12-17-2019

Analysis of the Duration and Energy Consumption of AES Algorithms on a Contiki-based IoT Device

Brandon Tsao

Santa Clara University

Department of Computer Engineering

Date: December 17, 2019

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Brandon Tsao

ENTITLED

Analysis of the Duration and Energy Consumption of AES Algorithms on a Contiki-based IoT Device

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTERS OF SCIENCE IN COMPUTER ENGINEERING

B. Dezfouli

Thesis Advisor
Dr. Behnam Dezfouli

Thesis Author
Brandon Tsao

Chairman of Department
Dr. Silvia Figueira

yuhong liu

Thesis Reader
Dr. Yuhong Liu

Analysis of the Duration and Energy Consumption of AES Algorithms on a Contiki-based IoT Device

By

Brandon Tsao

Dissertation

Submitted in Partial Fulfillment of the Requirements
for the Degree of Masters of Science
in Computer Engineering
in the School of Engineering at
Santa Clara University, 2019
Santa Clara, California

Robots do not complain, question, or rest, which makes them good role models for the rest of you.

- Vadim Kozlov, ficitonal character from Sid Meir's *Civilization: Beyond Earth*

Acknowledgments

First, I would like to extend an immeasurable thanks to my research advisor for his immense amount of patience helping me through this work. I would also like to thank my friends and family for encouraging me through my graduate studies journey.

Analysis of the Duration and Energy Consumption of AES Algorithms on a Contiki-based IoT Device

Brandon Tsao

Department of Computer Engineering
Santa Clara University
Santa Clara, California
2019

ABSTRACT

With the growing prevalence of the Internet of Things, securing the sheer abundance of devices is critical. The current IoT and security landscapes lack empirical metrics on encryption algorithm implementations that are optimized for constrained devices, such as encryption/decryption duration and energy consumption. In this paper, we achieve two things. First, we survey for optimized implementations of symmetric encryption algorithms. Seconds, we study the performance of various symmetric encryption algorithms on a Contiki-based IoT device. This paper provides encryption and decryption durations and energy consumption results on three implementations of AES: TinyAES, B-Con's AES, and Contiki's own built-in AES. In our experiments, we found the algorithms specifically built for constrained devices used about 0.16 the energy and time to perform encryption and decryption when compared to algorithm implementation that weren't optimized for constrained devices.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet of Things (IoT) is taking the world by storm, with solutions ranging from consumer convenience to life-critical systems. By the year 2020, there will be over 20 billion digital devices [1], which also means 20 billion distinct points of vulnerability. For example, implanted and networked medical devices such as pacemakers are already common [2]. While the Internet connectivity will allow streamlined patches from retailers, it also paves a path for potential hackers, as with the case of St. Jude Medical's Quadra Allure MP [3], where half a million pacemakers were recalled for a vulnerability that could grant unauthorized access. The consequences could be as major as the loss of personally identifiable medical data and as critical as the actual loss of life. Even seemingly innocuous devices such as DVR boxes could be leveraged in large-scale attacks, such as when a Mirai IoT Botnet [4] orchestrated one of the most prominent DDoS attacks in modern computing history. Self-driving cars could be driven off roads. Water flow sensors could be silenced, preventing the alerting of a flood, or worse, fed false data to actuate cybermechanical systems to actually start a flood [5]. The list goes on.

Unfortunately, while securing all these devices is certainly a good idea, traditional means of security does not scale well to the IoT landscape. The small physical size of many IoT devices, also known as motes, forces the use of smaller processors with fewer registers and less cache memory. Additionally, IoT devices usually prefer batteries over a power grid as a source of electrical power. While wired solutions exist, battery power usually allows for faster, cheaper, and more flexible installations in many cases. Most IoT

devices are constrained in terms of both processing power and energy consumption. Unfortunately, this makes it difficult to implement secure cryptographic algorithms and related operations that are known for their heavy footprint in both of these resources.

Regardless, a compromise must be reached the secure IoT devices. There has been much work trying to create security algorithms and protocols that specifically address the limitations of IoT [6]. Regardless of their viability, however, new encryptions algorithms have not been trialed as much, and present a larger risk of containing undiscovered bugs. On the other hand, traditional algorithms have been tried and proven to be robust in most circumstances. However, IoT does not fall under the purview of "most circumstances"; IoT devices cannot be treated as normal, classical computing devices.

Unfortunately, there is little empirical data on how well traditional algorithms actually perform on IoT devices, or how modified algorithms hold up. Will traditional algorithms take too long? Or will they use too much power, reducing a mote's lifetime to a mere hour? Will modified algorithm implementations prove to be just as secure? They might have less clock cycles, but do they actually use less power? While doing a survey, there were few papers found that provided actual data on the performance of traditional algorithm implementations and those optimized for constrained devices.

To fill the gap in the current space, we obtain various metrics regarding power usage and encryption/decryption duration for various algorithmic implementations of Rijndael AES [7], the eponymous Advanced Encryption Standard. We conduct experiments using multiple traditional and modified algorithm implementations on a constrained device, specifically the Texas Instruments CC2650 [8]. The CC2650 is a highly power-efficient System on a Chip (SoC) including an ARM Cortex-M3 processor and wireless connectivity transceivers. This device runs Contiki, which is a lightweight, event-based operating system. We quantitatively determine the extent to which optimized algorithms yield better results in both of these aspects. Additionally, we investigate why they fared bet-

ter, by examining the source code of the algorithmic implementations. We found what one might expect: that optimized algorithmic implementations performed far better than the ones that were not optimized or were optimized for higher power devices. In the experiments, multiple input payload sizes are used and all AES implementations scaled linearly with these inputs, which was also expected.

The rest of this thesis is organized as follows: Related work is examined in Chapter 2. We briefly give an overview of the Rijndael AES algorithm in Chapter 3. Experimental setup and methodology is explained in Chapter 4. Chapter 5 discusses results and discussions. We conclude the thesis in Chapter 6.

Chapter 2

Related Work

Regarding the survey portion of this thesis, we first survey other papers regarding the state of security regarding IoT in addition to the specific security problems that IoT faces. Second, explore attempted optimizations of IoT specific security solutions, especially ones that reduce required processing power or energy. Third, find and consolidate hard data regarding these optimizations, focusing on symmetric encryption algorithms that run on edge devices.

Kai Zhao and Lina Ge [9] performed a high level analysis in 2013, suggesting that IoT not only does has to deal with the same security issues as other information domains, but also faces other unique issues such as privacy protection and heterogeneous network authentication and authorization. The paper claims that IoT solutions will continue to grow into larger networks, making the task of securing the network much more difficult. Most IoT solutions aim for (i) comprehensive perception, (ii) reliable transmission, and (iii) intelligent processing.

(i) Comprehensive perception is a measurement of an edge device's ability to consistently, accurately, and reliably obtain information about an object, e.g. successfully taking a temperature reading from a room every hour. (ii) Reliable transmission is a measurement of how many transmissions make it from their source to their destination, e.g. sending the temperature readings from an edge device to a base station, and the base station forwarding the information to a cloud server. (iii) Intelligent Processing is a measurement is the actual computation of data (and where it is processed), e.g.

taking an average of the temperature readings in the cloud.

Availability comes into play with both (i) comprehensive perception and (ii) reliable transmission, as the prevalent nature and generally unsecured physical location of edge motes makes it relatively easy to disrupt. Unlike a cloud server which is locked behind a metal cabinet in a patrolled data center, edge motes often exist in easily-accessible public spaces. Denial of a mote would then be as easy as physically destroying the mote. Confidentiality also poses a large issue. If nodes aren't encrypted, physical access would immediately lead to the leakage of sensitive data, such as communications keys and destination IPs. Physical access also opens up a variety of side channel attacks that could defeat encryption, such as Differential Power Analysis, which can speed up the time it takes to crack an encryption key.

(ii) Reliable transmission and (iii) intelligent processing are more closely related to traditional security issues of the network layer and application layer. As such, they do not need to be addressed specifically in regards to IoT security and will not be discussed here.

Canteaut et al. [10] suggest multiple encryption algorithms that are suitable for IoT. Stream ciphers are especially good candidates, as they do not require padding and decryption can begin as soon as any amount of data is received. Creating additional padding, as would be done in a block cipher, takes additional processing power. The constant but potentially erratic transmission nature of edge motes encourages schemes which can encrypt/decrypt without a full discrete unit of data. Unfortunately, industry standards tend to prefer block ciphers such as AES. Some of these can be considered block-based stream ciphers, e.g. AES in CTR mode. However, these modes still fall behind in throughput and latency compared to dedicated stream ciphers. Therefore, dedicated stream ciphers are still preferred in IoT applications. Specifically, the authors strongly recommend the following stream ciphers: Grain v1, HC-128, MICKEY, Rabbit,

Salsa20/12, Sosemanuk, and Trivium. In our thesis, we want to provide hard data with a popular block cipher in a non-streaming mode. This will provide necessary information when deciding between the trade-offs of a stream cipher vs a block cipher.

