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

Brandon Tsao
btsao@scu.edu
Santa Clara University
Santa Clara, California

Yuhong Liu
yhliu@scu.edu
Santa Clara University
Santa Clara, California

Behnam Dezfouli
bdezfouli@scu.edu
Santa Clara University
Santa Clara, California

## ABSTRACT

With the proliferation of IoT, securing the abundance of devices is critical. The current IoT and security landscapes lack empirical evidence on algorithms optimized for constrained devices. In this paper, 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 symmetric encryption algorithm implementations of AES (tinyAES, B-Con's AES, and Contiki's own built-in AES), where we found algorithms specifically built for constrained devices fared much better than those not, optimized algorithms using about 0.16 the energy and the time to perform encryption and decryption.

## CCS CONCEPTS

• **Security and privacy → Block and stream ciphers**; • **Computer systems organization → Sensor networks**; • **Hardware → Wireless integrated network sensors**; **Chip-level power issues**.

## KEYWORDS

cryptography, Internet of things, security, sensor networks

**ACM Reference Format:**
Brandon Tsao, Yuhong Liu, and Behnam Dezfouli. 2019. Analysis of the Duration and Energy Consumption of AES Algorithms on a Contiki-based IoT Device. In *16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous), November 12–14, 2019, Houston, TX, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3360774.3368202

## 1 INTRODUCTION

The Internet of Things (IoT) is taking the world by storm, with its use cases spanning the full spectrum: from added consumer convenience to life-critical systems. By the year 2020, there will be over 20 billion digital devices [14], which also means 20 billion distinct possible vectors for a cyber attack. For example, implanted medical devices such as pacemakers are already common [19], and while the Internet connectivity will allow streamlined patches from retailers, it also paves a path for potential hackers, as with the case with St. Jude Medical's Quadra Allure MP [17]. 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 wide scale attacks, such as when a Mirai IoT Botnet [1] 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 [11]. 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 forces the use of smaller processors with fewer registers and less cache memory. Size is not the only consideration for smaller processors, as less circuits usually means less power consumption. While this can be good, it also means many IoT devices may not be directly wired to the power grid. While wired solutions exist, battery power allows for faster, cheaper, and more flexible installations in many cases. As most IoT devices are constrained in terms of both processing power and energy consumption, it can be difficult to implement secure cryptographic algorithms and operations that are known for their heavy footprint in both processing and energy. The apparent dichotomy is quite unfortunate.

Regardless, a compromise must be reached if IoT devices are to be secured. There has been much work trying to create security algorithms and protocols that specifically address the limitations of IoT [23]. Regardless of their viability, new 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" and many algorithms must be adapted to fit its own unique needs.

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 AES. We conduct experiments using multiple traditional and modified algorithm implementations on a constrained device, specifically the TI CC2650 [2]. The CC2650


*MobiQuitous, November 12–14, 2019, Houston, TX, USA*

ACM ISBN 978-1-4503-7283-1/19/11…$15.00
https://doi.org/10.1145/3360774.3368202

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 if optimized algorithms yield better results in both of these aspects, and if they did, by how much. Additionally, we investigate why they fared better, by examining the source code of the algorithms. We found what one might expect: that optimized algorithms 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 paper is organized as follows: Related work is examined in Section 2. Experimental setup and methodology is explained in Section 3. Section 4 discusses results and discussions. We conclude the paper in Section 6.

## 2 RELATED WORK

There have been a few surveys on the state of security regarding IoT. Kai Zhao and Lina Ge [25] performed a high level analysis in 2013, suggesting IoT does not only have to deal with the same security issues as other information domains, but faces other unique issues such as privacy protection and heterogeneous network authentication and authorization. A similar paper [13] outlines some empirical data collected on various symmetric key algorithms, such as AES and Blowfish [21]. We will expand on some of these findings later in this paper.

