valgrind, etc.). This was not necessary given the memory usage in the Raspberry Pi was far lower than the capacity and a few runs indicated that CPU utilization was mostly the same. We assume this is because the algorithms were mostly compute bottlenecked as opposed to memory bottlenecked or bottlenecked by inability to parallelize well on small devices.

Finally, we also tried to look at various compilation modes. As discussed previously, lightweight applications have many requirements, including reducing the program size due to limited storage, and also increasing the program speed due to limited compute. By default, the compiler for the Teensy 3.1/3.2 utilizes the "-Os" compilation flag to optimize for reducing program size. We attempted to optimize for program performance with "-O3", but we found little to no differences and results were almost exactly the same. Weatherly's cryptographic algorithm implementations were already written in a very optimized manner, and some included assembly code, and different optimization schemes did not make significant differences.

Please see the end of the paper for a link to our Github code.

4 RESULTS

In this section, we illustrate the results of our experiments on the Raspberry Pi and Arduino Teensy.

First, we considered the various different authenticated encryption with associated data (AEAD) schemes, including both standard non-lightweight and lightweight cryptography algorithms. These non-lightweight schemes include AES-128-ECB, ChaChaPoly, GCM AES-128, GCM-Speck-256, and GCM-SpeckTiny-256. The lightweight schemes include Acorn-128 and ASCON-128. Figure 2 shows the operation throughputs (operations/second) for key setup across each of these schemes, while Figure 3 shows the average throughput of encryption and decryption. As expected, the Raspberry Pi performs these operations at a faster rate than the Arduino Teensy. While Acorn-128 and ASCON-128 are regarded as lightweight algorithms, only the Acorn-128 has a significantly higher throughput in encryption and decryption. As we will discuss later, these algorithms are among other lightweight cryptographic algorithms that aim to optimize different design considerations, one of which is execution throughput.