Hui Suo et al. [11] performed a similar review. Their literature highlights the importance of key management and node authentication. Security measures such as frequency hopping communication and public-key encryption are less practical with the limited electric power, processing power, and storage capacity of constrained devices. The lack of asymmetric cryptography further highlights the difficulty in setting up secure keys among a considerably large number of motes. Assuming that keys can be set up, by-hop encryption poses certain dangers as each node might have a plaintext version of the data to be transmitted. This danger is increased in the IoT landscape, as nodes are more susceptible to physical capture. However, the need in fully encrypting environmental sensor data (e.g. weather temperature readings) is less important, as it would be easier for an attacker to simply place their own sensor in the same area, assuming the environment is public. In this scenario, it would be more important to ensure the integrity and authenticity of the sensor data, i.e. guaranteeing the sensor data is accurate and not tampered with. Privacy and confidentiality come more into play in regards to medical sensor data, such as human temperature readings, in which leaked data could result in legal issues. The literature also briefly mentions the lack of regulatory legislature regarding IoT due to its fairly young existence.

Xinlei Wang et al. [12] propose a solution for the issue of key management: Attribute Based Encryption (ABE). Traditional encryption schemes usually require each node to know not only the identity of the other nodes in the cluster, but also to have credentials that would allow secure communications, i.e. both nodes having a public key to allow the sharing of a symmetric key. In ABE, nodes do not need to pre-share secrets, which simplifies key management, especially in an IoT landscape. However, this literature

focuses on ABE for mobile devices. While mobile devices can certainly play a critical role in IoT, especially as access points for smaller devices, they usually are not as restricted as actual edge motes. As ABE is still constrained by the complex mathematics as all public key cryptographic schemes, it may not be the best solution for smaller motes. The paper provides hard data on execution time, data / network overhead, energy consumption, and CPU usage to allow for better informed decisions when considering the trade-offs of this scheme.

These experiments utilized a laptop with a 64-bit 1.60 GHz processor and 4 GB of RAM and a smartphone running Android 4.04 with a 32-bit, 1.6 GHz processor and 1GB of RAM. On both devies, ABE took longer and utilized more CPU and memory than RSA, the more traditional algorithm that was used for comparison. In the literature's conclusion, "without significant improvement, the classic ABE algorithms are best used when the computing device has relatively high computing power and the applications demand low to medium security." However, the literature also points out that ABE allows for better flexibility regarding access control, which would be a key factor in an IoT landscape.

From our survey of the state of security of IoT, we found the main issues barring traditional security measures were scalability (in the case of key management and mass node authentication) and resource usage. If traditional symmetric encryption algorithms were optimized for IoT devices and used less resources, they could be better used in conjunction with novel key management schemes. In the case that traditional key management schemes and traditional symmetric encryption algorithms are used in an IoT problem space, the same algorithms with optimized implementations should still fare better. The next part of the survey is to gather the work done in this area and see if optimized algorithmic implementations provide positive benefits.

Salajegheh et al. [13] explore various software methods of reducing energy consumption.

The authors mainly achieved this by avoiding as much local context switching as possible, especially the declaration of local variables within local functions. This limits the amount of context switching, and therefore the number pushes and pops a program has to carry out. These are expensive operations, and reducing them as much as possible will reduce energy. In the realm of encryption algorithms, however, these techniques might not be suitable for immediate IoT implementation for various reasons: (i) altering function scope can affect how the algorithm leverages memory, possibly opening more vectors for side channel attacks; (ii) memory changes can also affect how addresses are assigned, possibly increasing the chances of another function accessing sensitive variable information; and (iii) programs must limit local variables and functions, forcing redundant code which will increase the size of the binary. However, these are not necessarily negative consequences, but need to be considered when weighing different trade-offs.

Additionally, we must consider the use cases and context of such optimizations. Often times, changing function scope would not only require the modification of the core operating systems of the constrained device to implement various encryption algorithms by default, but would also require those functions to be part of the actual operating system. Calling these "pseudofunctions" could pose an even greater difficulty as many libraries requiring encryption would then have to invoke the operating system itself to perform encryption and decryption. Not only does this pose a security risk by giving external function access to OS level variables, but it also requires a costly context switch, which most likely undoes the savings gained by globalizing variables.

One of NEC's technical journals [6] also proposes a new algorithm altogether, specifically designed for constrained devices. NEC's own TWINE algorithm [14] can achieve a lower power footprint than that of AES by using customized hardware, which uses 2k gates compared to the 15k gates of a similar AES hardware core implementation. Of course, this requires said specialized hardware. AES still outperforms TWINE in pure software

implementations, though the executable of TWINE can achieve sizes as low as 500 bytes. This algorithm, while tested by NEC itself and others, is still less vetted than classic algorithms such as AES, and is not yet as popular in terms of encryption standards.

Another way of reducing power while still using traditional algorithms is to create dedicated hardware to perform said algorithms. By performing the necessary arithmetic operations of an encryption algorithm directly using gates allows the bypass of various layers of software abstraction (and their accompanied hardware), therefore reducing the overall power. Hardware also gives the advantage of generally being more resistant to side channel attacks, as they are usually less transparent and more difficult to directly interact with without physical access, usually acting like black boxes.

While the arithmetic operations can simply be carried out by normal arithmetic gates in serial, it makes sense to optimize the hardware to achieve as much as possible in a single clock cycle. There are different approaches to this with varying levels of trade-offs in terms of power usage, chip size, flexibility and scalability.

Hamalainen et al. [15] have created chips specifically designed to minimize the number of gates, therefore reducing overall chip size and power consumption. They managed to reduce the number of gates to about 3,100. With a 153 MHz clock, the chip is able to achieve a 121 Mbps throughput of encryption via AES 128. Using 30 to 62 $\mu W/Mhz$, this is about 4.8 to 9.9 nJ per block encrypted/decrypted.

S. Mangard et al. [16] created a chip architecture that focused on flexibility and scalability, having a total gate equivalent of 10,800. Operating with a 64 Mhz clock, the standard chip achieved throughput of 128 Mbps, with a high-performant variation using about 15,500 gates running a throughput of 241 Mbps. This paper unfortunately did not include power measurements, but we can assume the larger gate footprint would result in larger power usage. Looking at the quantified results, S. Mangard et al.'s architecture

outperforms Hamalainen et el but uses more gates, and therefore more power.

However, such solutions would require hardware additions to each constrained device, as well as firmware/software to interface with it for encryption/decryption, which would likely incur an upfront redesign cost. This needs to be considered a trade off, as this initial cost could still reduce costs in the long-run if lower power consumption leads to less maintenance or longer field life.

The thesis of A.A.A.Y Hassan et al. [17] outlines some empirical data collected on various symmetric key algorithms, such as AES and Blowfish [18]. We will expand on some of these findings later in this thesis. While the paper was oriented toward the IoT landscape, the tests were done on a laptop with a 64-bit Intel i5 processor running Windows 10. The thesis provides metrics on encryption duration, space complexity, and throughput. For a message inputs from 62 bytes to 223 bytes, AES and Blowfish performed similarly in terms of time and throughput, with Blowfish outperforming AES after 283 bytes, taking 0.80 and 1.24 times the time and throughput, respectively. However, there was only a second data point after 223 bytes so it's unclear whether this improvement continued or what the rate of further improvement was. It is also unclear if these improvements would translate to a more constrained device. Nonetheless, the paper recommends Blowfish for IoT given its reduced encryption time and increased throughput over AES.

A similar paper by Michael Healy et al. [19] examined hardware and software implementations of three symmetric block-cipher encryption algorithms: Rijndael AES, RC5, and Skipjack. Of these algorithms, AES remains to be the most secure. RC5 is vulnerable to differential attacks, in which an attacker would be able to obtain the encryption key provided they are able to encrypt a number of chosen plaintexts. However, in practice 244 of these plaintext/ciphertext pairs would be required to obtain the secret key, deemed by the paper's authors to be unlikely in the landscape of wireless sensor networks and

IoT. Additionally, the authors predicted that Skipjack's 80-bit key could be vulnerable to an exhaustive key attack within five years of the paper's writing. That date has since passed, and it is reasonable that this key can be brute forced by contemporary computing resources.

When testing AES, two platforms were used: A MICAz mote with 4 KB RAM and 512 KB NVM storage, and a Tmoke Sky mote with 10 KB RAM and 1 KB NVM storage. Both of these platforms contain a CC2420 radio chip with hardware security support, including encryption via AES-128. However, the chip does not provide the ability to perform AES-128 decryption, only encryption. This is likely a design choice, as forgoing decryption capabilities will produce a smaller chip and sensor motes often do not need to receive secure, encrypted data. Additionally, certain AES modes do not require separate decryption functions, such are CTR mode.

While the paper went over results from all algorithms, we will only discuss the results of the hardware and software AES as those are most relevant to this thesis. In performing actual encryption, hardware AES on the MICAz was performed 49 times faster than the software implementation. On the Tmote sky, hardware AES was only 4 times faster. In both experiments, the hardware AES implementation used less RAM than its software counterparts.

