Självständigt arbete på grundnivå

Independent degree project – first cycle

Datateknik
Computer engineering

Tiny security
- Evaluating energy use for security in an IoT application

Mårten Söderquist

Mittuniversitetet
MID SWEDEN UNIVERSITY

MITTUNIVERSITETET

DSV Östersund

Examiner	Felix Dobslaw	Felix.Dobslaw@miun.se
Supervisor	Raja-Khurram Shahzad	Raja-Khurram.Shahzad@miun.se
Author	Mårten Söderquist	maso1008@student.miun.se
Programme	Programvaruteknik, 180hp	
Course	DT133G, Självständigt Arbete	
Field of study	Computer Engineering	
Semester, year	VT, 2019	

Abstract

IoT devices are increasingly used in the process of gathering scientific data. In environmental monitoring IoT devices can be used as remote sensing devices to collect information about e.g. temperature. To keep data reliable, various security aspects have to be considered. Constrained devices are limited by memory size and battery life, a security solution has to be developed with this in mind. In this study an IoT security solution was developed in collaboration with a research group in environmental science at Umeå University. We selected commonly used algorithms and compared them with the goal to provide authentication and integrity for an IoT application, while minimizing energy use running on an Atmega 1284P. The results showed that the encryption algorithm AES-256-GCM is a good choice for a total security solution. AES-256-GCM provides authenticated encryption with additional data while, in relation to the other tested algorithms, using energy at a low level and leaving a small program size footprint.

Keywords: Security, constrained device, AES, IoT, sensors, AEAD, ATmega1284P, ACORN128

Table of Contents

Table of Tables

Table of Figures

Acronyms

AEAD	Authenticated Encryption with Additional Data
HMAC	Hash-Based Message Authentication Code
IETF	The Internet Engineering Task Force
JSON	Javascript Object Notation
EtM	Encrypt-then-MAC
GPRS	General Packet Radio Service
AES	Advanced Encryption Standard
SDK	Software Development Kit

1 Introduction

A common method to gather data during field research in environmental science is to use automated loggers that store data for later retrieval [1]. This process is workable but presents many problems. Locations are often remote meaning long hours traveling back and forth to maintain equipment and collect data. Since loggers are not observable there is an insecurity of not knowing the state of the equipment. If a logger is lost all data is gone with it. Loss can occur either due to environmental factors such as weather or due to tampering. This is in terms of data security a problem of availability; the data is not easily accessible. In order to solve this problem a logger could send data back to a central server. This gives immediate access to data in real time and the state of a logger can also be observed, making maintenance easier to plan. However, this setup presents a different set of problems related to data security; sent data needs to be verified for integrity and the sender needs to be authenticated [2].

This leaves an application using GPRS to then have to provide an application level implementation of any security that is needed. Determining which security algorithms are suitable for an environmental monitoring system is two-fold. Firstly, an acceptable level of security must be achieved. Secondly, an implementation must minimize the resource use of the device it is running on, in order not to interfere with the main function of the device. Limiting factors are mainly the battery life of a device and available space for storing program code. When developing the system one also needs to take into account resource use as these devices run on batteries and the deployment time must not be affected too greatly by any method used to secure the data. Trade-offs will have to be made in order to balance security level with resource use.

A research group at the Department of Ecology and Environmental Science focusing on environmental science is developing a remote sensing system for environmental data. The system will send data via GPRS to a server at the department. In the context of this development this study will examine ways to achieve various security goals while minimizing the resource use.

1.1 Problem Statement

In order to securely send data from a remote sensing device there needs to be a way to authenticate the device and verify the integrity of the data agnostic to what transport protocol is used. This requires processing on the remote device and these devices are resource limited. They have low memory capacity, slow processing speed and limited available energy since they run on batteries. In order to maximize deployment time any algorithms used to achieve the desired security must do so with the lowest possible use of resources. Furthermore an implementation should not severely affect the available space for other mission critical code. To find a suitable security implementation the answers to the following questions are sought:

- How does each algorithm affect energy use when preparing to send a message?
- What is the cost for the level of security for each algorithm in terms of energy?
- How much does each algorithm implementation affect the compile size?

1.2 Aim

The aim is to find a suitable security solution for authentication and data integrity in the context of the environmental research project, while minimizing resource use. A secondary goal is the inclusion of encryption for achieving data confidentiality.

1.3 Scope

Algorithms will only be tested on the hardware that has been selected for use in the context of the larger project. They must also follow Kerckhoff's principle [3]. Third party implementations of the algorithms written for the hardware will be used, but different implementations of the same algorithm will not be tested against each other. Algorithms that have been proven insecure for the purpose they would be used for in the this setting will not be investigated [13]. Different ways reduce power use by the hardware such as sleep modes and lower clock speeds will also not be investigated. Neither will different compiler settings for reducing compile sizes and their effects on performance be investigated.

1.4 Outline

Chapter 1 – Introduces the problem and sets the scope and limitations for the project

Chapter 2 – Relates the project to other work and gives a summary of cryptographic concepts.

Chapter 3 – Presents the methodology used in the project

Chapter 4 – Discusses the construction of implemented software.

Chapter 5 – presents the results.

Chapter 6 – discusses the results and suggests explanations for observed trends.

Chapter 7 – relates the results to the outset of the project and gives a suggestion to which algorithm to use.

2 Background

In the last decade the use of simple devices with network capacity has exploded. Dubbed the Internet of Things (IoT) these devices often have a specific purpose for which they are used and run autonomously. Typical uses include gathering data for different purposes or to enable remote control of processes. Another common trait for IoT-devices is that they generally have low performance in terms of processing power and memory capacity. Collectively these are called constrained devices. IoT has been adopted in almost every field imaginable from home automation and surveillance to monitoring of industrial processes. [2,4,5]

Since IoT-devices have so many uses there are many security related challenges and for every application these are different. Devices may be deployed in areas where electrical disturbances, low bandwidth or connectivity can be a factor, corrupting messages sent to and from a device. Other challenges are tampering resistance and outright hacking via the network connection. If the data collected is sensitive, eavesdropping on the communication presents a serious risk. [5]

Security in IoT is a vital and hot topic for research but an effect of the large variation in use and hardware is that no common standard for providing general data security has yet emerged. In essence, security in IoT devices is no different than that used on regular computers and the algorithms and theory is the same. However, the challenges are different due to the nature of the devices and where they are used. [2,4]

2.1 Terminology

For the purpose of this thesis the reader should be aware of some basic terminology used in the field of cryptography. Below is a brief summary of the more important concepts relevant to the thesis.

2.1.1 The CIA triad

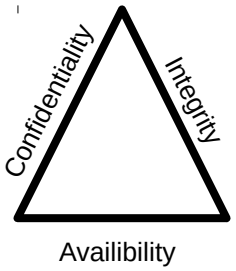

Figure 1: The CIA triad symbolizes that data security depends on the three concepts equally.

In data security the CIA triad (Figure 1) is a model of the aspects that need to be taken into consideration to achieve good data security. CIA is an acronym of Confidentiality, Integrity and Availability. Confidentiality is the aspect controlling who can access information. The term thus also encompasses authentication – verifying the identity – of an entity. Integrity refers to that data should be verifiably correct and complete at all times in the system. Included in this term is that data should not be modifiable by an unauthorized party. Finally, Availability is that data in the system should be possible to access for authorized entities when it is needed. As is obvious all three of these aspects are dependent upon each other to be meaningful; keeping data secret (Confidentiality) is only meaningful if the data is correct (Integrity) and can be accessed (Availability). [6]

2.1.2 Hash function

A hashing function takes input data of arbitrary size and, by means specific to each algorithm, calculates a value that maps to the given input. This value is called a hash value, digest or colloquially hash. The size of a hash is dependent upon which algorithm is used but is for any given algorithm always the same. In cryptography a subset of hash functions called cryptographic hash functions are used. These functions have several properties important to data security. They are designed to be one-way meaning that for a given hash value it is difficult, if not impossible, to reverse the process to obtain the original input data. Another property is that the output of a function is difficult to predict. This is important because it prevents the construction of messages that match the hash of another message thwarting forgery. Hash functions are used to ensure the integrity of a message. Being able to construct other message that produce a match to a given hash leaves the receiver no way of knowing if the message is what was originally sent. Thus, hash functions in themselves do not provide message authentication but they do, however, form the foundation for Hash-Based Message Authentication Codes (HMAC) described below. Common hash functions in use today are SHA-2 and SHA-3 [7, 8]. Older well known hash functions are SHA-1 used in for instance the Bittorrent protocol and MD5 [9, 10, 11], in wide use for integrity validation of downloaded files. These older functions have been proven less secure and should not be used in new cryptography applications [12].