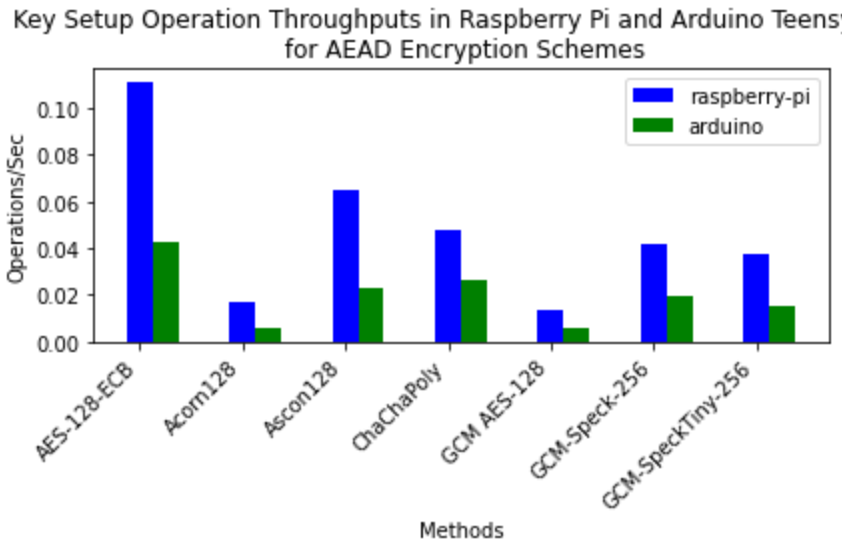

Figure 2: Key Setup Times in Raspberry Pi and Arduino Teensy for AEAD

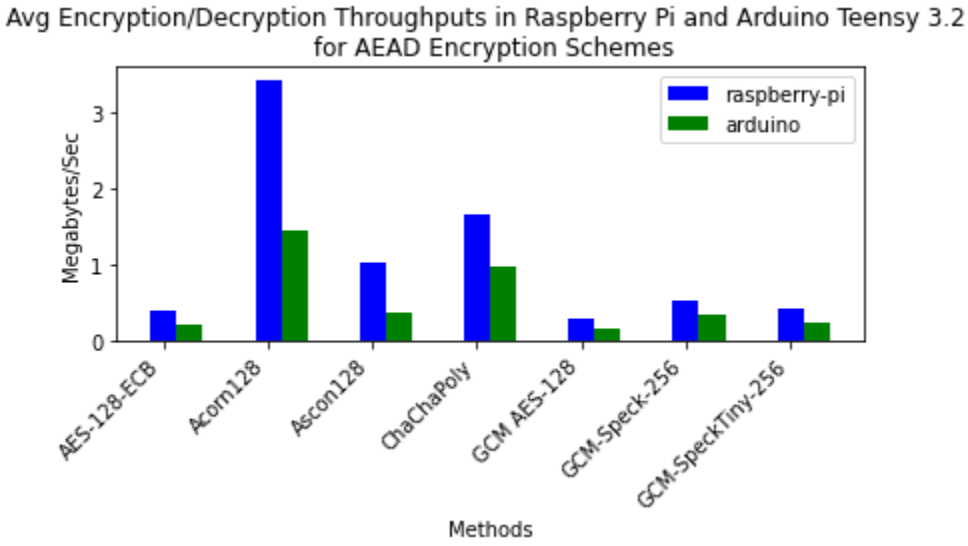

Figure 3: Average Encryption-Decryption Throughputs (in Megabytes/Sec) in Raspberry Pi and Arduino Teensy for AEAD

Next, we considered several hashing functions, of which SHA256 is a non-lightweight hashing function, while the rest are among the lightweight hashing function submissions to the NIST lightweight cryptography challenge. Figure 4 shows these results across both the Raspberry Pi and Arduino Teensy. Notice that SHA256 has a higher throughput than the lightweight hashing functions that were considered. Once again, we observe that the Raspberry Pi has a faster throughput than the Arduino Teensy in the hashing functions.

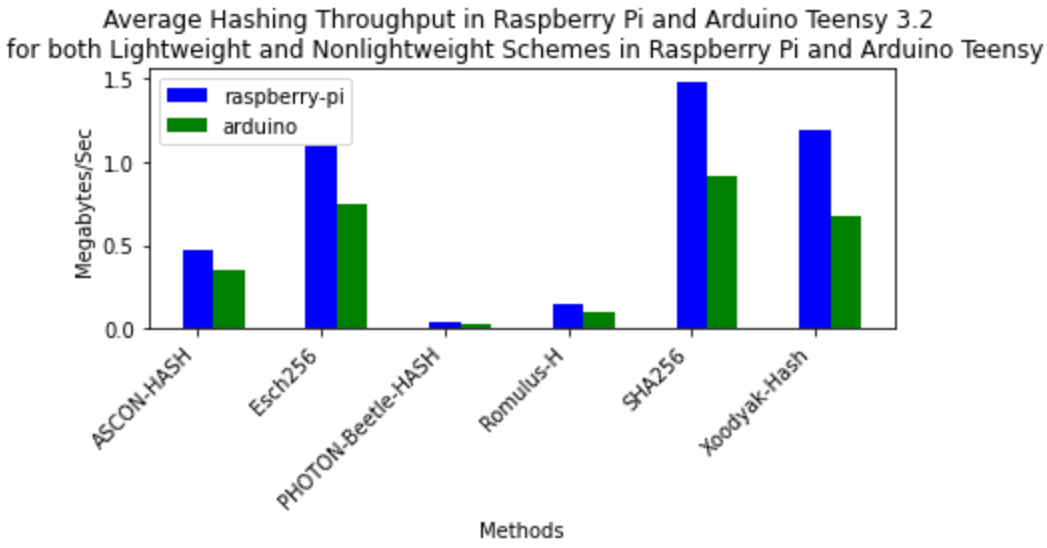

Figure 4: Average Hashing Throughputs (in Megabytes/Sec) for both Lightweight and Nonlightweight Schemes in Raspberry Pi and Arduino Teensy

Lastly, we considered several lightweight cryptography algorithms, most of which are submissions to the NIST lightweight cryptography challenge. Figure 5 shows the ratio between the throughputs of encryption between the encryption of 128-byte packets and the encryption of 16-byte packets. As we expect, the encryption of 128-byte packets has a higher throughput than the encryption of 16-byte packets for each of these lightweight cryptography algorithms. Figure 6 shows the average throughput of encryption and decryption of 128-byte packets on the Raspberry Pi and Arduino Teensy. As we discuss next, these throughputs for both devices are on the order of ~1 MB/sec, which does not pose a significant bottleneck for common usage of the Raspberry Pi and Arduino Teensy.