Shammi Didla's paper [20] is the closest in theme to this thesis, exploring their own AES optimizations in reducing RAM, ROM, and executable size for edge devices and outlining the methods used to achieve these optimizations. They even use a similar edge device as this thesis, a TI CC2420. The optimizations implemented are similar to the ones utilized in the algorithms we analyzed. The paper gives a concise overview of the types of optimizations often used: (i) specialization of code, (ii) varying data sizes, (iii) loop unrolling, (iv) function inlining, (v) reducing memory moves, (vi) eliminating local buffers, and the (vii) use of global variables.

Specialization of code would involve cutting out superfluous options that do not directly pertain to the task at hand. With regards to AES, this means only loading necessary tables for a given key size, e.g. 128, as different key sizes require different constants. This is explored more in this thesis in Results and Discussion Chapter (5)

Using varying data sizes that suit the processor size of the target machine understandably affects performance. For example, using 16-bit types on a 1-bit or greater processor will allow for more efficient operations, but would hard fail on a 8-bit machine.

Reducing memory moves, eliminating local buffers, and using global variables are all examples of trying to reduce scope and state changes as much as possible. Depending on the processor, changing local buffers or passing pointers compiles to varying numbers of operations, as different processors handle memory access differently. Having a solid understanding the processor, one can pick the option which will reduce the total number of operations the most. Taking this further, one can also exploit the C precompiler to define values to achieve a similar affect.

Loop unrolling and function inlining are similar optimizations in which loops of iterables and functions calls are replaced with repeated code, i.e. the entity that was being loops or the contents of a function. This reduces array accesses and scope changes but reduces code readability and greatly increases the source code size.

However, the paper showed that only some of these optimizations had a significant effect on RAM/ROM usage and encryption/decryption duration, such as specializing the code to the keysize, which led to a 184% increase in key expansion speed.

The paper also explains that GNU GCC has flags (O1, O2, O3) to automatically attempt its own optimizations such as loop unrolling, function inlining and register renaming. However, these automatic attempts did not result in the same gains and sometimes even had negative impacts, as was the case with the O3 compiler option which enables

automatic loop unrolling, resulting in a 14% decrease in encryption/decryption speed and also increasing code size by a factor of 3. In the paper's experiments, almost all the manual variation of the optimizations outperformed that of the automatic compiler flags.

Chapter 3

Symmetric Encryption

We chose to examine symmetric key encryption algorithm implementations primarily because most communication over the wire is done with symmetric encryption, whereas asymmetric algorithms are primarily used for key exchange protocols. For example, in the HTTPS and SSH protocols, after a secret key exchange, data in flight is encrypted with a symmetric key [21] [22]. For edge devices, this means that the majority of encryption is done with symmetric encryption, assuming that public key generation and symmetric key exchanges happen infrequently. This will especially be true if keys are loaded on ROM flash or pre-shared before the edge device has been fielded. The issue of regenerating a public-private key pair and constantly renegotiating symmetric key pairs will not be covered in this thesis, but will be discussed in future work.

We specifically chose the Rijndael AES algorithm due to its ubiquity and current status as an industry standard, as granted by NIST in 2001 [7]. There are many implementations of this algorithm in various languages, optimized for various platforms. This gave us a wide selection of fair use and open source algorithmic implementations to test. However, to understand how these different implementations are differently optimized for edge platforms, we must first understand how AES works in its various modes.

As a block cipher, AES operates on blocks, performing encryption per block of plaintext to produce a block of cipher text. AES always operates on 128-bit blocks regardless of key size (i.e. AES-128, AES-192, and AES-256 all use 128-bit blocks). These blocks are comprised of a 4x4 matrix of bytes, known as the state.

Table 3.1: An example AES state of a plaintext block in hexadecimal

19	a9	9a	e9
3d	f4	c6	f8
e3	e2	8d	48
be	2b	2a	08

Table 3.2: An example AES key, represented as 4x4 matrix in hexadecimal

2b	28	ab	09
7e	ae	f7	cf
15	d2	15	4f
16	a6	88	3c

In order to perform encryption on plaintexts that are larger than the key size, the key must first be expanded into round keys according to the Rijndael key schedule [7]. The first word of every 128-bits of the key expansion starts with the operation RotWord, which takes the previous word and rotates it leftwards. This transformation on the example key (Table 3.2) can be seen below:

$$\begin{bmatrix} 09 \\ cf \\ 4f \\ 3c \end{bmatrix} \Longrightarrow \begin{bmatrix} cf \\ 4f \\ 3c \\ 09 \end{bmatrix}$$

The result of RotWord then undergoes the substitution step (ii) SubBytes. This result is then XORed with the word four position previous (the first column of Table 3.2), and XORed again with the first round constant (Table 3.3).

The second, third, and fourth words of each 128-bit block is XORed with the word four positions previous, forgoing the RotWord transformation and SubBytes substitution.

Table 3.3: The AES Round Constants in hexadecimal

01	02	04	08	10	20	40	80	1B	36

Table 3.4: AES S-Box in hexadecimal

	0	1	2	3	4	5	6	7	8	9	0a	0b	0c	0d	0e	0f
0	63	7c	77	7b	f2	6b	6f	c5	30	1	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	4	c7	23	c3	18	96	5	9a	7	12	80	e2	eb	27	b2	75
4	9	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	0	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	2	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	6	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	8
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	3	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

After the creation of the first roundkey, AES will go through nine, eleven, or thirteen rounds of the following steps: (i) SubBytes, (ii) ShiftRows, (iii) MixColumns, and (iv) AddRoundKey. It will then go through a final round which omits the (iii) MixColumns step, only performing the (i) SubBytes, (ii) ShiftRows, and (iv) AddRoundKey steps. The omission of (iii) MixColumns in the last step makes for easier decryption, as it allows the decryption (inverse encryption) functions to better mirror the standard encryption ones.

(i) SubBytes is a simple substitution step where elements of the state are swapped out for elements according to a lookup table substitution box (S-Box). This S-Box is a constant generated from the multiplicative inverse over the finite field $GF(2^8)$. Notice how the first cell in the example state (Table 3.1) acts as a lookup guide in S-Box (Table 3.4), where the digit "1" refers to the row number and the digit "9" refers to the column number. This affine transformation can also be expressed mathematically with matrices (as below), with b_0 the i^{th} bit of a byte 01100011.

Table 3.5: AES state after SubBytes.

d4	e0	b8	1e
27	bf	b4	41
11	98	5d	52
ae	f1	e5	30

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

This step, in conjunction with AddRoundKey, is the primary way that AES achieves confusion. Confusion is a cryptographic property that measures how many parts the encryption key affects a single bit of the plaintext. The more confusion an encryption algorithm provides, the more difficult it is to make out the relationship between the plaintext and respective encryption key.

(ii) ShiftRows is a transposition step in which columns of the state are rotated. The first row is ignored, while the second row is shifted left 1 byte, the third row shifted left 2 bytes, and the fourth row shifted right 3 bytes. The transformation after ShiftRows from the current state (Table 3.5) can be seen in Table 3.6. This prevents the each column being encrypted independently with AES operating as four separate block ciphers instead of a single unified one.

(iii) MixColumns is a linear transformation step. The columns of the state are multiplied

Table 3.6: AES state after ShiftRows

d4	e0	b8	1e
bf	b4	41	27
5d	52	11	98
30	ae	f1	e5

Table 3.7: AES state after MixColumns

04	e0	48	28
66	cb	f8	06
81	19	d3	26
e5	9a	7a	4c

within $GF(2^8)$ by a constant matrix, one at a time.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} d4 \\ bf \\ 5d \\ 30 \end{bmatrix}$$

The product of these matrices result in the first column in Table 3.7, which shows the state after (iii) MixColumns. (iii) MixColumns in conjunction with (ii) Shift rows is the primary way AES achieves diffusion. Diffusion is a cryptographic property that measures of how much a given ciphertext would change if a single bit of the plaintext were flipped or how much a given plaintext would change if a single bit of the ciphertext were flipped.. The more diffusion that an encryption algorithm provides, the more difficult it is to make out the relationship between a plaintext and its respective ciphertext.

(iv) AddRoundKey is a combination step in which the state is XORed with the round

subkey.

$$\begin{bmatrix} 04 \\ 66 \\ 81 \\ e5 \end{bmatrix} \oplus \begin{bmatrix} a0 \\ fa \\ fe \\ 17 \end{bmatrix}$$

The (iv) AddRoundKey is the step in which the encryption key directly affects on the plaintext.

While delineated as separate steps, these operations can be combined in various arrangements, to reduce the total number of required operations to encrypt a block, thereby increasing throughput. This is an important consideration when optimizing for different goals and will be discussed in greater length in Results and Discussion Chapter 5.