2.1.3 Message Authentication Code

This term refers to a cryptographic construct for authentication of data. A Message Authentication Code (MAC), sometimes referred to as an authentication tag or simply tag, can be viewed as a type of checksum that can authenticate a message. Using a signing algorithm a key is used to compute the MAC for a given message. The key scheme is symmetric; parties communicating need to exchange the key securely in order for the MAC to be secure. From this also follows that a MAC does not provide non-repudiation of a message, that is it can not be proven which of the parties with access to the key has sent the message. [13]

2.1.4 HMAC

Using a cryptographic hash function together with a shared secret key two parties can verify the authenticity of sent messages. This process is called HMAC and is a subset of MAC. HMAC can be used with any cryptographic hash function. If the initial key sharing is done securely an HMAC provides a way for communicating parties to verify both the integrity, since that is the primary purpose of a hash function, as well as the authenticity of the data. HMAC works by incorporating the secret key when calculating the hash value. Note that this is not encryption; the message is still sent in plaintext. A secure implementation of HMAC will generally perform two hash calculations as shown in equation 1.

$$HMAC = H(key || H(key || message)) \quad (1)$$

The final HMAC is calculated from the key concatenated to the hash of the key concatenated to the message. This approach makes it very hard to guess the key even if

the message is known. A variant is to use two different keys derived from an original for each of the hashing rounds. [14]

2.1.5 Block cipher

These are a common type of ciphers that work on data of arbitrary length. The name block cipher comes from the shared property that they work on fixed sized chunks, referred to as blocks, of the input data. Encryption is done on each block in turn and the size of the block is determined by the algorithm used. Block ciphers use symmetric keys. Encryption is performed using a key to apply different permutations, bit shifts and other operations on the plaintext to produce the ciphertext. This is done several times, referred to as rounds, and the number of rounds is different for each algorithm. If the input data is not a multiple of the block size it is padded to achieve this. The complexity of a block cipher is generally constant time, O(1), for each block so the time for encrypting a messages is only dependent on the message length, m, making the time complexity of a block cipher O(m). [15]

2.1.6 Block cipher modes and Initialization Vectors

Blocks can be independently processed, a property which lends itself to parallelized implementations. However this approach is somewhat naive. Since the same key is used for encryption on every block, blocks with equal plaintext will always produce the same ciphertext revealing recurring patterns. This effect is visualized in Figure 2. An algorithm can instead use different ways to scramble the encryption process for each block. The way this is done is referred to as the mode of the algorithm. The previously described mode, where each block is independent of each other is called Electronic Code Book (ECB).

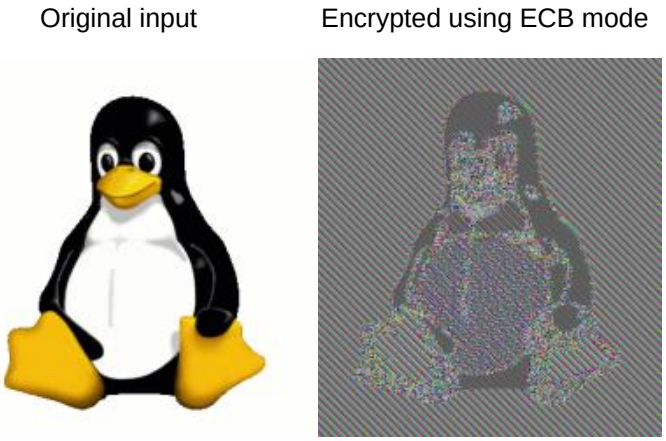

Figure 2: An image encrypted using the ECB mode of a block cipher. Using encryption does not always hide the meaning of data. [16]

To counter this weakness other modes have been developed that use additional data for each block to produce a seemingly random appearance of the final ciphertext even for recurring patterns in the plaintext. An early form developed was Cipher Block Chaining (CBC). While this is an outdated mode its simplicity still illustrates the effect of using different modes. In CBC mode an XOR is performed on the plaintext of a block with the ciphertext of the previous block before encryption. This achieves a chain reaction to produce

a different output for each block even if the original plaintext is the same for each block. However, since the first block does not have a previous block a value called the Initialization Vector (IV)[1] is used for this block. The IV is a random value used as a seed for the rest of the chain to start from. It is not important to keep the IV secret but it is necessary not to reuse the value and to generate a new IV for each encryption operation. Failure to do so will reveal patterns in the first block if several encrypted messages are intercepted by an attacker. In order to decrypt a message both the key and the IV has to be known. There are several other modes which try different ways to obfuscate patterns in plaintext. Noteworthy is Galois/Counter Mode (GCM) which is the presently recommended mode since it provides authenticated encryption [17]. [15]

2.1.7 Stream cipher

Similarly to the principle of the One-time pad, where the key has the same length as the plaintext, stream ciphers work by generating a pseudorandom bitstream, the same length as the data to be encrypted, the keystream. This keystream is seeded with the secret key and sometimes an IV depending on the algorithm. The plaintext stream is then encrypted using the keystream one bit at a time by an XOR operation producing the ciphertext (Figure 3). Decryption is then the process in reverse; performing an XOR of the keystream on the ciphertext produces the plaintext.

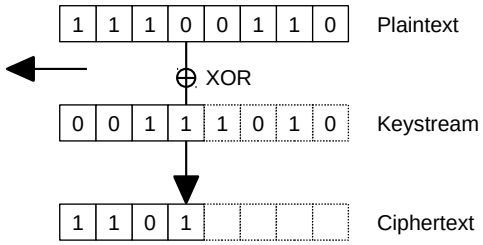

Figure 3: A stream cipher generates a pseudorandom keystream on-the-fly. Each bit of the plaintext is XOR with the corresponding bit of the keystream generating the ciphertext stream. Decryption is the same process but takes the ciphertext as input instead of plaintext.

The time complexity of stream ciphers have the same time complexity as block ciphers; message length is the limiting factor making them O(*m*), however stream cipher implementations are generally very fast due to their simplicity. [15]

1 A general term for this type of one time use values is *nonce*

2.1.8 Authenticated Encryption (with additional Data)

Authenticated Encryption with additional Data (AEAD) is a term describing methods which guarantee the authenticity and integrity in addition to providing confidentiality through symmetric key encryption. Originating from insights of that combining algorithms for authentication and integrity in the wrong way would produce a less secure system [6]. Using an AEAD scheme data can be sent both encrypted as well as unencrypted. The unencrypted part of a message, for which confidentiality is not needed or even undesirable, is typically a header or other meta data. The data is however still verifiable in terms of integrity and sender authenticity.

Authenticated encryption can be implemented by using an encryption algorithm combined with a hashing algorithm. The strongest combination variant is called Encrypt-then-MAC (EtM). Using this scheme the plaintext is first encrypted. The resulting ciphertext is then run through a hashing function that also uses the key in the process. The ciphertext and the MAC is then sent to a recipient. [18]

Some encryption algorithms provide AEAD in and of themselves such as the stream cipher Acorn128, a recent winner of the CAESAR competition. [19]

2.1.9 Level of Security and broken algorithms

Each algorithm that uses a key has a set number of keys that are possible. This is called the keyspace and is in cryptography often denoted as |K|. The |K| can be calculated from the number of bits the key uses as 2^n where n is the number of bits. The algorithm is then said to provide “n-bits of security”. For example a 128-bit key gives a |K| of $2^{128} \approx 3.40 \cdot 10^{38}$.

