Efficient Traffic Monitoring in IoT Networks for Attack Detection at the Edge

Engla Jansson
Abstract

The Internet of Things has rapidly expanded over the years, and with this comes significant security risks, with attacks increasing at an alarming rate. A way to detect attacks in the network is by having each device send traffic monitoring information to an edge device that can investigate if any devices have been exploited. Transmitting data in such a manner introduces additional data processing and network resources.

In this thesis, I show that using a general-purpose compression module is a more efficient method for compressing the traffic monitoring information data to create a more efficient data transmission in the network than using a special-purpose compression module. In the investigation, I use the Arithmetic Coding algorithm for the general-purpose compression module and Concise Binary Object Representation encoding for the special-purpose compression module.

This is done by comparing the two methods on artificially generated traffic monitoring data. The evaluation is based on the metrics of compression ratio, execution times, and storage requirements. In all metrics, the general-purpose compression module provided better results than the special-purpose compression module, making it the preferable method.
Acknowledgement

This Bachelor’s project was conducted at the Connected Intelligence Unit at the Research Institute of Sweden in Kista, Sweden. I want to thank my advisors at RISE, Nicolas Tsiftes and Niclas Finne, for all of the help they have provided, both regarding the scientific aspects of the project, and the implementation issues. I also want to thank them for creating an enjoyable and educational experience at RISE. I want to give additional thanks to my supervisor at Uppsala University, Thiemo Voigt, for giving valuable feedback on the report, and to my examiner, Johannes Borgström.
Contents

1 Introduction 2
  1.1 Problem Statement ........................................ 2
  1.2 Approach and Scope .................................... 3
  1.3 Main Contributions ..................................... 3
  1.4 Document Outline ...................................... 3

2 Background 5
  2.1 IoT Networks ............................................. 5
  2.2 The Contiki Operating System ............................ 5
  2.3 Cooja .................................................. 6
  2.4 TCP and IPv6 ............................................ 6
  2.5 Data Compression ........................................ 6
  2.6 Related work ............................................. 11

3 Design and Implementation 15
  3.1 Integrating New Software into Contiki-NG ................ 15
  3.2 TCP Connection and Modification ........................ 15
  3.3 Data Structure .......................................... 16
  3.4 Adapting Existing Compressors for Sensor Nodes ......... 17
  3.5 Implementing a Special-Purpose Compressor for Sensor Nodes 19

4 Evaluation 24
  4.1 Experiment Setup ....................................... 24
  4.2 Evaluation Measures .................................... 25
  4.3 Compression Ratio ....................................... 25
  4.4 Execution times .......................................... 28
  4.5 Storage Requirements ................................... 30

5 Conclusion and Future Work 33
  5.1 Results .................................................. 33
  5.2 Future Work ............................................. 34
Chapter 1

Introduction

Internet of Things (IoT) networks are constructed of small embedded devices that are able to collect, send and act on data obtained from their environments over the Internet, without human interaction. The application of IoT is spread across many fields, including transportation, smart homes, agriculture, and healthcare, to name a few.

Environmental sensing is widely used in delicate scenarios, such as healthcare, where it can enable continuous monitoring of the patient’s health and keep track of their treatment progress by providing real-time information [1]. Another real-world example of IoT usage and environmental sensing, is forest fire detection, described by Zang et al. [2], where the use of the wireless sensor network helped gather and provide real-time data. This data, combined with the traditional prevention measures, would then aid in detecting and taking the appropriate measures to fight the forest fires.

These are just some examples of IoT applications, that provide a picture of the broad usage of IoT networks and their effect on society. IoT has rapidly expanded over the last decade and is set to continue growing, with an increase in the number of IoT devices from 8 billion in 2019 to 41 billion by 2027 [3]. While IoT has transformed various industries, it also poses significant security risks. As stated by Al-Amiedy et al. [4], statistics from monitoring aggregated network traffic data of more than 150 million devices globally, show that IoT devices currently make up roughly 33% of exploited devices, compared to only 16% in 2019. This increase in attacks is at an alarming rate and can have a substantial effect on the well-being of people, the environment and society.

1.1 Problem Statement

A major challenge involving IoT networks is the presence of resource constraints within the devices. These devices are often powered by batteries and have restricted computational resources and memory. As a result, there are limited procedures to integrate extensive security features into the network’s architecture. Machine learning is one method that has been used to analyze traffic statistics to detect attacks. The cost of conducting such analysis on each IoT device is often too high, given that machine learning typically requires power-hungry hardware.

DETONAR is one of many systems, developed to detect attacks, using machine learning [5]. The system was developed by the SPRITZ research group at Padua University and operates by inspecting the network traffic obtained through independent sniffer devices and sounding an alarm in the event of any abnormalities detected.