Other than the actual key size, AES-128, AES-192, and AES-256 differ in the number of total rounds, with AES-128 running 10 rounds, AES-192 running 12 rounds, and AES-256 running 14 rounds. The way the key itself is generated also differs slightly, as different key sizes utilize different elements of the round constant word array rcon. This becomes important in optimizing for a smaller executable size, which will be discussed in greater length in Results and Discussion 5.

Chapter 4

Experimental Setup and Methodology

In this thesis, we aim to evaluate the performance of Rijndael AES, a representative symmetric key based cryptographic algorithm, on resource constrained IoT devices. We choose to study AES as a non-proprietary, standardized algorithm because it is used in modern information infrastructure (such as The Internet itself) and has been proven itself reasonably secure since it is the current Federal Information Processing Standard [7].

We tried and examined three AES implementations: TinyAES [23], B-Con's AES [24], and Contiki's own built-in AES [25].

The first implementation, TinyAES, is a "small and portable implementation of the AES ECB and CBC encryption algorithms written in C," written by kokke [23]. TinyAES provides four public functions, an encrypt and decrypt function for both ECD and CBC modes. The entire module uses less than 200 bytes of RAM and 2.3 KB ROM when compiled for 32-bit ARM. TinyAES has been optimized for 8-bit, 32-bit, and 64-bit processors. Porting TinyAES into contiki only requires copying and including the `aes.c` and `aes.h` files. Symlinking is a possible alternative, though was not implemented in our tests.

The second implementation, B-con, has written a collection of cryptographic algorithms, including AES in both ECB and CBC modes [24]. None of B-con's AES algorithms have

been optimized for speed or space. This library was included to test an AES library that was not optimized for 8-bit processors. Porting B-con's AES functions into Contiki was similar to the porting of TinyAES.

The third implementation, Contiki AES, is the builtin AES library that comes with contikiOS [25]. It can be found under `contiki/core/lib/` [26]. This is actually a wrapped implementation of Texas Instruments AES-128 implementation [27]. At the moment, regarding platform independent functions, Contiki only provides the core AES function, without supporting specific modes. ECB can be emulated by calling the AES function on subsequent blocks, but CBC must be written independently. No additional porting was necessary because this AES function is built-in to Contiki OS; however, since we needed an apples-to-apples comparison of algorithm encryption/decryption modes, we wrote our own CBC mode in accordance with NIST's *Recommendation for Block Cipher Modes of Operation* [28]. This source code is included in the Appendix.

To compare the energy and time cost of these implementations, we have adopted the following test cases: all symmetric key algorithms were AES-128 operating solely in the Cipher Block Chaining mode with constants KEY and IV (Initialization Vector); A zeroed array of words `IN` (plaintext input payload) of `SIZE` bytes was encrypted into a zeroed array of words `OUT` (encrypted output payload) of `SIZE` bytes. Array sizes from 8 bytes to 64 bytes were tested in steps of 8 bytes.

In terms of the evaluation platform, we use a RaspberryPi in tandem with a customized Energy Measurement Platform for Wireless IoT Devices (EMPIOT)[29] to interpret the triggers. This evaluation platform is primarily used for energy measurement.

The EMPIOT board allows us to physically measure power consumed in a given interval. This is advantageous over using analytical and simulation-based energy estimation schemes because the board measures the power used by a device's physical later, allow-

ing the measurement of systems which have not already been profiled for simulation. It also takes into account all of the attached peripheral systems.

To conduct the energy measurement portion of the experiment, we use two GPIO pins, connected from the power measurement board to the test board to act as the START and STOP triggers for the power measurement, with an additional grounding pin. A double male USB-A to Micro USB cable conducts the actual power measurement. Featuring a sampling rate of approximately 1000 Hz, EMPIOT is accurate to 0.4 μW in its energy measurement and has less than 3% error in energy measurement for IoT devices using 802.15.4 or 802.11 wireless standards. Therefore, it is used to collect all energy data presented in this work.

Custom software on a GMPIOT board uses triggers to measure the shunt voltage, amperage, and voltage at various clock intervals. By calculating a Lebesgue integral provided below we are able to calculate the total energy consumption of various operations. Due to the low energy nature of IoT platforms, 1000 trials are always performed in order to artificially increase the total energy. A 10 millisecond clock delay is also introduced to allow the measurement triggers to properly reset between iterations of each trial batch. The source code of all the mentioned custom software is included in the Appendix A.

The ARM embedded tool chain is used to compile each Contiki binary. We use Texas Instruments' own UniFlash to flash the binary to the CC2650 sensortag. The utility Screen was used to grab stdout and serial output from the cc2650 sensortag in order to verify certain information, such as the buffer size of the current trial. While Contiki comes with its own implementation of AES 128, it is only available in its most basic form, only supporting the Electronic Code Book (ECB) mode. As such, we develop our own C module to implement Cipher Block Chaining (CBC) mode. This implementation is not strictly cryptographically-secure as it was not made with side-channel attacks in

mind and has not been thoroughly tested.

Besides the energy consumption, we also evaluate the duration of a payload encryption and decryption, because a slow algorithm could reduce the overall performance and increase latencies for other operations. This is typically directly correlated with mathematical complexity: the more work a processor has to do, the longer it will take to produce a valid result. This can be measured with clock cycles, which can then be converted to a more human-friendly format, such as milliseconds.

The actual encryption and decryption duration measurements are performed using Contiki's builtin Rtimer library [26]. Rtimer was built to schedule real-time tasks. However, we only used the library to obtain the current system time in ticks by taking the current system time before and after the encryption or decryption of the input payload, taking the difference in ticks, and then converting the ticks to seconds via on a conversation constant based on the platform architecture. To maintain consistency with the energy measurements, 1000 trials were also performed per input payload.

Each algorithm was tested for both encryption and decryption on input payloads of 128, 256, 384, 512, 640, 768, 896, and 1024 bytes. Each input payload was hard-coded as an array of 8-bit unsigned integer hexadecimal bytes. A custom python script was then used to aggregate and average the collected data. The python plotting library *matplotlib* [30] was used to generate the graphs.

Chapter 5

Results and Discussion

In this work we mainly evaluated the energy consumption and encryption duration for different AES implementations. Our results on mean energy usage and encryption/decryption duration are shown in Figure 5.1 and Figure 5.2, respectively, in which the five curves represent the performance of the various encryption algorithm implementations. The x-axis is composed of the input sizes of the encryption or decryption payloads, which were multiples of 128 bites up to 1024 bites. The y-axis is the mean energy in nanojoules of encryption or decryption times in milliseconds plot points, which are then extrapolated to a best fitting function, which happens to be linear. Each encryption algorithm implementation has its own color and marker which can be seen in the key of both graphs: TinyAES encryption is represented by blue circle markers, TinyAES decryption by green triangle markers, B-Con's encryption by red square markers, B-Con's decryption by teal pentagon markers, and Contiki's built-in encryption and decryption by violet star markers. Error bars are included but are not visible as the standard de-

Table 5.1: Energy Consumption of TinyAES Encryption

Input Size	Mean	Standard Deviation
128	5776254.41	453748.090481
256	11316233.34	556543.545837
384	16851178.41	625003.229646
512	22307266.75	682791.689526
640	27880061.05	650531.153705
768	33418061.07	583860.227807
896	38919628.44	422866.189497
1024	44488234.96	350960.044748

Table 5.2: Energy Consumption of TinyAES Decryption

Input Size	Mean	Standard Deviation
128	9222666.73	582636.404009
256	18096184.37	638714.350689
384	27074312.20	629518.659772
512	35992270.97	446109.033318
640	44905314.57	361226.233756
768	53825850.26	568328.822880
896	62716041.24	644512.078094
1024	71679571.65	622131.081676

Table 5.3: Energy Consumption of bcon's AES Encryption

Input Size	Mean	Standard Deviation
128	34980536.19	480931.577144
256	69648778.67	778682.112150
384	104473726.13	662301.146915
512	139225982.05	582437.445015
640	174113421.20	462882.471642
768	208796428.57	627184.169925
896	243726691.69	657868.505597
1024	277728811.23	648033.439441

Table 5.4: Energy Consumption of bcon's AES Decryption

Input Size	Mean	Standard Deviation
128	37512281.31	461101.889541
256	74721708.89	615664.977466
384	111882055.08	647768.776925
512	149038857.20	655695.414154
640	186269443.71	581670.874856
768	223590153.77	339377.467727
896	261001405.10	422574.634364
1024	298185637.56	678970.709300

Table 5.5: Energy Consumption of Contiki's builtin AES Encryption

Input Size	Mean	Standard Deviation
128	5511917.57	471070.692758
256	10570225.27	581100.922584
384	15609257.91	722173.748751
512	20798382.28	638867.691422
640	25912167.82	611921.279023
768	30913768.54	665213.459255
896	36112398.92	466012.060723
1024	41262320.63	98961.8039541