A cryptographic algorithm is always susceptible to brute-force attacks; theoretically nothing stops an attacker from simply trying all possible keys. Since |K| is finite a brute-force attack while eventually succeed. However, in practice this is impractical since the time complexity of a brute force attack is O(N), where N = |K|. For a large |K| a complete search can require hundreds or thousands of years on current hardware. An attacker can always get lucky and guess the correct key, but the probability for that is 1/|K|, an infinitesimally small number. This has lead to attempts to find more efficient ways to search for a key. When a way to find a key that has lower time complexity than O(N) an algorithm is considered broken as the benefit of the size of |K| is no longer a guarantee of security. [15]

2.1.10 Base64 and JSON

Base64 is a standardized way to encode binary data in text form. Binary data is encoded to ASCII characters using an index table. The conversion rate is 3 bytes of binary data to 4 bytes when encoded to Base64. This gives a size overhead of about 33% but varies since data needs to be padded to multiples of 3. [20]

Java Script Object Notation (JSON) is a standard data format for data interchange. It is text based and uses key-value pairs to structure data. JSON supports arrays and nested objects. A limitation in JSON is that it does not support binary data; to overcome this Base64 can be used to first encode binary to text. [21]

2.2 Related Work

Several studies have presented designs of wireless sensor networks [22-24]. While often very informative about many challenges that face such systems, it seems like no study has explicitly evaluated how different approaches to these problems can affect energy use. A contributing factor to this is likely that the sensor networks presented have generally been deployed in places such as metropolitan areas or in agriculture where limited energy has not been a factor. A common protocol used is the ZigBee protocol [25]. This protocol uses AES to provide authentication, integrity and encryption for messages. However, the ZigBee protocol is designed for a mesh network of devices and is specified to distances of 10-100 m. All communication has to go through a central gateway to reach the Internet. While this protocol does provide good security it is not applicable to long distance communication making it unsuitable in the context of this study.

Energy use is often identified as a constraint in IoT, as many devices are expected to run for a long time on limited power. Trappe et al. discusses this in the 2015 paper “Low energy security”. Moreover, they present a direct energy measurement of a sensor node framing a data packet and sending it over radio using a Texas Instrument MSP440g2553. From this data can be seen that radio communication is the major source of power use; about 75% of all energy needed is used to send a message. They do not, however, use any security processing in this practical test. From their findings they draw the conclusion that conventional security such as SSL is infeasible on lower end IoT devices since they are simply too constrained and goes on to explore other ways to achieve security for instance authentication using location and radio wave signatures and dedicated encryption hardware [26].

Cryptographic primitives designed for constrained devices has also been proposed as a solution to security in IoT as well as protocols specifically targeting use on constrained devices [27]. A recent cryptography competition, CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness, aimed to spawn authenticated encryption algorithms and specified resource constrained devices as a target use case scenario for algorithms submitted [19]. The competition concluded in March 2019 and resulted in two encryption schemes, Ascon128 and ACORN128, deemed suitable for use in embedded applications by the committee. These algorithms have also been further investigated and proven to be strong [28].

From what is found in the literature no one seems to have investigated how different security implementations affect energy use in a live setting. The findings from this study can be used to provide insight in how software implemented security affects energy use in MCU based applications.

3 Methodology

To investigate how different types of security implementations would perform on the target hardware an experimental approach was taken. After selecting algorithms to be included in the study and how to combine them these were scored on security level and the performance on the hardware was investigated.

3.1 Approach

An important consideration when approaching the problem was that all data collected should be representative of a production environment. To achieve this all performance tests were run on the target hardware. Furthermore, to make the results as relevant as possible, performance tests used simulated data that matched the structure and size range of the expected data to be processed in production.

3.2 Target hardware

The study used the ATmega1284P microcontroller [29]. This device was chosen as it has been selected to be deployed in the field by the research group at Umeå University. It is part of the Atmel AVR family of microcontrollers and has an 8-bit processor. Onboard are three types of storage:

- 128 kb of Flash memory used to store program code
- 16 kb SRAM used as volatile storage when running a program
- 4 kb EEPROM which is non-volatile memory used for persisting data

Furthermore, it has two USART[1] that are used for serial communication and uploading of program code. The device operates between 1.8-5.5 volts. The development board used was the SODAQ Mbili [30]. This board runs the MCU on 8 Mhz at 3.3 V. During development the board was connected via USB for power and communication. The voltage of the USB port (5 V) was transformed to 3.3 V by a voltage regulator on the Mbili board.

3.3 Choice of algorithms

The first step was to choose which algorithms were to be included. One criteria was that the algorithms needed to fulfill Kerckhoff's principle: an algorithm has to be secure even if the implementation details are known as long as the key is kept secret [3]. Failure to follow this principle has in the past proven to lead to weak security [31]. In the same vein, an algorithm should also not be considered broken; the definition of this concept being that a way to attack with lower cost than brute force has been found [32]. There are many algorithms to choose from and the selected ones were chosen because they met the requirements and implementations suitable for the hardware were available. Whether an algorithm was of block or stream type did not affect the selection process. Only symmetric key algorithms were included and thus public key algorithms were excluded from the study as key sharing

1 Universal synchronous and asynchronous receiver-transmitter

and other concomitant security issues that a public key algorithm would solve, was not considered severe enough threats to warrant the added complexity. When sharing a key in production, the key will be stored securely on the device ROM during the initial configuration.

To achieve the security goals, algorithms need to be combined to form a complete security solution. There are several ways to do this but three main variants where chosen due to time limitations. First, a hashing with HMAC scheme was chosen (Figure 4). This would provide authentication and message integrity. It was also hypothesized that this minimalistic approach would minimize resource use.

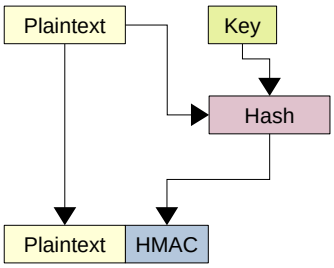

Figure 4: Using HMAC provides authentication and validation of integrity.

The second scheme chosen was to combine an encryption algorithm with key-based hashing to achieve a security solution with authenticated encryption. There are several ways to combine the algorithms but the one chosen was Encrypt-then-MAC (EtM) (Figure 5). This is the recommended way to combine algorithms for authenticated encryption [18].

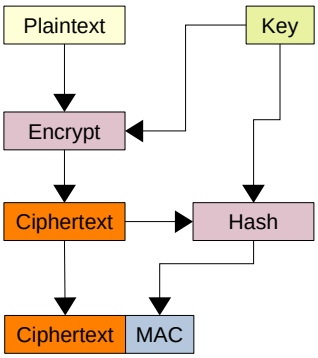

Figure 5: The Encrypt-then-MAC scheme for combining encryption with hashing to achieve authenticated encryption.

As the final variant some of the algorithms providing AEAD capability were chosen. Besides authenticated encryption, using additional data some parts of the message can be left as plaintext but still benefit from authentication and implicitly message integrity [18]. There is also the choice to forgo encryption altogether but still use authentication. For these algorithms no combining was necessary and they could be used directly (Figure 6).

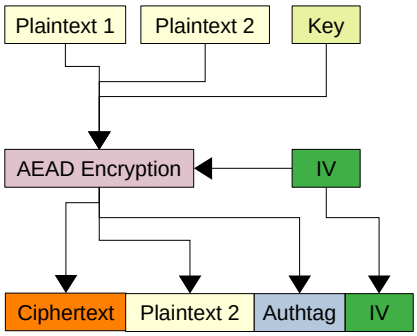

Figure 6: Using an AEAD capable algorithm gives an out-of-the-box solution.

These three patterns will be referred to as HMAC, EtM and AEAD.

The chosen algorithms are listed in Table 1. A listing of all algorithms with detailed information is listed in Appendix C. For implementations the Arduino Cryptography Library was used [33]. This library provides a good interface to the algorithms making the construction easier.

Table 1: List of selected algorithms.

Name	Type	Pattern
SHA-256, SHA-512, SHA3-256, SHA3-512	Hash	HMAC
AES-128, AES-256	Block cipher (ECB)	EtM (combined with HMAC)
EAX, GCM	Block cipher mode	AEAD
Acorn128	Stream cipher	AEAD

3.4 Test input data

As input data to the algorithms random sensor readings were generated. The generated readings matched the stipulated format specification of sensor readings developed in collaboration with the research group for which the system is being created. Sensor readings were formatted in JSON in minified format i.e. excluding all unnecessary white space, and contains a header with metadata for the reading and an array of data. The JSON string was created using the mature and widely used library ArduinoJson (ver. 6.10.0) [34]. The generated data simulated readings from a temperature chain sensor, a type of sensor that is being developed in-house and simultaneously reads temperatures from several thermometers positioned vertically at different depths in a water body. The number of thermometers used by a sensor will vary with the water depth at the deployment site. Simulating temperature chains of various length presents the opportunity to easily produce input data of different lengths while still being realistic. An example of a sensor reading is presented in Appendix A with additional information regarding the structure. During testing chain lengths with 1, 3, 5, 10, 15 and 25 thermometers were used. Generating input data

using these lengths encompasses the estimated full size range of sensor readings expected to be present in production.