Canteaut et al. [4] 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. Due to the popularity of various block ciphers such as AES, block-based stream ciphers can be utilized, e.g. AES in CTR mode. However, these modes still fall behind in throughput and latency compared to that of dedicated stream ciphers. Therefore, dedicated stream ciphers are still preferred. Specifically, the following ciphers have been heavily considered by them: Grain v1, HC-128, MICKEY, Rabbit, Salsa20/12, Sosemanuk, and Trivium.

Salajegheh et al. [20] 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 PUSHes and POPs a program has to carry out, which reduces overall energy expensive operations. In the realm of encryption algorithms, however, these techniques might not be suitible 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; (iii) local variables and functions must be limited, forcing redundant code which will increase the size of the binary.

Additionally, we must consider the use case. 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 also requires a costly context switch, more than likely to undo the savings gained by globalizing variables.

One of NEC's technical journal [23] also proposes a new algorithms altogether, specifically designed for constrained devices. NEC's own TWINE algorithm [22] 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 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. While hardware AES chips already exist, Hamalainen et al. [12] have created chips specifically designed to minimize the number of gates, therefore reducing overall size and power consumption. However, such solutions would require hardware editions to each constrained device, as well as firmware/software to interface with it for encryption/decryption. Such would likely incur an increased financial and possible redesign cost upfront. This needs to be considered a trade off, as this initial cost might still reduce costs in the long run if lower power consumption leads to less maintenance or longer field life.

## 3 EXPERIMENTAL SETUP AND METHODOLOGY

In this paper, we aim to evaluate the performance of Advanced Encryption Standard (AES), a representative symmetric key based cryptographic algorithm, on resource constrained IoT devices. We choose AES as a non-proprietary, standardized algorithm to study because it is used in modern information infrastructure (such as The Internet itself) and have been proven to be reasonably secure.

In particular, three AES implementations were tried and examined, including: TinyAES [15], B-Con's AES [7], and Contiki's own built-in AES [9]. The first implementation, TinyAES, is a "small and portable implementation of the AES ECB and CBC encryption algorithms written in C," written by kokke [15]. TinyAES provides four public functions, an encrypt and decrypt function for each mode. 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 myriad of cryptographic algorithms, including AES in both ECB and CBC modes [7]. 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 that of TinyAES. The third implementation, Contiki AES, is the builtin AES library that comes with contikiOS [9]. It can be found under contiki/core/lib/ [3]. This is actually a wrapped implementation of Texas Instruments AES-128 implementation [16]. 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. As this AES function is built-in to Contiki OS, no additional porting was necessary. 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* [10]. 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, a RaspberryPi was used in tandem with a customized Energy Measurement Platform for Wireless IoT Devices (EMPIOT)[**?** ] to interpret the triggers. Two GPIO pins, connected from the power measurement board to the test board, acted as the START and STOP triggers for the power measurement. An additional grounding pin was also used. The actual power measurement occurred through a double male USB-A to MicroUSB cable. Featuring a sampling rate of approximately 1000 Hz, EMPIOT is accurate to 0.4 $\mu$W in its energy measurement and able to boast less than 3% of energy measurement error for IoT devices using 802.15.4 or 802.11 wireless standards. Therefore, it was used to collect all energy data presented in this work.

Custom software on a GMPIOT board used triggers to measure the shunt voltage, amperage, and voltage at various clock intervals. By calculating a Lebesgue integral provided below we were able to calculate the total energy consumption of various operations. Due to the low energy nature of IoT platforms, 1000 trials were always performed in order to artificially increase the total energy. A 10 millisecond clock delay was 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 will be included.

The ARM embedded toolchain was used to compile each Contiki binary. Texas Instruments' own UniFlash was then used to flash the binary to the CC2650 sensortag. The utility Screen was used to grab stdout / serial output from the cc2650 sensortag, as 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 Codebook (ECB) mode. As such, we developed our own C module to implement CBC (Cipher Block Chaining) 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 evaluated the duration for encrypting/decrypting a payload, as a slow algorithm could reduce the overall performance and increase latencies for other operations. This is typically directly correlated with mathematical complexity; the more a processor has to work, the longer it will take to produce a valid result. This can be measured via clock cycles, which can then be converted to a more human-friendly format, such as milliseconds.