Table 5.6: Timing of TinyAES Encryption

Input Size	Mean	Standard Deviation
128	19.1281605003	1.39136440974
256	34.6738419443	1.65430405547
384	50.1406183503	1.70571148401
512	65.5634304345	1.78799878172
640	81.1430832435	1.73572347509
768	96.7376708984	1.54846664546
896	112.282752991	1.17970977749
1024	128.113058339	0.857033079629

Table 5.7: Timing of TinyAES Decryption

Input Size	Mean	Standard Deviation
128	28.6338681080	1.56291429807
256	53.6867294633	1.77974669423
384	78.7305559431	1.73421748834
512	103.788340136	1.4023591396
640	129.099712113	0.873406994751
768	154.160738972	1.54908713729
896	179.228907797	1.77475800433
1024	204.290133804	1.7281423693

Table 5.8: Timing of B-con's AES Encryption

Input Size	Mean	Standard Deviation
128	101.560784152	1.40652522797
256	199.581337846	1.70730502491
384	297.543749397	1.73907246526
512	395.409915610	1.17621334125
640	493.353461653	1.17543460821
768	591.557512380	1.60771287288
896	689.600569479	1.78939773981
1024	787.689324581	1.63760184798

Table 5.9: Timing of B-con's AES Decryption

Input Size	Mean	Standard Deviation
128	108.534087405	1.18521548711
256	213.380601671	1.6684281947
384	318.463527795	1.7820109496
512	423.322100972	1.74191246103
640	528.178384859	1.39258588696
768	633.340040843	0.902440647994
896	737.896597959	1.42046695757
1024	843.266676479	1.56507014887

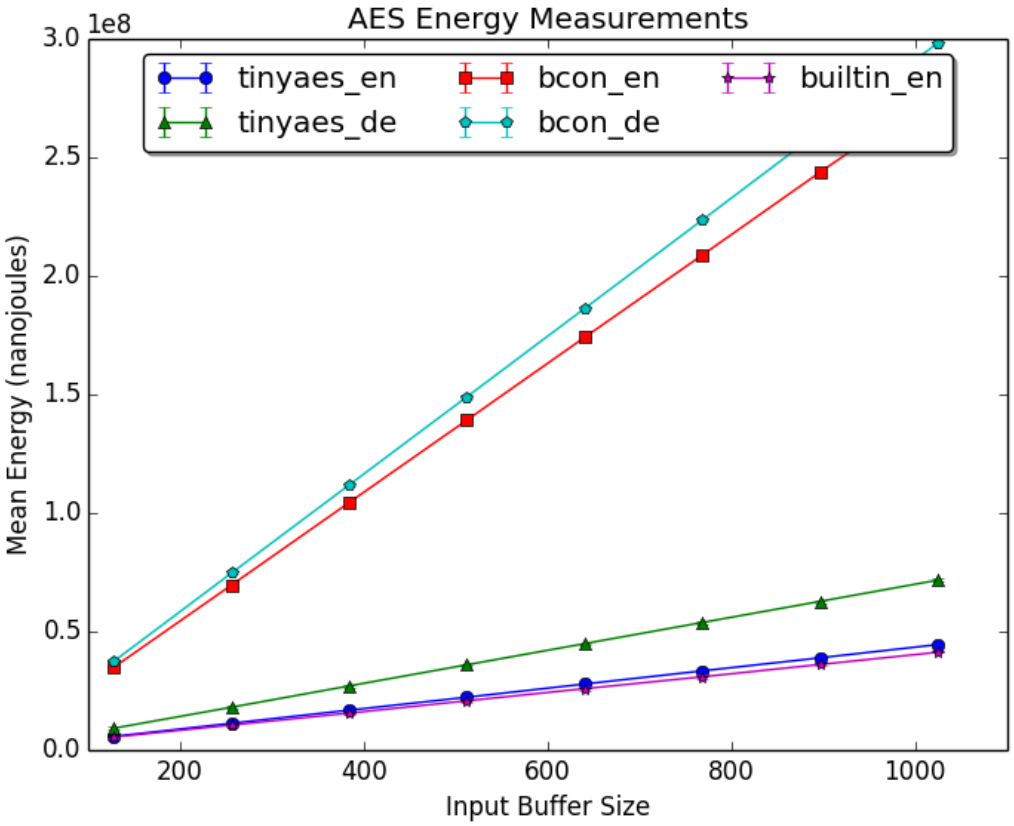

Fig. 5.1: Energy consumption of AES implementations

viation is so small compared to the y-axis values. The specific numbers in our findings on mean energy usage can be found in Tables 5.1, 5.2, 5.3, 5.4, and 5.5. The specific number on encryption/decryption duration can be found in Tables 5.6, 5.7, 5.8, 5.9, and 5.10.

All the algorithms scale linearly with input size, in terms of both energy consumption and time to perform the encryption/decryption. The durations are expected because AES is a block cipher encryption algorithm. Energy is also expected to scale linearly, although initial confidence was not as high, because it was unclear whether algorithmic implementations that were not optimized for a smaller processor would have unexpected adverse effects on energy, such as forcing a much larger drain beyond a certain input

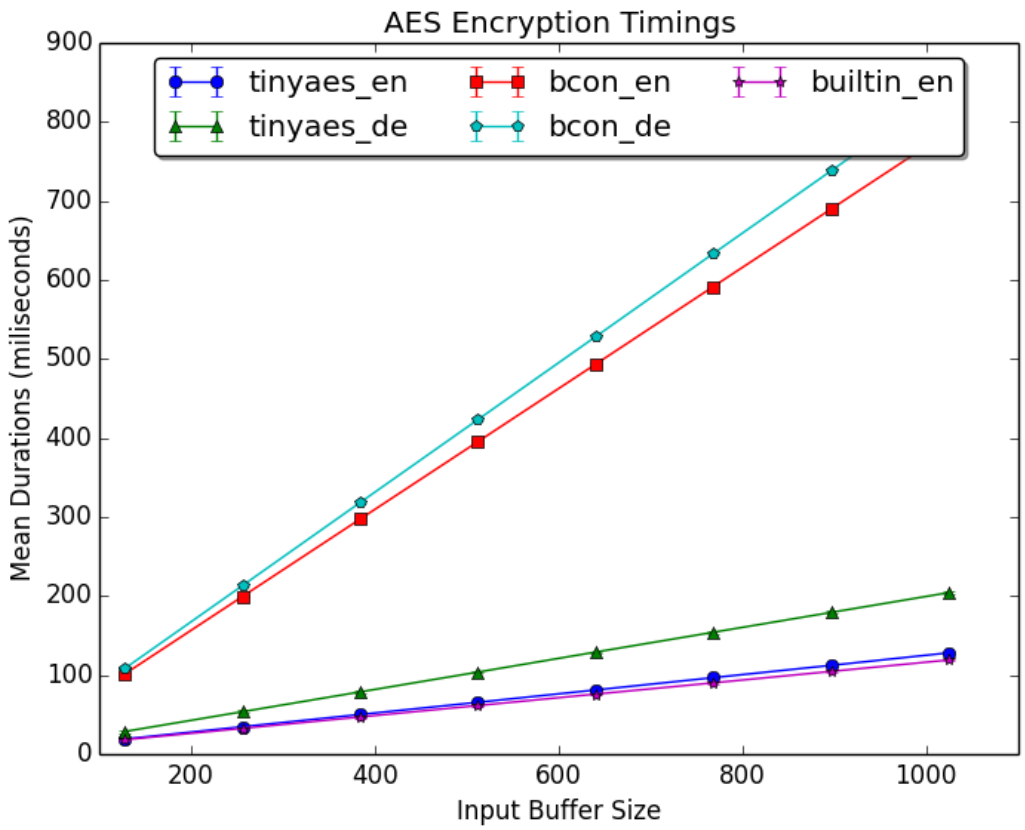

Fig. 5.2: Encryption duration of AES implementations

size.

Comparing the performance of the three implementations of AES, the results show that the best algorithm is TinyAES, an algorithmic implementation specifically optimized for smaller processors. B-Con's AES algorithm implementation, which is not optimized for smaller processors, performed more poorly. According to its author, "These algorithms are not optimized for speed or space. They are primarily designed to be easy to read, although some basic optimization techniques have been employed."

In general, it can be assumed that if the implementation is optimized for a larger processor, the algorithm would not run correctly on a smaller one. For example, on 32-bit or larger systems, one can combine the SubBytes and ShiftRows steps of AES, lever-

Table 5.10: Timing of Contiki's builtin AES Encryption

Input Size	Mean	Standard Deviation
128	18.217198989	1.24170762346
256	32.5560349684	1.55711460509
384	46.9968715009	1.73069093396
512	61.4654389187	1.78708865444
640	75.9161223406	1.74041673410
768	90.1617752878	1.66166751355
896	104.6128975330	1.40152716244
1024	119.0844699670	0.847051489196

aging 32-bit tables using 4096 bytes, something that is impossible on a 16-bit or lower architecture.