3.5 Scoring level of security

For symmetric key and hash algorithms the key length and digest size respectively is used as the measurement of level of security. The respective values for each algorithm is of a predetermined length; common lengths are 128, 256 and 512 bits. From this follows that for a length *n* the number of possible permutations is 2^n. An algorithm is then said to have *n*-bits of security. [35]

For each tested algorithm and algorithm combination the number of bits of security was noted. As a trade-off for simplicity, algorithms combined in the EtM-form were scored as the highest bit count in the combination.

3.6 Estimating power consumption

In order to study power use in low power devices the preferred approach is direct measurements [26]. However, since neither the expertise nor equipment for making direct measurements of power use were available the choice was made to use processing time as a proxy. Derived from Ohm's law, equation 2 shows that the electrical energy (E) used by a system is dependent upon by three factors: voltage (V), current (I), and time (t).

$$E = V \cdot I \cdot t \tag{2}$$

Assuming that the voltage and the processor current draw is constant during all tested security algorithms, the time spent performing an operation is correlated to the energy consumed. This approach does not give a direct measurement of the actual energy used but it does enable relative comparison of the different algorithms.

Time measurements were made on the ATmega 1284P by using the Arduino SDK function `micros()`[1]. This functions returns the current time from power on of the unit in µs. A limitation in this method is that the MCU has a time resolution of 4 µs at 8 MHz.

To keep with the intent of creating a representative experimental setup, time measurements where made of the complete process of creating a message from a sensor reading. A message was formatted as a JSON with the key: "payload", containing the data and additional keys for data needed for the encryption process. All binary data was encoded in Base64; the time for this is also included in the measured time. The process is outlined in Figure 7. The structure of a message is detailed together with examples in Appendix B.

1 https://www.arduino.cc/reference/en/language/functions/time/micros/

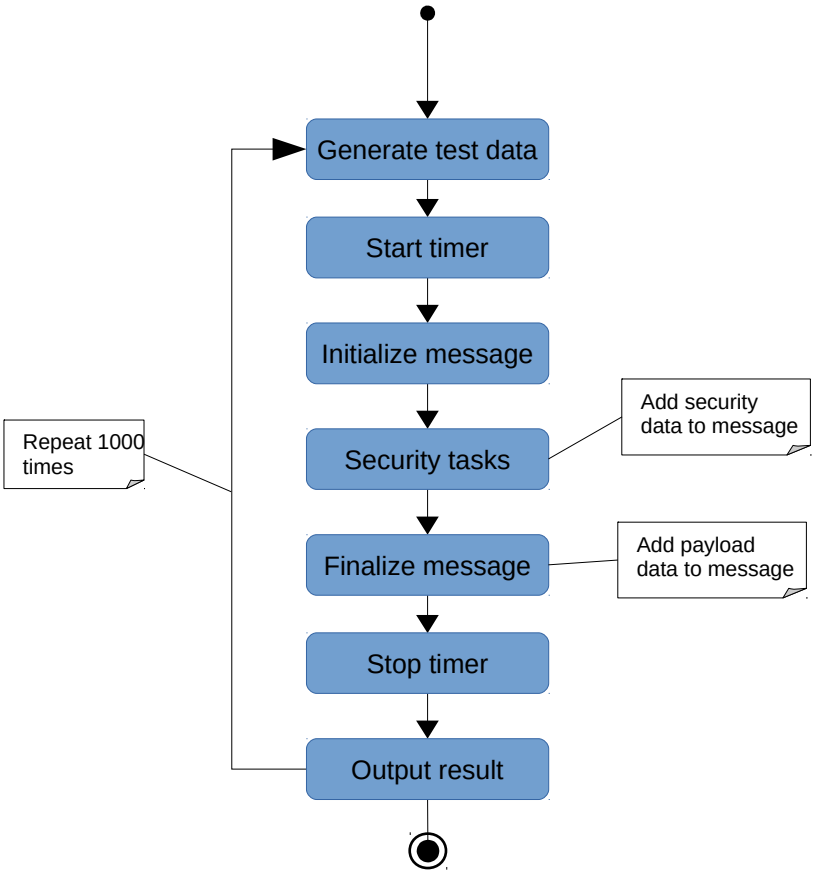

Figure 7: Activity chart for the time measuring process.

The baseline is a setup where no security is used, essentially skipping step “Security tasks”. In the baseline no Base64 encoding would be necessary and the sensor reading JSON would be sent as is. This means that the baseline for security is zero. The generation of simulated data was not part of the timing process since in production this would be accomplished by reading a real sensor. Energy used by a real sensor is dependent upon which type of sensor is being read, thus timing the creation of test data is not relevant to this study.

When measuring performance data each test case was run 1000 times. There intention with this high number was two-fold: to catch any memory leaks that would build up over longer deployments; 1000 iterations was estimated to be equivalent to about 2 months of field use, as well as stressing the processor to reveal any potential problems from e.g. overheating. The tests were performed at sea level and in room temperature, approximately 23°C.

3.7 Program size footprint

One aspect not related to energy use is how much program space each implementation requires. When compiling each implementation the total size of the program code uploaded to the flash memory of the device was recorded. By subtracting the size of a void security implementation, i.e. no security was present, an estimate of the contribution of each security implementation could be deduced. Note that the size is that of the complete solution

including any support libraries needed, e.g. Base64. The ArduinoJson library was present in all implementations including the baseline since in that case it would still be used for sensor readings even in a scenario without any security. Changing compiler settings was not investigated; the default for the PlatformIO environment was used.

3.8 Implementation

The first step was to establish the requirements of the experimental program. Once the requirements were established the process to implement a solution used a workflow based on agile development strategies.

To measure all necessary parameters empirically a test suite was created.

3.8.1 Tools used

All code was written in the IDE Microsoft Visual Studio Code[1] using the plug-in PlatformIO[2]. This combination was chosen since it gives easy installation of third party libraries needed for the implementation from a central repoistory. It also provides easy upload of compiled code to the MCU and a terminal for serial communication with the MCU. Furthermore it has integrated support for the Arduino SDK framework, which is used by the research group for other projects. This allowed a seamless integration into their workflow. Other development environments considered were the Eclipse IDE[3] with corresponding Arduino plug-in or the native Arduino IDE[4] but these were rejected due to lacking functionality and compatibility with other software used by the team. Visual Studio Code also provides integrated support for Git, which was used for version control. No other version control software was considered. For algorithm implementations the Arduino Cryptography Library was used since this was deemed the most mature and complete library available [33]. Output from the test runs was organized and analyzed using LibreOffice Calc spreadsheet software.

3.8.2 Functional Requirements

Functional requirements for the test suite are focused on specifying what parameters to gather. Some parameters were not gathered in code but where gathered from output from the IDE, such as compile sizes.

Functional requirements established were:

- The total time to create a message ready to be sent should be reported
- The test data should be randomized to be able to detect if there are any variation in the measurements due to what data is processed
- Keys are static and do not change during running
- Any nonce should be randomly generated

1 https://code.visualstudio.com/
2 https://platformio.org/
3 https://www.eclipse.org/ide/
4 https://www.arduino.cc

3.8.3 Non-functional Requirements

Some non-functional requirements were established. Since the only use case for the implementation is to gather performance data of the algorithms this list was short.

- The final implementation should perform the measurements when processing data structures that are realistic and of varying lengths
- It shall take little effort to change the algorithms to be tested
- The total addition to compile size for each algorithm or algorithm set should be recorded

4 Construction

4.1 Design

The test suite implementation was designed in an object-oriented style in C++11. A simplified UML diagram is presented in Figure 8. An abstract base class, `MessageCreator`, was used as the root of the inheritance tree. From this base class an abstraction layer for each algorithm group was defined as separate template classes. This provided an easy way to specify which algorithm was used. In a separate header algorithm combinations were then defined. To achieve easy selection of which security implementation to use each definition was enclosed in a preprocessor `#ifdef`–statement. This allowed a convenient way to select which algorithm to run by simply adding the corresponding `#define` in the main-file. Since the target hardware has limited memory each test case had to be separately compiled and uploaded to the device for each test run.