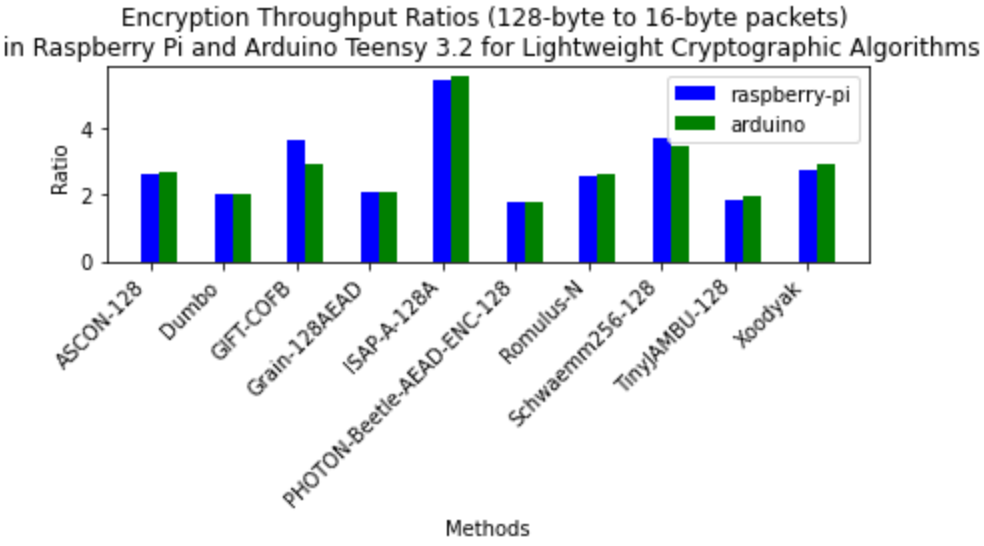

Figure 5: Encryption Throughput Ratios in Raspberry Pi and Arduino Teensy for NIST Lightweight Cryptography Algorithms