## 4 RESULTS AND DISCUSSION

In this work, we mainly evaluated the energy consumption and timing for different AES implementations. Our results are shown in Figure 1 and Figure 2, in which the five curves represent the performance of TinyAES encryption and decryption, B-Con's encryption and decryption, and the builtin AES encryption, respectively.

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

Comparing the performance of the three implementations of AES, the results show that the best algorithm was TinyAES, which was specifically optimized for smaller processors. B-Con's AES algorithm implementation, which was 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." It can be assumed that if the implementation was 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, leveraging 32-bit tables using 4096 bytes, something impossible on a 16-bit or lower architecture. TinyAES and Contiki's built-in algorithms for encryption performed similarly, with Contiki's outperforming TinyAES by a hair. 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, even ranging to embedded systems with as little as 8-bit processors. This does not mean we cannot further optimize, especially with the contemporary ubiquity of constrained devices.

To better understand how these three implementations achieve their performance, we further investigated their implementation details. Since much of the components of AES are constant, such as the S-BOX lookup tables and round constants (rcon) [8], all three algorithms declare these arrays as static constants as to leverage ROM versus RAM, freeing up memory when memory is considered a scarce resource. TinyAES offers the ability to dynamically generate the S-BOX tables, trading ROM for RAM, as ROM can be the limiting factor in certain IoT devices, making it's implementation

more flexible. B-Con's implementation, on the other hand, also pre-calculates all possible calculations (primarily multiplication) of the Galois Field. This Galois Field is used in the MixColumns step of AES, making the operation a two-dimensional array access versus a faster, vectorized, multiple step multiplication (which 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 as a single step as possible. It also seems that TinyAES tried to limit the number of variable declarations. Though most compilers should have optimization options for constant folding/propagation [24], which would negate any advantages this would hope to gain.

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 544. For this particular function, this is primarily because TinyAES simply uses less array accesses, only making two 2-dimensional array access compared to 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, but this should compile into more or less the same code. Overall, it does not seem TinyAES and Contiki's AES implementation use drastically different coding methods than that of B-con's implementation. The optimized algorithms 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 startup, as AES-128, AES-196- and AES-256 all use different indices of Rcon, and the zeroth index is actually used by none of them [8]. 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, as this can further reduce the executable size depending on the compiler used. For example, using the Keil ARM compiler [5], defining multiplication as a function reducing the executable from 2,087 bytes to 1,268 bytes. Conversely, using the Mentor Bench ARM GCC toolchain [18], the executable size is smaller when compiling with multiply defined as a macro than when defined as a function, respectively 2,087 bytes to 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, 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 attempt to only leverage instructions that use the fewest clock cycles and least amount of power, the compiled binaries *per compiler* must be examined for the number of these few clock cycle / low power instructions. Another technique could be simply reducing the total number of instructions, as did TinyAES and Contiki's implementation, by combining multiple operations.