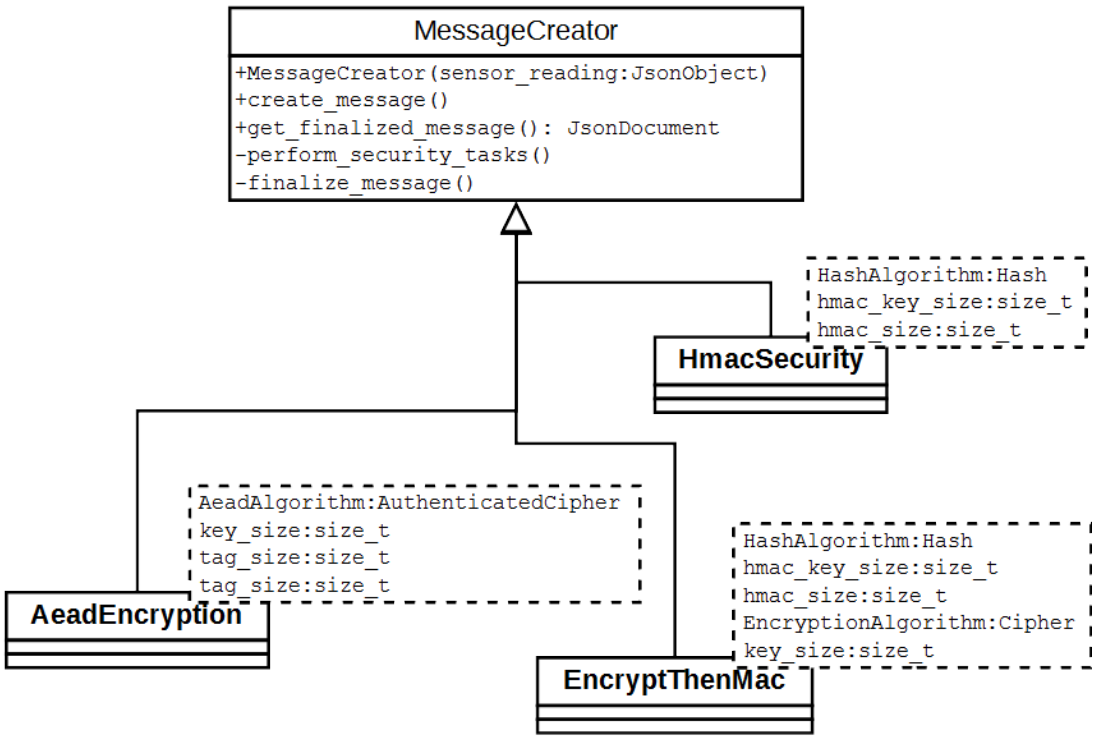

Figure 8: Simplified UML diagram of the inheritance structure. Each implementation type was abstracted as template classes. These in turn inherited from an abstract base class. Each concrete implementation of the template classes was defined in a header file.

The timing test was implemented as two nested for-loops. The outer loop iterated over the indices in an integer array where each index held the value for how long the simulated sensor chain should be. In the inner loop this value was passed to a function which generated a temperature chain sensor reading of the specified length with random values. After this the current time was recorded and the sensor reading was then used as input data

for the security function. Once the security function returned the current time was recorded again and the elapsed time could be calculated. The result for that iteration was then sent via serial communication to the development computer and where it was captured. Results were formatted to output to a CSV-file[1] structure. The inner loop ran 1000 iterations for each iteration of the outer loop.

Keys used by algorithms that needed them were drawn from a predefined array of randomly generated bytes. This meant that all keys were the same for each test run. Key lengths determined how big a portion of the byte array was used. In contrast nonce values (IV) were generated randomly using the pseudorandom Arduino SDK function `random()`[2]. The state of this was seeded by reading an analog noise value from an unconnected pin on the MCU. This was good enough for the purpose of these tests but in a production implementation this seeding would be done be for instance reading a real-time clock.

Due to the constrained environment care had to be taken not to run out of memory. As many as possible of the variables were allocated on the stack. Only variables whose size could not possibly be known beforehand were allocated on the heap. Great care was taken that all dynamically allocated memory was cleared. Here the constrained nature actually played in the favor of development. Since the memory space for dynamic allocation was so small any failures to correctly handle memory was very quickly discovered after just a few iterations. When the device fails to allocate memory it reboots and starts the program from the top again and never finishes.

To indicate a completed test run an LED on the board was programmed to blink when all test iterations had been completed. This ensured that a test was successful and program flow had reached the end.

4.2 Testing

During development the code was continuously tested. In order to verify the correctness of the implementation of the algorithms various online tools were used. By generating test vectors it was possible to test the implementation to be sure that all generated data was correct. [36-39]

1 comma separated value
2 https://www.arduino.cc/reference/en/language/functions/random-numbers/random/

5 Results

5.1 Energy consumption

5.1.1 Message processing cost

In order to minimize energy use the time to create a message from a sensor reading was measured. This was done by timing the different algorithms when processing simulated input data of increasing lengths. Input sizes were approximately 66, 98, 131, 212, 298, 470 bytes. For each combination of algorithm and input size 1000 test timings were performed. Simulated data was generated for each iteration. The average time per byte was then calculated over for each input size. These results were then used to calculate an average and standard deviation as seen in Figure 9. The AEAD group (blue) performed fastest in general with little impact of increasing input size. In comparison the HMAC group (red) showed a large variation between the different algorithms in the group. This trend is also present in the EtM group (green). The error bars in the graph indicates the variation with input size. Wide error bars indicate a large difference between processing short and long input data. For the AEAD group this variation was consistently small while vary greatly within the other two groups.

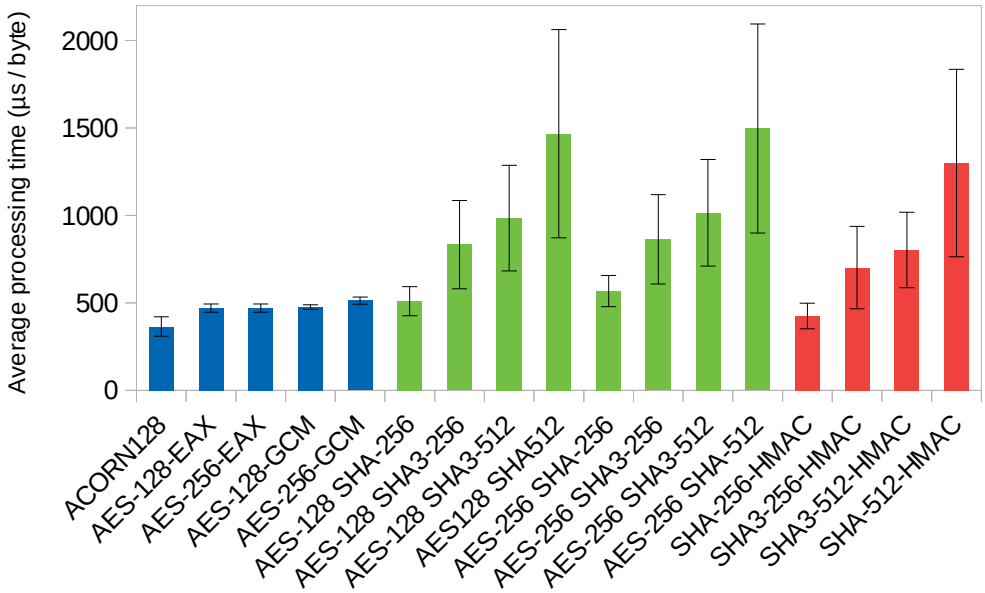

Figure 9: Average time for framing a message from sensor data of varying length. The time used to process and construct a finished message for each input length was averaged. The error bars should be interpreted as a large difference between processing a small amount of input data versus a large amount.