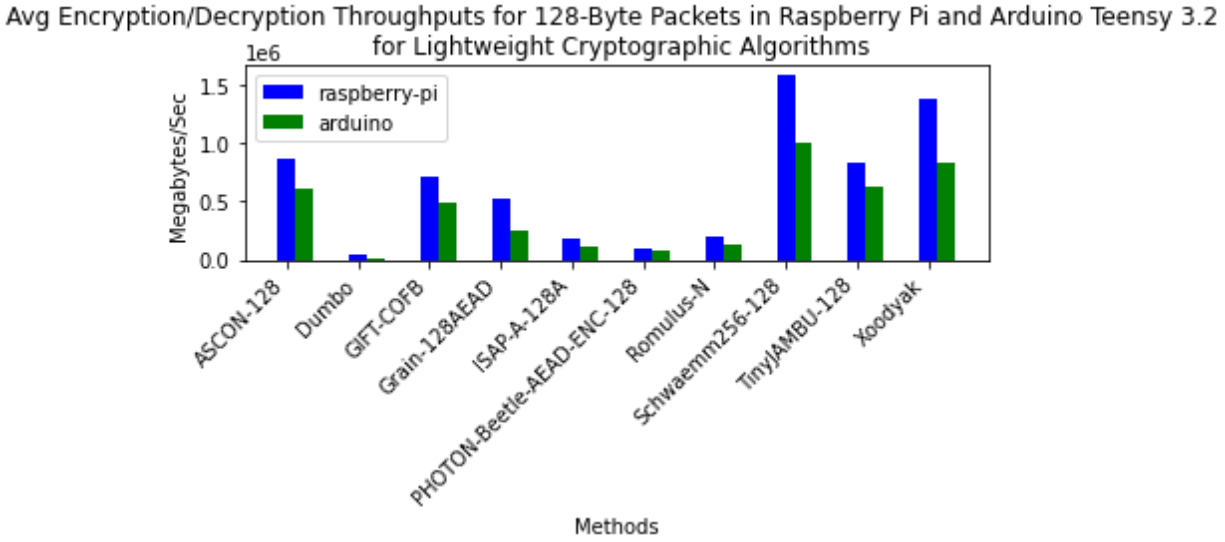

Figure 6: Average Encryption-Decryption Throughputs (in Megabytes/Sec) for 128-Byte packets in Raspberry Pi and Arduino Teensy for NIST Lightweight Cryptography Algorithms

5 Discussion

In the previous section, we showed the results of experiments across both the Raspberry Pi and Arduino Teensy. As anticipated, in all cases, the Raspberry Pi actually does much better than the Arduino Teensy in the throughput and operations per second. But in both cases, neither the Raspberry Pi and Arduino Teensy always need the use of lightweight cryptography algorithms. Moreover, these algorithms were shown to be able to handle the throughput of non-lightweight

cryptography algorithms, as seen in the plots. However, in some specific use cases that may need to handle high throughput, such as video recording on an Arduino, video data would be streamed at 1-6 MBps [15], while AES encryption on the Arduino can take 0.2 MBps, which can be a bottleneck. On the other hand, SCHWAEMM has a throughput of 0.98 MBps, which is less of a bottleneck.

As we now consider the need for lightweight cryptography, we seek to understand how these methods are standardized and optimized for different needs.

As researchers in the field seek to assess each cryptography algorithm, there is a need to standardize the algorithms using various benchmarks. Many of the algorithms claimed to be lightweight, such as Grain-128AEAD, which claimed in their paper to be 3.8 times faster than AES-GCM [11]. However, when collecting results on Weatherly's implementation, we found that many algorithms, including Grain, were actually slower than AES-GCM. These results were surprising for us and we sought to understand why this was the case.

One important factor we found is that the results are highly implementation-dependent. When reading some of the finalist papers from the NIST lightweight cryptography challenge, it seems that many of them utilize cryptography implementations from FELICS (Fair Evaluation of Lightweight Cryptographic Systems) [8]. However, Weatherly's algorithm for AES performs significantly faster than the FELICS version of AES. Based on data collected by NIST, the FELICS implementation is ~2.5 smaller in program size compared to Weatherly's implementation, but also ~4.0 times slower than Weatherly's version. Thus when lightweight cryptographic algorithms are compared to the optimized AES in Weatherly's code, it turns out to actually be slower, while still being faster than the FELICS version [6].

	Program Size (Bytes)	Cycles to Encrypt 1 Byte
Implementation A (felics-v2)	3272	16824
Implementation B (rweather)	8096	4156

Table 3: Implementation statistics for AES128-GCM [6]

Furthermore, we find that not all of the lightweight cryptography algorithms are optimal on all lightweight compute systems, and were not necessarily designed with optimizing the same metrics. We researched further and found data collected regarding the performance of lightweight cryptographic algorithms on the NIST challenge on FPGAs. For example, in a plot from the paper below, we see a clear relationship between average throughput and energy per bit for various algorithms submitted to the NIST competition. While ASCON has the highest throughput on the FPGA, SCHWAEMM has the highest throughput on the Arduino Teensy, and the performances differ by large factors. Meanwhile the Xoodyak algorithm performs well on the Teensy and the FPGA, but is optimal on neither system [14].