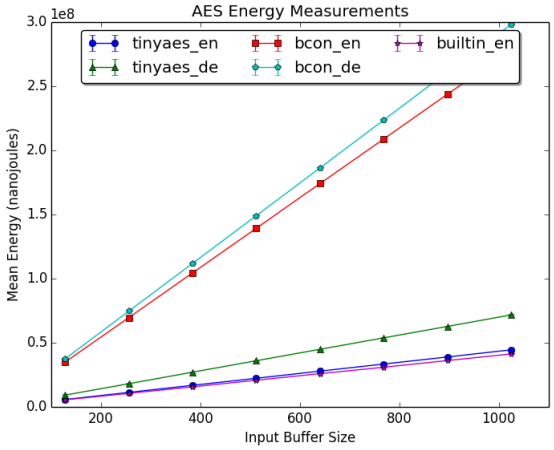


**Figure 1: Energy consumption of AES implementations**

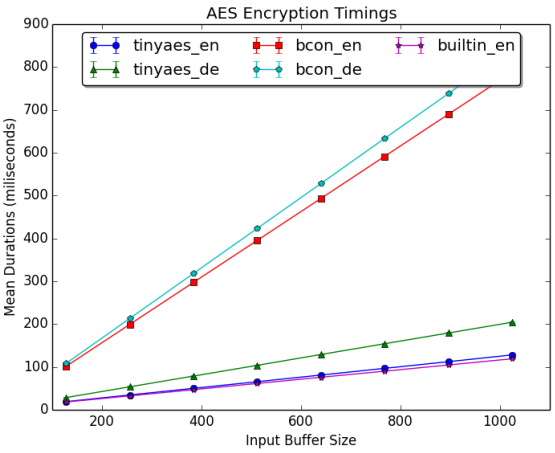


**Figure 2: Encryption duration of AES implementations**

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 mean less maintenance to swap out power cells (or the entire IoT device), further reducing costs in the form of labor.

## 5 FUTURE WORK

To truly evaluate the value of these algorithm specific implementations for constrained devices, the encryption itself should be tested against side-channel attacks. Optimized algorithms might forgo certain security measures to save on clock cycles. Others that focus on saving energy could increase the risk of a power analysis side channel attack. Modifying memory usage opens the door for memory or cache based side channel attacks.

We should briefly touch on the differences of environment that IoT normally lives in, compared to that of normal computing devices. It is an information security motif that physical access to a device opens up a plethora of new threats. These threats can be so great that sometimes the device might as well be considered compromised. This can be addressed by introducing physical barriers such as locked doors, metal cages for server racks, security cameras for monitoring, etc. Nevertheless, IoT devices tend to be scattered in an environment in large numbers, usually to collect data. This very nature of IoT makes it difficult to physically secure these devices in the wild. Even cataloging these devices could be difficult. For example, flood sensors on a river bank have no physical protection (other than hopefully being waterproof). If physical access to these devices is more feasible, then it is necessary to harden them against attack vectors that physical access would grant. This includes making algorithms more resistant to side channel attacks that leverage energy usage. However, masking energy usage may involve calling dummy functions in order to keep power levels randomized as to not reveal certain heavier cryptographic operations. Such masking would incur its own additional energy drain to the device. This highlights the need for specific algorithms that cater to IoT devices over generic ones, as IoT encryption algorithms need to account not only for efficiency, but for elevated resistance against physical side channel attacks.

AES was designed with a wide performance range; it should not only run at an acceptable speed on low performance devices but should scale up to higher speeds on high performance devices [6]. This was partially achieved via the instruction of various modes. ECB and CTR (counter) modes both allow for the parallel encryption of multiple blocks simultaneously, unlike CBC mode, which requires the previous block in order to properly encrypt the next. Parallel processing in these contexts usually requires the use of multiple threads, a usually expensive feature set that may not exist on constrained device operating systems, such as Contiki OS. However, many contemporary constrained devices offer high performance features such as multi-threading, albeit with trade offs such as increased energy complexity consumption. It would be worthwhile to compare these trade offs in various devices, especially in ones whom's power source and input can vary (e.g. a solar powered mote might want to use multi-threading if the weather is extremely sunny and its battery is more or less full).

Other experimental investigations with energy consumption and timing (and all the peripheral trade offs) would also be useful for key sharing frameworks and algorithms, and whether they use public-key cryptography to establish shared symmetric keys. What if a shared symmetric key is compromised? How expensive would a key re-share be? What about initial versus subsequent shares? These questions are already common for managed devices, but the environment of IoT once again pulls in other considerations, such as the lack of power source and unprecedented quantity of devices.

## 6 CONCLUSION

In this paper, we found that optimizing encryption algorithm implementations, at least for AES, yields smaller executables, provides faster encryption/decryption run times, and reduces overall power consumption for the platforms they are optimized for. In our findings, optimized AES implementations used about 0.16 the energy and took only 0.16 the time to complete encryption and decryption compared to those not optimized.