5.2 Cost of security level

To further investigate the energy cost of security, the level of security for each algorithm was correlated to how many bits of security an algorithm provides. Since energy use was estimated by processing time per byte in the input data (µs/byte), the cost is presented as bits of security per µs/byte. This gives insight into how well the energy spent is put into security, see Figure 10. Examining the AEAD group, which was previously shown to be the cheapest in terms of actual energy use, reveals that since all the algorithms in this group performed similarly in terms of actual energy use the higher bit versions of the algorithms are cheaper for the level of security they provide. AES-128 in EAX and GCM mode gave the lowest security level to energy usage yield of all tested algorithms. In the EtM group the higher bit count of the combined algorithms were used. This indicates that the security they provide is sometimes cheaper per bit than for members of the AEAD group. The HMAC group shows similar cost per bit, but this group does not provide the encryption present in the other two groups.

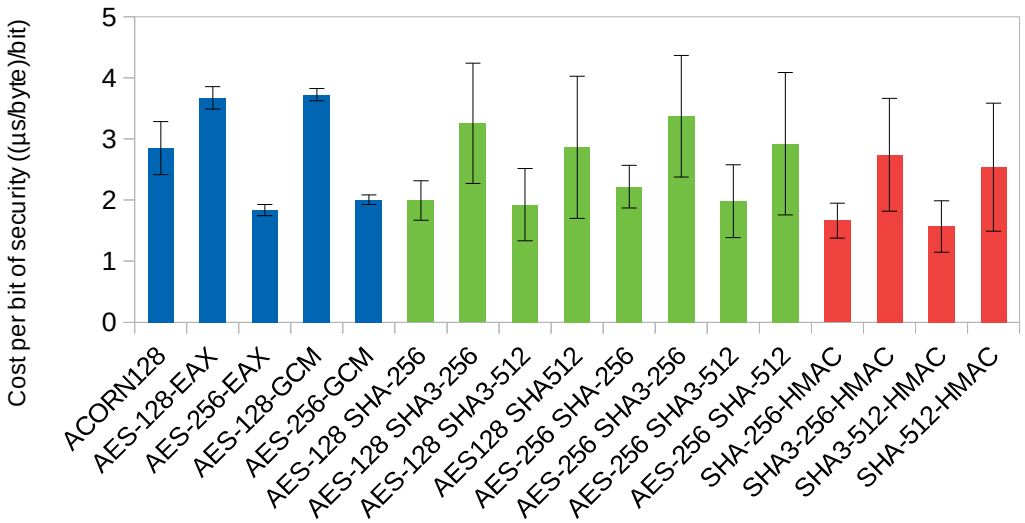

Figure 10: Energy cost per bit of security. Each algorithm has a fixed security level measured in bits. The graph shows the energy cost for each bit of security expressed as how much time each byte in the original sensor reading takes to process into a finalized message. The error bars indicate how the cost varies with input data size; larger input size gives a lower cost per bit.

5.2.1 Message length overhead

The final aspect regarding minimizing energy use was how much data overhead each algorithm adds to the final message. Sending data via radio is a significant energy cost. To investigate this the average final message length was expressed as a ratio of the average input data length for each algorithm. The data presented in Figure 11 shows that in general this is relatively similar for all algorithms. In the AEAD group all algorithms append an IV and an Authentication Tag to the message. The length of these are only slightly different between each algorithm. The trend is similar in both the EtM and HMAC groups. The smallest increase in message size was found in SHA-256 and SHA3-256 of the HMAC groups.

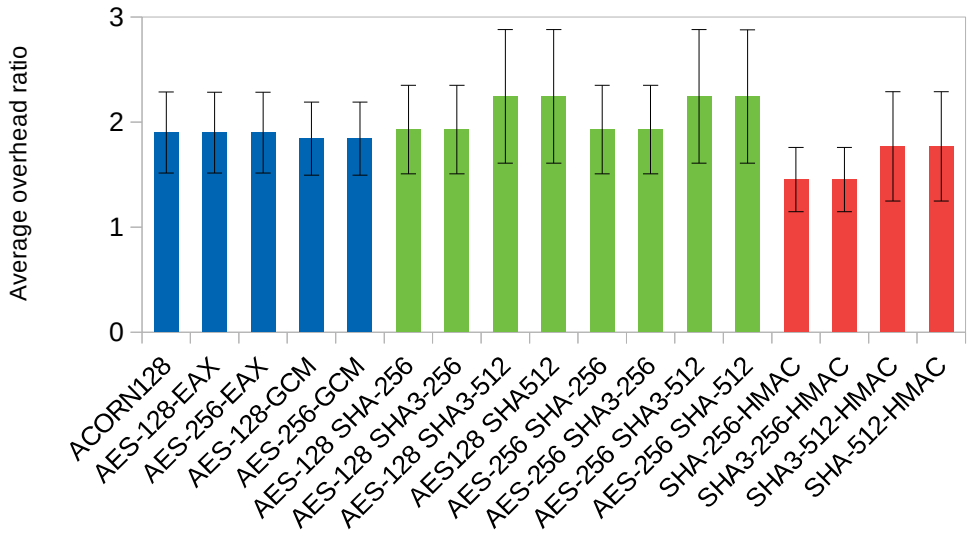

Figure 11: Average overhead ratio. When framing a sensor reading to a finalized message additional data has to be added. This data differs between algorithms both in length as well as content. The graph shows how much larger a final message is compared to the original input data on average. The error bars indicate how the ratio differs with input data size; increasing size yields a proportionally lower overhead.

5.3 Program size footprint

The distribution for the implementations is shown in Figure 12. In general all algorithms contribute from around 7500 to slightly above 10000 bytes. The noticeable outliers are implementations using SHA512. These implementations are about 6000 bytes larger than the second largest ones. A no security implementation compile size was 9408 bytes.

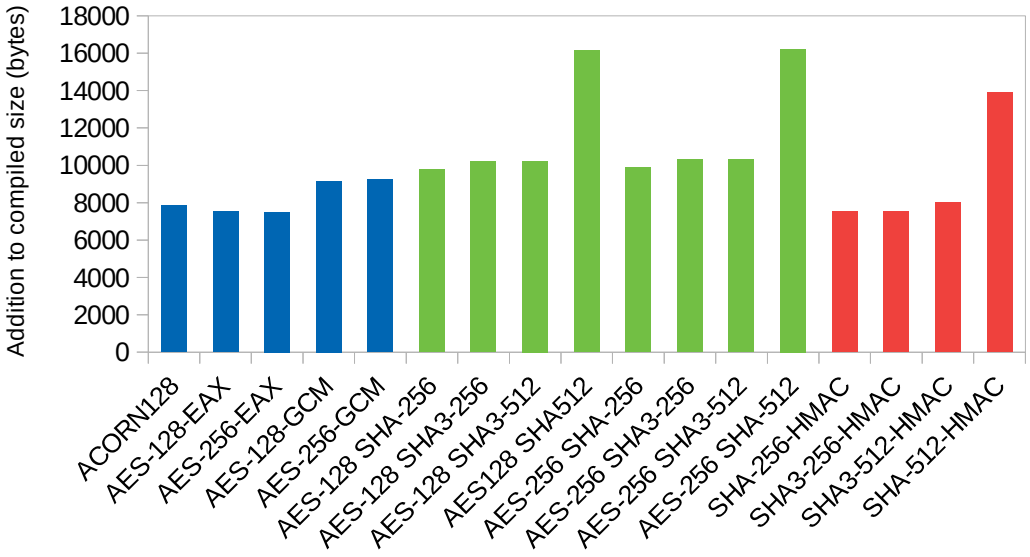

Figure 12: Security implementation contribution to total compiled size. Compiling the test suite without a security implementation yielded a compiled size of 9408 bytes.

6 Discussion

The focus of the study was to estimate energy use in a real deployment scenario with added processing for security; the total cost for adding security to a hypothetically existing solution was estimated. This focus on realism meant that factors other than hashing and encryption was included, such as the encoding of data to fit the selected transport format and the finalizing of a message up to the point where it could be sent.

Processing time was used as a proxy to compare the relative energy use of different solutions. This way to estimate energy use relies on timing the processing of similar input data for different algorithms and is a convenient way to get relative energy use data. The results from this investigation (Figure 9) showed that the AEAD group of algorithms used the least amount of energy in general. In this group ACORN128 was the algorithm that used least energy of all algorithms. In the other two groups the variation between algorithms was large. Both the EtM and HMAC groups relied on SHA and SHA3 families of algorithms. This gave both groups a similar trend. In the EtM group the addition of AES was adding only slightly to the total time when compared to HMAC hashing. A surprising discovery was that solutions using SHA3-512 were faster compared to using SHA-512. The reverse was found for the 256 bit versions of the algorithms. According to literature, SHA3 can be implemented using different bitrates [40], trading speed for security. This gives a plausible explanation; the discrepancy has likely to do with how the algorithm is implemented in the used library. However, investigation of how the algorithms were implemented was outside the scope of this study.