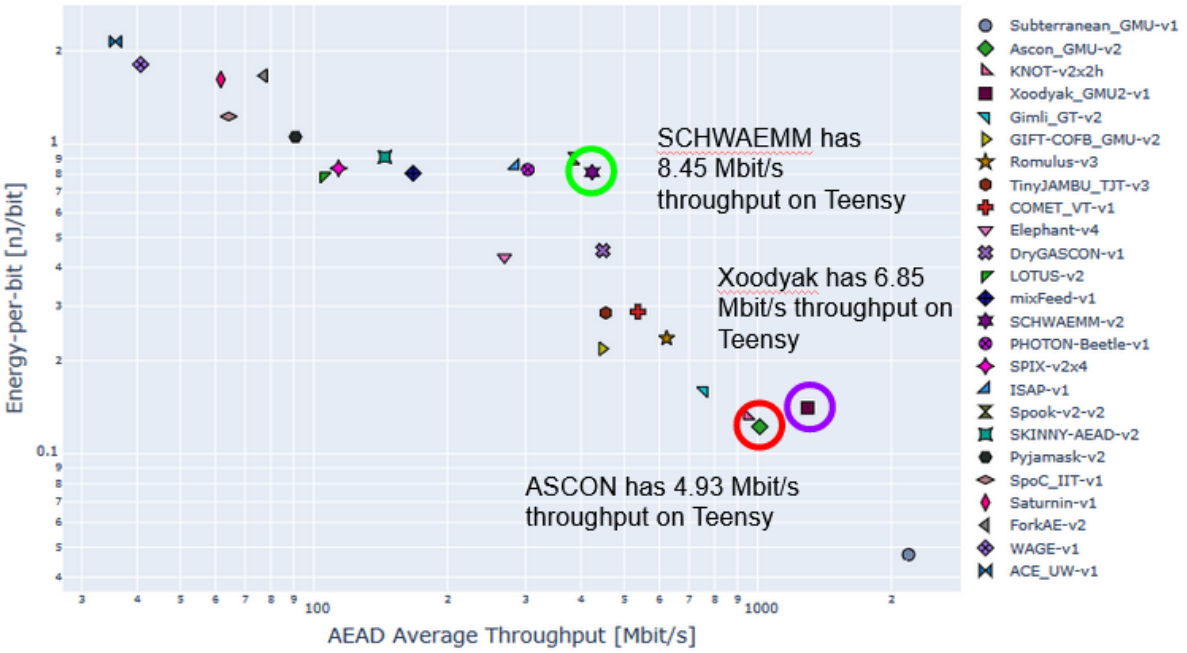

Figure 7: Energy per bit vs Average Throughput for Llightweight algorithms on an FPGA [14]

From our analyses we see that lightweight cryptography is highly necessary for future applications. This shows that when creating lightweight cryptography standards, the compute architecture needs to be taken into consideration, as well as which factors an algorithm standard is trying to optimize (such as program size, throughput, area and energy). While some algorithms like Xoodyak appear to work well on multiple systems, if the focus is maximizing throughput or any other metric, different standards will likely be needed for different architectures. Since performance changes drastically with different implementations based on various optimization choices, this may require multiple standards as well (to account for each optimal choice). Both these reasons indicate that standardization may not be a good idea.

Many algorithms have been proposed that are lightweight in the sense of utilizing low memory, however they are very slow in implementation, and this indicates that some level of standardization is needed in order to ensure applications in the future have a minimum level of performance on lightweight devices.

Our conclusion is that lightweight cryptography should be standardized but not as a hard rule but more as a recommendation for a good algorithm that is performant on most architectures and has a decently low program size, although it may not be optimal for most cases. However, if performance is a strong bottleneck or high security is needed for example, developers/engineers should be encouraged to select their own algorithm, and not necessarily stick to a standard.

6 Future Directions

In the future, this project could be extended to measure both the CPU utilization and memory utilization, as other papers do not seem to measure these quantities. As noted, in our current software setup, this would have required several hundreds of executables to run each of the individual cryptography tests, thus requiring significantly more time for data collection. However, with some modifications to the software, this may be possible to collect in the future. Additionally, beyond the scope of this project, we could have tested on different hardware, including FPGAs and ASICs. Of course, for these hardware circuits, we would need to program each of these methods in some hardware description language. As previously noted, these hardware circuits provide some value for faster processing but lower memory availability, and can, thus, serve as another hardware baseline for comparison of different lightweight cryptographic algorithms.