TinyAES and Contiki's built-in implementation for encryption perform similarly, with Contiki's outperforming TinyAES by a hair. At an input size of 1024 bits, Contiki's built-in implementation used about 8% less energy and time to perform encryption than that of TinyAES (Tables 5.1, 5.5, 5.6, 5.8). TinyAES' implementation used about 84% less energy and time than that of B-Con's (Tables 5.1, 5.3, 5.6, 5.8). The aggregated data shows that specialized algorithms such as TinyaES and Contiki's built-in AES algorithm perform better on their target platforms. This is of no surprise, as these algorithms were quite literally designed to perform better on these systems.

This being said, it should also be noted that AES was originally designed with the criteria of high speed performance on low RAM devices with processors as small as 8-bits. This, however, does not mean we cannot further optimize its implementations for speed and energy consumption, especially with the contemporary ubiquity of constrained devices.

We investigate the three algorithmic implementations in further detail to better understand how they achieve their performance. AES contains many constant components, such as the S-BOX lookup tables and round constants (rcon) [31]. All three algorithms declare these as static constant arrays to leverage ROM versus RAM. as a result, this

frees up memory when memory is considered a scarce resource. TinyAES offers the ability to generate the S-BOX tables dynamically, which would trade ROM for RAM. This makes the TinyaES implementation more flexible, as ROM can be the limiting factor in certain IoT devices. B-Con's implementation pre-calculates all possible calculations of the Galois Field, mainly multiplication. This Galois Field is used in the MixColumns step of AES, making the operation a two-dimensional array access a faster, vectorized, multiple step multiplication that can operate on multiple values at the same time.

In general, B-Con's implementation uses a lot of double array accesses. Additionally, both TinyAES and Contiki's implementation attempt to perform as many operations in a single step as possible. It also seems that TinyAES tried to limit the number of variable declarations. Although most compilers should have optimization options for constant folding/propagation [32], which should negate any advantages this would hope to gain. However, as we had seen in Shammi Didla's findings, automatic compiler optimization does not always provide positive results, though manually accommodating code for optimization, as in `aes.c` of TinyAES, always did (at least within their experiments).

The following code contains two multiplication helper functions. Notice how TinyAES gives the option to declare the function either as an actual function or a macro. Within the actual function, near the last bitshift, the last call to `xtime()` (line 13) is actually redundant and unneeded. However, in kokke's experiments, omitting it tends to create a larger binary, suggesting that the last call somehow helps the compiler vectorize the function better.

Listing 5.1: TinyAES's aes.c

```
1 static uint8_t xtime(uint8_t x)
2 {
3   return ((x<<1) ^ (((x>>7) & 1) * 0x1b));
```

```
4  }
5
6  #if MULTIPLY_AS_A_FUNCTION
7  static uint8_t Multiply(uint8_t x, uint8_t y)
8  {
9    return (((y & 1) * x) ^
10         ((y>>1 & 1) * xtime(x)) ^
11         ((y>>2 & 1) * xtime(xtime(x))) ^
12         ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^
13         ((y>>4 & 1) * xtime(xtime(xtime(xtime(x))))); /* this
               last call to xtime() can be omitted */
14   }
15 #else
16 #define Multiply(x, y)                                \
17       (  ((y & 1) * x) ^                              \
18       ((y>>1 & 1) * xtime(x)) ^                       \
19       ((y>>2 & 1) * xtime(xtime(x))) ^                \
20       ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^         \
21       ((y>>4 & 1) * xtime(xtime(xtime(xtime(x)))))    \
22
23 #endif
```

We also examined the number of instructions these implementations compiled to for 32-bit ARM without any special compiler options. For example TinyAES's SubBytes function compiled to 308 operations while B-Con's SubByte's function compiles to 544 operations. For this particular function, this is primarily because TinyAES simply uses less array accesses and only makes two 2-dimensional array access compared to

Listing 5.2: "TinyAES SubBytes()"

```
1  static void SubBytes(state_t* state)
2  {
3    uint8_t i, j;
4    for (i = 0; i < 4; ++i)
5    {
6      for (j = 0; j < 4; ++j)
7      {
8        (*state)[j][i] = getSBoxValue((*state)[j][i]);
9      }
10   }
11 }
```

B-Con's implementation, which makes three 2-dimensional array accesses (including assignment) while also performing bitwise operations on the index. TinyAES also uses a pointer to get the head of the array. Examples of this can be seen in listings 5.2 and 5.3. Overall, it does not seem like TinyAES' and Contiki's AES implementations use drastically different coding methods than that of B-con's implementation. The optimized algorithmic implementations simply limit superfluous code, reducing overall total operations.

TinyAES also manages to achieve a smaller executable size compared to the other algorithm implementations. TinyAES goes further in its static declarations of constants, even omitting some of the indices of the round constant word array (Rcon) altogether, requiring the AES key size to be determined beforehand on start-up. As AES-128, AES-196- and AES-256 all use different indices of Rcon and none of them use the zeroth index [31]. TinyAES also chooses to forgo certain conveniences, such as automatically padding inputs to match the 128 bit block size. TinyAES even has the option to define multiplication as a function or macro, to further reduce the executable size depending on the compiler used. For example, using the Keil ARM compiler [33], defining multiplication as a function reducing the executable from 2,087 bytes to 1,268 bytes. Conversely, when

Listing 5.3: "B-Con's AES SubBytes()"

```
1  void SubBytes(BYTE state[][4]
2  )
3  {
4     state[0][0] = aes_sbox[state[0][0] >> 4][state[0][0] & 0x0F];
5     state[0][1] = aes_sbox[state[0][1] >> 4][state[0][1] & 0x0F];
6     state[0][2] = aes_sbox[state[0][2] >> 4][state[0][2] & 0x0F];
7     state[0][3] = aes_sbox[state[0][3] >> 4][state[0][3] & 0x0F];
8     state[1][0] = aes_sbox[state[1][0] >> 4][state[1][0] & 0x0F];
9     state[1][1] = aes_sbox[state[1][1] >> 4][state[1][1] & 0x0F];
10    state[1][2] = aes_sbox[state[1][2] >> 4][state[1][2] & 0x0F];
11    state[1][3] = aes_sbox[state[1][3] >> 4][state[1][3] & 0x0F];
12    state[2][0] = aes_sbox[state[2][0] >> 4][state[2][0] & 0x0F];
13    state[2][1] = aes_sbox[state[2][1] >> 4][state[2][1] & 0x0F];
14    state[2][2] = aes_sbox[state[2][2] >> 4][state[2][2] & 0x0F];
15    state[2][3] = aes_sbox[state[2][3] >> 4][state[2][3] & 0x0F];
16    state[3][0] = aes_sbox[state[3][0] >> 4][state[3][0] & 0x0F];
17    state[3][1] = aes_sbox[state[3][1] >> 4][state[3][1] & 0x0F];
18    state[3][2] = aes_sbox[state[3][2] >> 4][state[3][2] & 0x0F];
19    state[3][3] = aes_sbox[state[3][3] >> 4][state[3][3] & 0x0F];
20 }
```

using the Mentor Bench ARM GCC toolchain [34], compiling with multiply defined as a macro creates a smaller executable than when defining as a function, respectively 2,087 bytes and 2,130 bytes. TinyAES also uses many `#ifndef` directives to allow the option of only loading the functions that are required (e.g. only using ECB). B-Con's executable, on the other hand, is 190,245 bytes when compiled for 32-bit ARM. This is likely due to the numerous pre-calculated, hard-coded, two-dimensional arrays.

As is the case with TinyAES, it seems different compilers will produce different executable sizes despite building from the same source code and for the same target architecture. If the best way to reduce encryption duration and energy consumption is to time and measure the energy of each instruction, and only attempt to only leverage instructions that use the fewest clock cycles and least amount of power, then the compiled binaries *per compiler* must be examined for the number of these few clock cycle and low power instructions. Another technique could be simply reducing the total number of instructions by combining multiple operations, as seen in the TinyAES and Contiki implementations.

These results demonstrate that spending the effort to further reduce the energy consumption of IoT devices, allowing for a longer field life. This would also take less maintenance to swap out power cells (or the entire IoT device), which further reduces labor costs.

Chapter 6
Conclusion

In this thesis, we survey the state of security of the Internet of Things landscape and found that IoT faces many unique challenges that traditional security solutions cannot yet address. This is primarily due to the sheer number of motes that exist within an IoT network and motes' resource constrains, such as lower-bit processors, less storage capacity, less RAM, and less electrical power. We also survey attempts at optimizing traditional algorithmic implementations for constrained devices. We found successful optimizations that leverages hardware and software techniques. In our final surveys, we find some hard data regarding the performance some of these optimizations compared to that of traditional algorithms.

Our own findings show that two encryption algorithm implementations optimized for constrained devices, TinyAES and Contiki's built-in AES, performed better than B-Con's AES, which was not optimized for constrained devices. The optimized algorithm implementations yield smaller executables, provide faster encryption/decryption run times, and reduce overall power consumption for the platforms they are optimized for. Specifically, optimized AES implementations used about 0.16 of the energy and time to complete encryption and decryption compared to unoptimized implementations.