The strength of a cryptographic function is correlated to the number of bits involved in the process. For an encryption algorithm it is the key size, and for a hash function the size of the digest. To make a fair comparison for different algorithms the study approached this by estimating the energy cost per bit of security. The results (Figure 10) showed that while algorithms with a lower bit security was cheaper in actual power use, many of the higher bit algorithms were cheaper per bit of security. In the AEAD group the actual power use was very close to equal for all included algorithms, but since two of the algorithms had twice the number of bits the cost per bit for these were halved. Interestingly, from this perspective ACORN128, which was developed with the explicit intent of being used by constrained devices is not the cheapest due to its lower security bit count. The EtM and HMAC groups showed a shared pattern since they were both based on the same underlying algorithms. HMAC had as slightly lower cost per bit than the corresponding EtM implementations. Again, the SHA3-512 algorithm based solutions stood out for the same reason as before.

It has been shown that a significant part of energy use in wireless remote sensing is the power consumption of the radio device [26]. Minimizing bytes that need to be sent is thus of interest when trying to reduce energy use. Each solution in this study produces different data overhead that needs to be appended to a message for a recipient to be able to decode the message. For this reason, an average ratio of the message length in relation to the sensor reading was calculated. The results (Figure 11) showed that the length of a final message was about two times larger that the input size for most algorithms. Algorithms with a 512 bit output was adding slightly more to the overhead. The bulk of the increase in size comes from the inflation of data size when converting the binary data to Base64; a process

which increases the size in the order of 3 to 4, or roughly 1.33. This was done because the chosen transport format (JSON) cannot handle binary data very well. The smallest increase in size was observed in the HMAC solutions. HMAC algorithms provide no encryption meaning the plaintext of the sensor reading is packaged directly together with a Base64 encoded HMAC in the final message; very little data is actually inflated in these cases. Using other transport formats not requiring encoding into Base64 and instead sending the bytes as is would yield a great reduction of message size and is of great interest for the future.

The final aspect of concern was how adding security affected the size of the compiled program. Since the main purpose of the device is to perform measurements through various sensors, the code for controlling these must be included. Presently, the size of this is not known but can be expected to take a significant part of the available space. When adding a security implementation, the addition seems to be in the order of 10 kb (Figure 12). However, for this parameter a significant outlier was SHA512-based solutions which were about 6000kb larger. Some investigation into this did not produce a clear explanation, however SHA algorithms use standardized initial values that need to be stored. For the 512 bit version this byte array is larger but does not fully explain the observed results.

Before this study the suggested algorithm to use was a hashing algorithm with HMAC. The results however contradict this initial inclination since AEAD type algorithms are shown to provide a better solution in several regards. Firstly they match or outperform the HMAC algorithms in terms of energy use. Secondly they are more flexible; they can provide both authenticated encryption but also be used without encryption while still providing authentication and integrity of the data. This would lower the message overhead and also keep the data in a human readable form, a desired feature expressed by some members of the research team. Since encryption was not a required feature this would still fulfill the requirements while keeping the door open for encryption should the need arise in the future. Based on these arguments the suggested algorithm to use at this date is AES-256-GCM. This will give the application a good security solution at a low energy cost. The goal of the larger monitoring application is a complete infrastructure with a backend in addition to the remote sensing devices. The remote sensing devices are developed with the intent to be able to easily implement different types of sensors in the future via a common interface.

Initially when starting this project the expectations were that the constrained nature of the target device would make useful implementations hard. However, this turned out not to be the case since these types of devices, while still being constrained, are very capable. The total test suite program use only about 5-7 % of the total program space available on the ATmega 1284P. This gives ample space for implementing the main features of controlling sensors.

The main weakness of this study is that energy use was not measured directly. Actual power consumption can be estimated by using the current rating from the MCU specification. However, many variables affect the actual power use. Lowering the voltage would theoretically use less power, but doing so would also mean that the clock speed of the processor would have to be lowered. Lower clock speed means that an operation takes more time thus increasing the theoretical power use.

Overall, the findings of this study show that energy use has a relatively small impact on the total energy consumption. As was previously shown, radio communication is the major source of energy use for communicating data from a remote sensor [26]. For long time deployments lasting several months, other ways of reducing energy use are also of interest, such as using low power sleep mode for periods between sensor readings. In this mode the MCU minimizes energy consumption by essentially turning itself off, only running the internal clock to trigger a wake up interrupt after a set amount of time [29]. Total deployment time is also dependent upon how large the used battery is. An obvious way to extend deployment time is to use larger batteries. This would require no modifications to the other hardware and is only limited by how economical and practical a given battery size is. Solar panels are also a way to extend deployment time if the seasonal timing and deployment site allows it.

In conclusion, the results show that a good level of security is possible and practical on the target hardware.

7 Conclusions

A good level of security is possible and practical on the target hardware and a method that can be readily used in the environmental monitoring project has been developed.

The suggested security implementation is using AES-256-GCM. This will give a security solution expected to be usable in the foreseeable future. The flexible nature of this solution will provide authentication and integrity while also incorporating an optional encryption of data if required. AES-256-GCM is also a common cryptographic primitive and is expected to be available for use when implementing the server backend.

Where ethical concerns can be become a problem is in the larger context where the method of this study is used, such as what types of sensors are used and where they are deployed. How collected data is stored and used may then also be subject to ethical concerns depending on the sensitivity of the data. As the currently intended purpose is to collect environmental data, such as temperature and wind speeds, ethical and privacy concerns are expected to be irrelevant. No image or sound capture is planned.

References

[1] P. Rodríguez, et al., "Do warming and humic river runoff alter the metabolic balance of lake ecosystems?." *Aquatic sciences* 78.4, pp. 717-725. 2016

[2] T. M. Tukade, R. M. Banakar, “Data Transfer Protocols in IoT-An Overview”, I*nternational Journal of Pure and Applied Mathematics*, Volume 118, no. 16, pp. 121-138, 2018.

[3] A. Kerckhoffs, "La cryptographie militaire." *Journal des sciences militaires* 9, 538, 1883.

[4] M. A. Khan, K. Salah,”IoT security: Review, blockchain solutions, and open challenges”, Future Generation Computer Systems, no. 82, 2018, pp. 395–411.

[5] A. Perrig, et al., "SPINS: Security protocols for sensor networks." *Wireless networks,* 8.5 pp. 521-534, 2002

[6] M. Bellare, P. Rogaway and D. Wagner, "A conventional authenticated-encryption mode." *manuscript*, April 2003. [Online]
Available: https://eprint.iacr.org/2003/069
[Accessed: 2019-04-26]

[7] D. Eastlake III and T. Hansen, "RFC 4634-US Secure Hash Algorithms (SHA and HMAC-SHA)." Motorola Labs and AT &T Labs, 2006. [Online]
Available: https://tools.ietf.org/html/rfc4634
[Accessed: 2019-04-26]

[8] M. J. Dworkin, SHA-3 standard: Permutation-based hash and extendable-output functions. No. Federal Inf. Process. Stds.(NIST FIPS)-202. 2015.

[9] R. Rivest, "The MD5 message-digest algorithm (rfc 1321)." Internet Activities Board (1992). [Online]
Availabile: https://www.ietf.org/rfc/rfc1321.txt
[Accessed: 2019-04-26]

[10] D. Eastlake and P. Jones. "RFC 3174: US secure hash algorithm 1 (SHA1).",Network Working Group, 2001 [Online]
Available: https://tools.ietf.org/html/rfc3174
[Accessed: 2019-04-26]

[11] B. Cohen, "BEP0003-The BitTorrent Protocol Specification.", 2008 [Online]
Available: http://www. bittorrent. org/beps/bep_0003. html
[Accessed: 2019-04-26]

[12] P. Hawkes, M. Paddon and G. G. Rose. "Musings on the Wang et al. MD5 Collision." IACR Cryptology ePrint Archive 2004 (2004): 264.