7 Acknowledgements

We would like to thank the 6.857 professors Ron Rivest, Yael Kalai and the course staff, especially Kyle Hogan and Andres Fabrega, for their guidance and suggestions on this project.

8 Implementation

For greater detail on the exact code that was written and executed, please visit https://github.com/man2machine/6857-final-project.

9 References

[1] Turan, Meltem Sonmez. “Applications and Standardization of Lightweight Cryptography,” 1901, 108.

[2] Shumow, Dan. “Thanks, But No Thanks Current Cryptographic Standards Are Sufficient for Software,” n.d., 14.

[3] Weatherly, Rhys. *Arduino Cryptography Library*. C++, 2022. https://github.com/rweather/arduinolibs.

[4] Weatherly, Rhys. *LWC Finalists*. C, 2022. https://github.com/rweather/lwc-finalists.

[5] Liu, Leibo, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. “A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications.” *ACM Computing Surveys* 52, no. 6 (November 30, 2020): 1–39. https://doi.org/10.1145/3357375.

[6] *Benchmarking of Lightweight Cryptographic Algorithms on Microcontrollers*. C. 2020. Reprint, National Institute of Standards and Technology, 2022. https://github.com/usnistgov/Lightweight-Cryptography-Benchmarking.

[7] Computer Security Division, Information Technology Laboratory. "Lightweight Cryptography | CSRC | CSRC." CSRC | NIST, January 3, 2017. https://csrc.nist.gov/Projects/Lightweight-Cryptography.

[8] Dinu, Daniel, Alex Biryukov, Johann Großschädl, Dmitry Khovratovich, Yann Le Corre, and Léo Perrin. "FELICS - Fair Evaluation of Lightweight Cryptographic Systems," n.d., 14.

[9] Grimes, Roger A. "Why Aren't We Using SHA3?" CSO Online, February 21, 2018. https://www.csoonline.com/article/3256088/why-arent-we-using-sha3.html.

[10] Hernandez, Paul. paul.hernandez@nist.gov. "NIST General Information." Text. NIST, December 24, 2008. https://www.nist.gov/director/pao/nist-general-information.

[11] Maximov, Alexander and Martin Hell. "Software Evaluation of Grain-128AEAD for Embedded Platforms." IACR Cryptol. ePrint Arch. 2020 (2020): 659. https://eprint.iacr.org/2020/659.pdf

[12] Prakash, A., et al. (2020, October 30). Raspberry Pi Zero W Specs, features and more. It's FOSS. Retrieved May 10, 2022, from https://itsfoss.com/raspberry-pi-zero-w/

[13] "Teensy® 3.2 Development Board." PJRC, https://www.pjrc.com/store/teensy32.html.

[14] Mohajerani, Kamyar et al. "FPGA Benchmarking of Round 2 Candidates in the NIST Lightweight Cryptography Standardization Process: Methodology, Metrics, Tools, and Results." IACR Cryptol. ePrint Arch. 2020 (2020): 1207. https://eprint.iacr.org/2020/659.pdf.

[15] "Bitrate | Video Streaming Definition." Haivision, www.haivision.com/resources/streaming-video-definitions/bitrate/.

[16] Rezvani, Behnaz and William Diehl. "Hardware Implementations of NIST Lightweight Cryptographic Candidates: A First Look." IACR Cryptol. ePrint Arch. 2019 (2019): 824. https://eprint.iacr.org/2019/824.pdf

[17] Pfau, Johannes et al. "A Hardware Perspective on the ChaCha Ciphers: Scalable Chacha8/12/20 Implementations Ranging from 476 Slices to Bitrates of 175 Gbit/s." 2019 32nd IEEE International System-on-Chip Conference (SOCC) (2019): 294-299. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9088102&tag=1Beaulieu, Ray et al. "SIMON and SPECK: Block Ciphers for the Internet of Things." IACR Cryptol.

ePrint Arch. 2015 (2015): 585. https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/papers/session1-shors-paper.pdf

[18] Handschuh, Helena. “SHA Family (Secure Hash Algorithm).” Encyclopedia of Cryptography and Security (2005). https://link.springer.com/referenceworkentry/10.1007/0-387-23483-7_388