These energy and time savings was primarily achieved through the trade offs of leveraging ROM/RAM, minimizing total operations, and taking advantage of vectorized instructions.

The actual reduction in executable size, runtimes, and power consumption may vary depending on the compiler used, even for the same source code and target architecture. Run times and power consumption can be further reduced by the addition of a dedicated hardware module.

## REFERENCES


[1] [n. d.]. ([n. d.]). https://doi.org/10.1109/mc.2017.201
[2] [n. d.]. CC2650. http://www.ti.com/product/CC2650
[3] 2018. https://github.com/contiki-os/contiki
[4] Caroline Fontaine Jacques Fournier Benjamin Lac Maria Naya-Plasencia Renaud Sirdey et al. Anne Canteaut, Sergiu Carpov. 2017. End-to-end data security for IoT: from a cloud of encryptions to encryption in the cloud. In *Cesar Conference 2017.*
[5] ARM 2016. *ARM Compiler v5.06 for µVision armcc User Guide.* ARM.
[6] Doug Whiting David Wagner Chris Hall Bruce Schneier, John Kelsey. 1999. Performance Comparison of the AES Submissions.
[7] Brad Conte. 2015. Basic implementations of standard cryptography algorithms, like AES and SHA-1. https://github.com/B-Con/crypto-algorithms.
[8] Joan Daemen and Vincent Rijmen. 2002. *The design of Rijndael: AES — the Advanced Encryption Standard.* Springer-Verlag. 238 pages.
[9] A. Dunkels, B. Gronvall, and T. Voigt. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks.* 455–462. https://doi.org/10.1109/LCN.2004.38
[10] Morris J. Dworkin. 2001. *SP 800-38A 2001 Edition. Recommendation for Block Cipher Modes of Operation: Methods and Techniques.* Technical Report. Gaithersburg, MD, United States.
[11] John Gutekunst. 2019. Docks flooded as failed dam sensor causes water levels to rise near Parker. Retrieved September 28, 2019 from https://www.havasunews.com/news/docks-flooded-as-failed-dam-sensor-causes-water-levels-to/article_0b34d75b-335a-5a72-bf63-6ec44ffe7a60.html
[12] P. Hamalainen, T. Alho, M. Hannikainen, and T. D. Hamalainen. 2006. Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core. In *9th EUROMICRO Conference on Digital System Design (DSD'06).* 577–583. https://doi.org/10.1109/DSD.2006.40
[13] Abdalla Adam Abdalla Yousif Hassan. 2017. *Evaluation of encryption algorithms for IOT security.* Master's thesis. University of Almughtaribeen.
[14] Mark Hung. 2017. *Leading the IoT.* Gartner. https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf
[15] kokke. 2019. Small portable AES128/192/256 in C. https://github.com/kokke/tiny-AES-c.
[16] Uli Kretzschmar. 2009. *AES128 – A C Implementation for Encryption and Decryption.* Texas Instruments.
[17] Kristen Linsalata. 2017. Recall: Abbott Pacemakers for Hacking Threat. https://www.webmd.com/heart/news/20170905/recall-abbott-pacemakers-for-hacking-threat
[18] Mentor [n. d.]. *Sourcery CodeBench.* Mentor.
[19] Matej Mikulic. 2019. Global number of pacemakers in 2016 and a forecast for 2023 (in million units). Retrieved September 28, 2019 from https://www.statista.com/statistics/800794/pacemakers-market-volume-in-units-worldwide/
[20] Mastooreh Salajegheh. 2013. Software Techniques to Reduce the Energy Consumption of Low-Power Devices at the Limits of Digital Abstractions. *Open Access Dissertations.*
[21] Bruce Schneier. 1994. Description of a new variable-length key, 64-bit block cipher (Blowfish). In *Fast Software Encryption*, Ross Anderson (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 191–204.
[22] Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka, and Eita Kobayashi. 2013. TWINE: A Lightweight Block Cipher for Multiple Platforms. In *Selected Areas in Cryptography*, Lars R. Knudsen and Huapeng Wu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 339–354.
[23] Okamura Toshihiko. 2017. Lightweight Cryptography Applicable to Various IoT Devices. *NEC Technical Journal* 12 (2017).