We investigate the cause of these gains and found the energy and time savings was primarily achieved through a variety of manual optimizations. These include optimizations in leveraging ROM versus RAM, minimizing total operations, and taking advantage of vectorized instructions. The actual reduction in executable size, run times, and power

consumption may vary depending on the compiler used, even for the same source code and target architecture. These optimizations are not dependent on the actual functions of Rijndael AES, which means that these optimization techniques can be utilized as to optimize algorithmic implementations in general.

In combination with the data sets in the literature we surveyed, we hope the quantifiable metrics found during our experiments further assist in evaluating optimized security solution for the IoT landscape.

Bibliography

[1] Mark Hung. *Leading the IoT*. Gartner, 2017. URL https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf. 1

[2] Matej Mikulic. Global number of pacemakers in 2016 and a forecast for 2023 (in million units), sep 2019. URL https://www.statista.com/statistics/800794/pacemakers-market-volume-in-units-worldwide/. 1

[3] Kristen Linsalata. Recall: Abbott pacemakers for hacking threat, 2017. URL https://www.webmd.com/heart/news/20170905/recall-abbott-pacemakers-for-hacking-threat. 1

[4] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. DDoS in the IoT: Mirai and other botnets. *Computer*, 50(7):80–84, 2017. doi: 10.1109/mc.2017.201. URL https://doi.org/10.1109/mc.2017.201. 1

[5] John Gutekunst. Docks flooded as failed dam sensor causes water levels to rise near parker, apr 2019. URL https://www.havasunews.com/news/docks-flooded-as-failed-dam-sensor-causes-water-levels-to/article_0b34d75b-335a-5a72-bf63-6ec44ffe7a60.html. 1

[6] Okamura Toshihiko. Lightweight cryptography applicable to various iot devices. *NEC Technical Journal*, 12, 2017. 2, 8

[7] National Institute of Standards and Technology. Federal information processing standard 197, the advanced encryption standard (aes). Technical re-

port, 2001. URL https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf. 2, 14, 15, 20

[8] Cc2650. URL http://www.ti.com/product/CC2650. 2

[9] Kai Zhao and Lina Ge. A survey on the internet of things security. In *Proceedings of the 2013 Ninth International Conference on Computational Intelligence and Security*, CIS '13, pages 663–667, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-1-4799-2549-0. doi: 10.1109/CIS.2013.145. URL https://doi.org/10.1109/CIS.2013.145. 4

[10] Caroline Fontaine Jacques Fournier Benjamin Lac MarÄśa Naya-Plasencia Renaud Sirdey et al. Anne Canteaut, Sergiu Carpov. End-to-end data security for iot: from a cloud of encryptions to encryption in the cloud. In *Cesar Conference 2017*, 2017. 5

[11] H. Suo, J. Wan, C. Zou, and J. Liu. Security in the internet of things: A review. In *2012 International Conference on Computer Science and Electronics Engineering*, volume 3, pages 648–651, March 2012. doi: 10.1109/ICCSEE.2012.373. 6

[12] X. Wang, J. Zhang, E. M. Schooler, and M. Ion. Performance evaluation of attribute-based encryption: Toward data privacy in the iot. In *2014 IEEE International Conference on Communications (ICC)*, pages 725–730, June 2014. doi: 10.1109/ICC.2014.6883405. 6

[13] Mastooreh Salajegheh. Software techniques to reduce the energy consumption of low-power devices at the limits of digital abstractions. 2013. doi: 10.7275/js4x-0t46. URL https://scholarworks.umass.edu/open_access_dissertations/704. 7

[14] Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka, and Eita Kobayashi. Twine: A lightweight block cipher for multiple platforms. In Lars R. Knudsen

and Huapeng Wu, editors, *Selected Areas in Cryptography*, pages 339–354, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-35999-6. 8

[15] P. Hamalainen, T. Alho, M. Hannikainen, and T. D. Hamalainen. Design and implementation of low-area and low-power aes encryption hardware core. In *9th EUROMICRO Conference on Digital System Design (DSD'06)*, pages 577–583, Aug 2006. doi: 10.1109/DSD.2006.40. 9

[16] S. Mangard, M. Aigner, and S. Dominikus. A highly regular and scalable aes hardware architecture. *IEEE Transactions on Computers*, 52(4):483–491, April 2003. ISSN 2326-3814. doi: 10.1109/TC.2003.1190589. 9

[17] Abdalla Adam Abdalla Yousif Hassan. Evaluation of encryption algorithms for iot security. Master's thesis, University of Almughtaribeen, 2017. 10

[18] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In Ross Anderson, editor, *Fast Software Encryption*, pages 191–204, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. ISBN 978-3-540-48456-1. 10

[19] Michael Healy, Thomas Newe, and Elfed Lewis. Analysis of hardware encryption versus software encryption on wireless sensor network motes. In *Smart Sensors and Sensing Technology*, pages 3–14. Springer, 2008. 10

[20] Shammi Didla, Aaron Ault, and Saurabh Bagchi. Optimizing aes for embedded devices and wireless sensor networks. In *Proceedings of the 4th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*, TridentCom '08, pages 4:1–4:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-963-9799-24-0. URL http://dl.acm.org/citation.cfm?id=1390576.1390581. 11

[21] A. Schiffman E. Rescorla. The secure hypertext transfer protocol. RFC 2660, IETF, August 1999. URL https://tools.ietf.org/html/rfc2660. 14

[22] T. Ylonen. The Secure Shell (SSH) Protocol Architecture. RFC 5251, IETF, January 2006. URL https://tools.ietf.org/html/rfc4251. 14

[23] kokke. Small portable aes128/192/256 in c. https://github.com/kokke/tiny-AES-c, 2019. 20

[24] Brad Conte. Basic implementations of standard cryptography algorithms, like aes and sha-1. https://github.com/B-Con/crypto-algorithms, 2015. 20

[25] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Nov 2004. doi: 10.1109/LCN.2004.38. 20, 21

[26] 2018. URL https://github.com/contiki-os/contiki. 21, 23

[27] Uli Kretzschmar. *AES128 âĂŞ A C Implementation for Encryption and Decryption.* Texas Instruments, 2009. URL https://e2e.ti.com/cfs-file/__key/communityserver-discussions-components-files/156/slaa397a.pdf. 21

[28] Morris J. Dworkin. Sp 800-38a 2001 edition. recommendation for block cipher modes of operation: Methods and techniques. Technical report, Gaithersburg, MD, United States, 2001. 21

[29] Behnam Dezfouli, Immanuel Amirtharaj, and Chia-Chi Li. Empiot: An energy measurement platform for wireless iot devices. *Journal of Network and Computer Applications*, 121, 04 2018. doi: 10.1016/j.jnca.2018.07.016. 21

[30] Paul Barrett, John Hunter, J Todd Miller, J-C Hsu, and Perry Greenfield. matplotlib–a portable python plotting package. In *Astronomical data analysis software and systems XIV*, volume 347, page 91, 2005. 23

[31] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002. ISBN 3-540-42580-2. 29, 32

[32] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, April 1991. ISSN 0164-0925. doi: 10.1145/103135.103136. URL http://doi.acm.org/10.1145/103135.103136. 30

[33] *ARM Compiler v5.06 for ÂţVision armcc User Guide*. ARM, 2016. URL http://infocenter.arm.com/help/topic/com.arm.doc.dui0375g/DUI0375G_mdk_armcc_user_guide.pdf. 32

[34] *Sourcery CodeBench*. Mentor. URL https://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/overview. 34

Appendix A
Source Code

A.1 CBC Mode for Contiki's AES implementation

```
1  static void
2  encrypt_cbc(uint8_t * in, uint8_t * key, uint8_t * iv, unsigned
       long size){
3      uint8_t xor[1024] = {0};
4
5      int blocks = size / 16;
6
7      for(int k = 0; k < blocks; k++){
8          int j = k * 16;
9          for(int i = 0; i < 16; i++){
10             in[i+j] = in[i+j] ^ iv[i];
11         }
12         aes_128_driver.encrypt(in+j);
13         iv = in+j;
14     }
15 }
16
17 const struct aes_128_driver aes_128_driver = {
18   set_key,
```

```
19     encrypt,
20     encrypt_cbc
21 };
```