[13] R. Shirey, "RFC 4949-Internet Security Glossary.", 2007. [Online]
Availabile: https://tools.ietf.org/html/rfc4949
[Accessed: 2019-04-22]

[14] M. Bellare, C. Ran and H. Krawczyk. "Keying hash functions for message authentication." Annual international cryptology conference. Springer, Berlin, Heidelberg, 1996.

[15] C. Paar and J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[16] Wikipedia, Block cipher mode of operation [Image]
https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation
[Accessed: 2019-04-25]

[17] M. J. Dworkin, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC" NIST. No. Special Publication (NIST SP)-800-38D. 2007.

[18] M. Bellare and C. Namprempre, "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm", 2007. [PDF]
Available: https://cseweb.ucsd.edu/~mihir/papers/oem.pdf
[Accessed: 2019-04-26]

[19] CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness [Online]
Available: https://competitions.cr.yp.to/index.html
[Accessed: 2019-04-06]

[20] S. Josefsson, "The base16, base32, and base64 data encodings", No. RFC 4648. 2006.

[21] Bray, Tim. The javascript object notation (json) data interchange format. No. RFC 8259. 2017.

[22] Jiang, Peng, et al. "Design of a water environment monitoring system based on wireless sensor networks." Sensors 9.8 pp. 6411-6434., 2009.

[23] Khedo, Kavi K., Rajiv Perseedoss, and Avinash Mungur. "A wireless sensor network air pollution monitoring system." arXiv preprint arXiv:1005.1737 (2010).

[24] A. Dorri, et al., "Blockchain for IoT security and privacy: The case study of a smart home." IEEE international conference on pervasive computing and communications workshops (PerCom workshops). IEEE, 2017.

[25] A. F. Molisch, et al. "IEEE 802.15. 4a channel model-final report." IEEE P802 15.04 0662, 2004.

[26] W. Trappe, R. Howard, and R. S. Moore. "Low-energy security: Limits and opportunities in the internet of things." IEEE Security & Privacy 13.1 pp. 14-21. 2015

[27] Z. Shelby et al. "RFC 7252: The constrained application protocol (CoAP)." Internet Engineering Task Force (2014). [Online]
Availabile: https://tools.ietf.org/html/rfc7252
[Accessed: 2019-04-26]

[28] D. A. Dhar, et al. "SAT-based Cryptanalysis of Authenticated Ciphers from the CAESAR Competition.", IACR Cryptology ePrint Archive", 2016 (2016): 1053. [PDF]

Available: https://eprint.iacr.org/2016/1053.pdf
[Accessed: 2019-04-26]

[29] ATmega164A/PA/324A/PA/644A/PA/1284/P megaAVR® Data Sheet [PDF]
Available: http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega164A_PA-324A_PA-644A_PA-1284_P_Data-Sheet-40002070A.pdf
[Accessed: 2109-04-26]

[30] SODAQ Mbili 1284P [Online]
https://support.sodaq.com/sodaq-one/sodaq-mbili-1284p/
[Accessed: 2019-05-23]

[31] K. Wirt, "Fault attack on the DVB common scrambling algorithm." International conference on computational science and its applications. Springer, Berlin, Heidelberg, 2005.

[32] J. Siegfried et al., "Examining the encryption threat.", International Journal of Digital Evidence, Volume 2:3, 2004.

[33] R. Weatherley, Arduino Cryptography Library [Software Library]
https://github.com/rweather/arduinolibs

[34] B. Blanchon, ArduinoJson [Software library]
Available: https://arduinojson.org/
[Accessed: 2019-04-20]

[35] A. K. Lenstra, “Key Lengths”, (Contribution to The Handbook of Information Security, 2007) [PDF]
Availabile: https://infoscience.epfl.ch/record/164539/files/NPDF-32.pdf
[Accessed: 2019-04-26]

[36] Online HMAC Generator [Online]
https://www.liavaag.org/English/SHA-Generator/HMAC/
[Accessed: 2019-05-03]

[37] AES calculator [Online]
http://extranet.cryptomathic.com/aescalc/index
[Accessed: 2019-05-03]

[38] Decode from Base64 format [Online]
https://www.base64decode.org/
[Accessed: 2019-05-03]

[39] Hex To String Converter[Online]
http://string-functions.com/hex-string.aspx
[Accessed: 2019-05-03]

[40] G. Bertoni et al., "Keccak implementation overview.", 2012. [PDF]
Available: http://keccak. neokeon. org/Keccak-implementation-3.2. pdf

Appendix A

Example of a sensor reading


```
{
    "t": 1555666777,
    "id": "Logger1",
    "s_t": "tc",
    "s_d": [
        {
            "d": 0,
            "t": 12.5
        },
        {
            "d": 1,
            "t": 12.3
        },
        {
            "d": 2,
            "t": 12
        },
        {
            "d": 3,
            "t": 5.8
        },
        {
            "d": 4,
            "t": 6.5
        },
        {
            "d": 5,
            "t": 8.7
        }
    ]
}
```


This example shows an artificially generated reading from a temperature chain sensor. The reading contains three header keys: “t”, the timecode for when the reading was done, “id”, the identity of the logger unit to which the sensor is attached and “s_t”, the sensor type, which holds a string code for the sensor type.

The “data” key holds an array of sensor reading values. The contents of this array will vary with what type of sensor is used. In this case it is an array depth-temperature objects, one for each thermometer in the chain. Each depth-temperature object has two keys: “d”, the depth in meters and “t”, the temperature in degrees Celsius.

In the application the string is stripped of all white space, but is present here for clarity.

Appendix B

Example of a message.


```
{
    "iv": "JeUT0IrOEhOcXGnX",
    "tag": "GhginGu1Y3ehl7laE0zlrQ==",
    "data":
"veS23b903kWjI+IClJ4O5fU4ZBLZ9YcWtGhnVOUPn87LjRG7ItZxyBqtiltHxKG4mLt
7Wj9ERirXxa8PK3LgE4i0ZsBqFZRwpkJiv38ATUxXjwcaMkLyCGi661SYLxPMoLBbgal
XxBaoywqtkQJRrJNEcP9Vf+b4LOddDO9QRmFz1FH06CIm7M9SP5ZUPeFvBbzrS4UZEvD
Z0GxvhembKTj0PyHA+NxXs4klV4iC9q7VJDoDX0DSFWTIVrHQQICdML+ozOk/
uxEYgaLXhpTY2RLqd0AA"
}
```


The sample shows the formatted message of a SHA-256-GCM encrypted sensor reading. Three keys are present: iv, tag and data. The iv tag holds the Base64 nonce used during encryption. The tag key holds the Bse64 encoded authentication tag for the sensor reading. The data key holds the encrypted sensor reading encoded in Base64.

In the application the string is stripped of all white space, but is present here for clarity.

Appendix C

List of algorithms used. Hash algorithms support any key size but internally either hash the key first if it is too long or pad it to the correct length. Block ciphers use ECB mode in EtM implementations. All combinations of block ciphers with HMAC hash algorithms were investigated. EtM used separate keys for the encryption and HMAC.

Pattern	Type	Algorithm	Security level (bits)	Block size / Bitrate[1] (bits)	Key size (bits)	Digest / Tag size[2] (bits)	IV size (bits)
HMAC	Hash	SHA-256	256	256	256	256	n/a
HMAC	Hash	SHA-512	512	512	128	512	n/a
HMAC	Hash	SHA3-256	256	1088	1088	256	n/a
HMAC	Hash	SHA3-512	512	576	576	512	n/a
EtM	Block cipher	AES-128	128	128	128	n/a	n/a
EtM	Block cipher	AES-256	256	256	256	n/a	n/a
AEAD	Stream cipher	Acorn128	128	n/a	128	128	128
AEAD	Block cipher mode	GCM	Depends on underlying block cipher			128	96
AEAD	Block cipher mode	EAX	Depends on underlying block cipher			128	128

1 For SHA3
2 For AEAD algorithms

Acknowledgments

My sincere thanks to Marie Lindgren, Ph.D. for many hours proofreading. Erik Geibrink, for (unintentionally) spawning the idea for the subject in the first place and giving input from the electronics and hardware perspective. Peter Näs and Alice Hedmark for good and helpful peer reviews.