[24] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 181–210.
[25] Kai Zhao and Lina Ge. 2013. A Survey on the Internet of Things Security. In *Proceedings of the 2013 Ninth International Conference on Computational Intelligence and Security (CIS '13)*. 663–667.

# A SOURCE CODE

## A.1 CBC Mode for Contiki's AES implementation

```
static void
encrypt_cbc(uint8_t * in, uint8_t * key,
    uint8_t * iv, unsigned long size){
    uint8_t xor[1024] = {0};

    int blocks = size / 16;

    for(int k = 0; k < blocks; k++){
        int j = k * 16;
        for(int i = 0; i < 16; i++){
            in[i+j] = in[i+j] ^ iv[i];
        }
        aes_128_driver.encrypt(in+j);
        iv = in+j;
    }
}

const struct aes_128_driver aes_128_driver = {
  set_key,
  encrypt,
  encrypt_cbc
};
```

## A.2 Sample Clock Cycle Measurement Script

measure_clock_cycle.c

```

//Santa Clara University
//Internet of Things Research Lab (SIOTLAB)
//2017

#include "contiki.h"
#include "ti-lib.h"
#include "sys/etimer.h"
#include "sys/ctimer.h"
#include "dev/leds.h"
#include "power_measurement.h"
#include "cpu/cc26xx-cc13xx/clock.c"
#include "sys/clock.h"
#include "dev/tiny-AES128-C/aes.c"

#include <stdio.h>
#include <stdint.h>

#define LOOP_INTERVAL    (150)
#define CBC 1
#define SECOND 1000000


static struct etimer et;
static struct ctimer timer;

volatile bool status = false;

PROCESS(sensortag_led_experiment, "
    sensortag_led_experiment");
AUTOSTART_PROCESSES(&sensortag_led_experiment)
    ;

void dump(uint8_t * str, unsigned long size){
    for(int i = 0; i < size; i++){
        printf("%.2x", str[i]);
    }
    printf("\n");
}

static void process_task(void *ptr) {

    // Local Variables

    // Timer Variables
    unsigned long time_start;
    unsigned long time_stop;
    unsigned long cycles;

    // encryption variables
    const uint8_t SIZE = 16 * 7; // 128 bytes,
         1024 bits
    const uint16_t PAYLOAD_SIZE = SIZE * 8;

    uint8_t key[] = {0x2b, 0x7e, 0x15, 0x16, 0
        x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0
        x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
    uint8_t iv[] = {0x00, 0x01, 0x02, 0x03, 0
        x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0
        x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

    uint8_t in[]  = { 0x6b, 0xc1, 0xbe, 0xe2,
        0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0
        x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
        0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0
            xac, 0x9c, 0x9e, 0xb7, 0x6f, 0xac,
             0x45, 0xaf, 0x8e, 0x51,
        0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0
            xe4, 0x11, 0xe5, 0xfb, 0xc1, 0x19,
             0x1a, 0x0a, 0x52, 0xef,
        0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0
            x9b, 0x17, 0xad, 0x2b, 0x41, 0x7b,
             0xe6, 0x6c, 0x37, 0x10,
        0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0
            x9f, 0x96, 0xe9, 0x3d, 0x7e, 0x11,
             0x73, 0x93, 0x17, 0x2a,
        0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0
            xac, 0x9c, 0x9e, 0xb7, 0x6f, 0xac,
             0x45, 0xaf, 0x8e, 0x51,
```

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

```
60          0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0
                x9b, 0x17, 0xad, 0x2b, 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();
130 }
```