A.2 Sample Clock Cycle Measurement Script

measure_clock_cycle.c

```
 1
 2 //Santa Clara University
 3 //Internet of Things Research Lab (SIOTLAB)
 4 //2017
 5
 6 #include "contiki.h"
 7 #include "ti-lib.h"
 8 #include "sys/etimer.h"
 9 #include "sys/ctimer.h"
10 #include "dev/leds.h"
11 #include "power_measurement.h"
12 #include "cpu/cc26xx-cc13xx/clock.c"
13 #include "sys/clock.h"
14 #include "dev/tiny-AES128-C/aes.c"
15
16 #include <stdio.h>
17 #include <stdint.h>
18
19 #define LOOP_INTERVAL   (150)
```

```
20 #define CBC 1
21 #define SECOND 1000000
22
23 static struct etimer et;
24 static struct ctimer timer;
25
26 volatile bool status = false;
27
28 PROCESS(sensortag_led_experiment, "sensortag_led_experiment");
29 AUTOSTART_PROCESSES(&sensortag_led_experiment);
30
31 void dump(uint8_t * str, unsigned long size){
32     for(int i = 0; i < size; i++){
33         printf("%.2x", str[i]);
34     }
35     printf("\n");
36 }
37
38 static void process_task(void *ptr) {
39
40     // Local Variables
41
42     // Timer Variables
43     unsigned long time_start;
44     unsigned long time_stop;
45     unsigned long cycles;
46
47     // encryption variables
```

```
48      const uint8_t SIZE = 16 * 7; // 128 bytes, 1024 bits
49      const uint16_t PAYLOAD_SIZE = SIZE * 8;
50
51      uint8_t key[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0
           xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
52      uint8_t iv[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0
           x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
53
54      uint8_t in[]  = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f,
           0x96, 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
55          0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0
               xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
56          0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0
               xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
57          0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0
               x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10,
58          0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0
               x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
59          0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0
               xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
60          0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0
               xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
61          0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0
               x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10}; //8 - 1024
62
63      uint8_t in[2048] = {0};
64      uint8_t out[2048] = {0};
65
```

```
66      // Driver
67      printf("Size: %d\n", PAYLOAD_SIZE);
68      printf("PRE out: ");
69      dump(out, SIZE);
70
71      leds_on(LEDS_RED);
72      // start_power_measurement();
73      time_start = RTIMER_NOW();
74
75      // ================ //
76      // START MEASURING //
77      // ================ //
78
79      uint16_t TRIALS = 100;
80      for(int i = 0; i < TRIALS; i++){
81          AES_CBC_encrypt_buffer(out, in, SIZE, key, iv);
82      }
83      // ================ //
84      // END MEASURING //
85      // ================ //
86
87      // end_power_measurement();
88      time_stop = RTIMER_NOW();
89      cycles = time_stop - time_start;
90
91      printf("POST out: ");
92      dump(out, SIZE);
93
```

```
94      printf("START: %lu\n", time_start);
95      printf("STOP: %lu\n", time_stop);
96      printf("cycles,rtimer_second: %lu %d\n", cycles,
            RTIMER_SECOND);
97      printf("MILISECONDS: %g\n", miliseconds);
98
99      leds_off(LEDS_RED);
100
101     ctimer_reset(&timer);
102 }
103
104 PROCESS_THREAD(sensortag_led_experiment, ev, data)
105 {
106     PROCESS_BEGIN();
107     printf("CC26XX LED Experiment\n");
108
109     clock_init();
110     etimer_set(&et, LOOP_INTERVAL);
111     ctimer_set(&timer, LOOP_INTERVAL/2, process_task, NULL);
112     init_power_measurement();
113
114     // Time to sleep in microseconds (e.g. 1000000 = 1 second)
115
116     while(1) {
117
118         // sleep 1 seconds
119         // clock_delay_usec takes uint16_t, so a for loop was
                the best way to abstract
```

```
120             uint16_t SLEEP_MILISECONDS = 10;
121             for(int i = 0; i < SLEEP_MILISECONDS; i++){
122                 clock_delay_usec(1000);
123             }
124
125           PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
126
127           // returns the current system time in clock ticks
128           printf("Clock time: %lu\n", clock_time());
129
130           // returns the current system time in seconds
131           printf("Clock seconds: %lu\n", clock_seconds());
132
133           // printf("Toggle red LED\n");
134           etimer_reset(&et);
135
136               }
137
138       PROCESS_END();
139   }
```

A.3 Sample Energy Measurement Script

power_measurement.h

```
1
2  //Santa Clara University
```

```
3  //Internet of Things Research Lab (SIOTLAB)
4  //2017
5
6  #ifndef POWER_MEAS
7  #define POWER_MEAS
8
9  #include "contiki.h"
10 #include "ti-lib.h"
11 #include "sys/etimer.h"
12 #include "sys/ctimer.h"
13 #include "dev/leds.h"
14
15 void init_power_measurement ()
16 {
17     GPIO_setOutputEnableDio(BOARD_IOID_DP0, 1 );
18     GPIO_setOutputEnableDio(BOARD_IOID_DP2, 1 );
19
20     GPIO_setDio(BOARD_IOID_DP0);
21     GPIO_setDio(BOARD_IOID_DP2);
22 }
23
24 // Commands the power measurement device to start measurement
25 void start_power_measurement()
26 {
27     GPIO_clearDio(BOARD_IOID_DP0);
28     GPIO_setDio(BOARD_IOID_DP0);
29 }
30
```

```
31
32 // Commands the power measurement device to stop measurement
33 void end_power_measurement()
34 {
35       GPIO_clearDio(BOARD_IOID_DP2);
36       GPIO_setDio(BOARD_IOID_DP2);
37 }
38
39 #endif POWER_MEAS
```

processing_energy.c

```
1  //Santa Clara University
2  //Internet of Things Research Lab (SIOTLAB)
3  //2017
4
5  #include "contiki.h"
6  #include "ti-lib.h"
7  #include "sys/etimer.h"
8  #include "sys/ctimer.h"
9  #include "dev/leds.h"
10 #include "power_measurement.h"
11 #include "cpu/cc26xx-cc13xx/clock.c"
12 #include "sys/clock.h"
13 #include "dev/tiny-AES128-C/aes.c"
14
15 #include <stdio.h>
16 #include <stdint.h>
```

```
17
18 #define LOOP_INTERVAL   (150)
19 #define CBC 1
20 #define SECOND 1000000
21
22 static struct etimer et;
23 static struct ctimer timer;
24
25 volatile bool status = false;
26
27 PROCESS(sensortag_led_experiment, "sensortag_led_experiment");
28 AUTOSTART_PROCESSES(&sensortag_led_experiment);
29
30 void dump(uint8_t * str, unsigned long size){
31     for(int i = 0; i < size; i++){
32         printf("%.2x", str[i]);
33     }
34     printf("\n");
35 }
36
37 static void process_task(void *ptr) {
38
39     // Local Variables
40
41     // Timer Variables
42     unsigned long time_start;
43     unsigned long time_stop;
44     unsigned long cycles;
```

```
45
46      // encryption variables
47      const uint8_t SIZE = 16 * 7; // 128 bytes, 1024 bits
48      const uint16_t PAYLOAD_SIZE = SIZE * 8;
49
50      uint8_t key[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0
           xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
51      uint8_t iv[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0
           x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
52
53      uint8_t in[]  = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f,
           0x96, 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
54          0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0
               xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
55          0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0
               xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
56          0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0
               x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10,
57          0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0
               x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
58          0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0
               xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
59          0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0
               xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
60          0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0
               x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10}; //8 - 1024
61
62      uint8_t in[2048] = {0};
```

```
63      uint8_t out[2048] = {0};
64
65      // Driver
66      printf("Size: %d\n", PAYLOAD_SIZE);
67      printf("PRE out: ");
68      dump(out, SIZE);
69
70      leds_on(LEDS_RED);
71      start_power_measurement();
72
73      // ================ //
74      // START MEASURING //
75      // ================ //
76
77      uint16_t TRIALS = 100;
78      for(int i = 0; i < TRIALS; i++){
79          AES_CBC_encrypt_buffer(out, in, SIZE, key, iv);
80      }
81      // ============== //
82      // END MEASURING //
83      // ============== //
84
85      end_power_measurement();
86
87      printf("POST out: ");
88      dump(out, SIZE);
89
90      leds_off(LEDS_RED);
```

```
91
92      ctimer_reset(&timer);
93  }
94
95  PROCESS_THREAD(sensortag_led_experiment, ev, data)
96  {
97      PROCESS_BEGIN();
98      printf("CC26XX LED Experiment\n");
99
100     clock_init();
101     etimer_set(&et, LOOP_INTERVAL);
102     ctimer_set(&timer, LOOP_INTERVAL/2, process_task, NULL);
103     init_power_measurement();
104
105     // Time to sleep in microseconds (e.g. 1000000 = 1 second)
106
107     while(1) {
108
109         // sleep 1 seconds
110         // clock_delay_usec takes uint16_t, so a for loop was
                the best way to abstract
111         uint16_t SLEEP_MILISECONDS = 10;
112         for(int i = 0; i < SLEEP_MILISECONDS; i++){
113             clock_delay_usec(1000);
114         }
115
116         PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
117
```

```
118         // returns the current system time in clock ticks
119         printf("Clock time: %lu\n", clock_time());
120
121         // returns the current system time in seconds
122         printf("Clock seconds: %lu\n", clock_seconds());
123
124         // printf("Toggle red LED\n");
125         etimer_reset(&et);
126
127             }
128
129     PROCESS_END();
```