With a different approach compared to the DETONAR system, data is collected from the devices in the network at an edge device instead of using the sniffer approach. The edge device can then be powered by a main power source and has the capability of being equipped with a
GPU and other resources to process the data. The traffic information, sent from the devices to the edge device, may contain summaries of different types of data packets that each IoT device in a network has sent or received, including packets sent using protocols such as CoAP, IPv6, ICMPv6, RPL, TCP, and UDP. This introduces an overhead, such as additional data processing resulting in higher energy consumption and network resources required to transmit the data, which can affect the regular functionality of an IoT network. There is therefore an interest in reducing the overhead as much as possible.

1.2 Approach and Scope

This project aims to design a more lightweight transmission of data without compromising the performance of the IoT devices or attack detection. This is done by implementing and evaluating data compression modules on artificially generated traffic monitoring data, transmitted from the devices. The data is generated by filling a given C structure with either all random values or solely the integer 1 to stimulate predictable and non-predictable data. The first step is to adapt an already existing general-purpose data compression module to the Contiki-NG operating system and integrate it with the traffic monitoring module at both the IoT device (compression part) and the edge device (decompression part). The next step is to investigate methods to compress data with known content using a special-purpose data compression module. This module is then be implemented and integrated into Contiki-NG, in the same manner as the general-purpose data compression module.

The evaluation is done on simulated scenarios, using Contiki-NG’s built-in simulation tool Cooja, and the project does not involve developing a complete solution, only a prototype. The structure of the data that is sent by the nodes remains, and the focus is on how to transmit the data and not on the data itself or the attack analysis of the data. The main metrics of the evaluation are compression ratio, execution times and storage requirements.

1.3 Main Contributions

This report contains the following contributions:

- Adapting an existing general-purpose data compression module to Contiki-NG and integrating it with the traffic monitoring module at both the IoT device (compression part) and the edge device (decompression part).

- Investigating methods to compress known traffic data using a special-purpose data compression module and integrating it with the traffic monitoring module, similar to the general-purpose module.

- Evaluating the data compression modules, with compression ratio and execution times as main metrics.

1.4 Document Outline

The report is structured as follows; concepts relevant to understand the thesis, such as IoT, the Contiki-NG operating system, the simulation tool Cooja and data compression methods, are introduced in Chapter 2. The chapter is concluded with a brief overview of some related work. In Chapter 3, the data structure to store the traffic statistics is introduced, as well as the different data compression methods implemented for the data compression and design choices.
Chapter 4 contains the evaluation, which consists of several experiments with different setups. Conclusions and suggestions for future work are presented in Chapter 5.
Chapter 2

Background

In this chapter, the following concepts are covered, required for a better understanding of the thesis project: IoT networks, the ContikiOS along with the simulation tool Cooja and the concept of data compression along with some compression algorithms. Additionally, toward the end of the chapter, some related works are provided.

2.1 IoT Networks

In this thesis, I work with IoT low-power multi-hop sensor networks. What defines such networks are described in this section.

Sensor networks [6] consist of low-cost, low-power, multi-functional sensor nodes that are small in size and communicate untethered in short distances. These sensor nodes, which are comprised of sensing, data processing, and communicating components, rely on the concept of a collaborative effort of a large number of nodes.

The definition of a multi-hop network is that devices are spread out, making direct communication between some nodes and the border router unavailable. This causes the traffic within the network to be forwarded, from the source nodes to the border router, through intermediate nodes. Certain packets will therefore take several “hops” to reach the border router, making the total range of the network larger.

Sensor networks are often designed as Low power and Lossy Networks (LLNs) [7] to achieve energy efficiency and long battery life for the sensor nodes. They also have techniques for mitigating some challenges arising when using sensor networks, such as network layer routing, hop-by-hop communication, and adaptive modulation schemes to maintain network connectivity, minimize energy consumption, and improve data reliability.

2.2 The Contiki Operating System

The Contiki-NG operating system started as a fork from the original Contiki OS in November 2017. Development of the Contiki OS started as early as 2003, and became open-sourced in 2006 [8]. While the original operating system has stopped being developed, with its last update being in 2018, Contiki-NG has been updated continuously since.

Contiki-NG [9], is a software system that focuses on reliable and secure, low-power communications along with established protocols, including 6LoWPAN, IPv6, 6TiSCH, RPL, and CoAP. This renders it highly suitable for the creation of IoT applications. The operating system provides support to various toolsets and libraries, such as Cooja, that enable developers to quickly prototype and deploy IoT applications.
For this project, the Contiki-NG OS is used when operating on simulated IoT devices.

2.3 Cooja

Cooja [10] is a network simulator and emulator for wireless sensor networks that are included in the Contiki-NG operating system. It allows users to simulate and test various aspects of the behavior of a network, such as communication protocols, traffic patterns, and energy consumption, in a virtual environment. Cooja provides a graphical user interface and supports various plugins for additional features and functionality and is a useful tool for researchers, developers, and educators working with wireless sensor networks.

In this project, I use Cooja to simulate the IoT network with the altered approach of DETONAR, and varied data traces of traffic and data content to explore the performance of the data compression modules.

2.4 TCP and IPv6

Transmission Control Protocol (TCP)[11] and Internet Protocol version 6 (IPv6)[12] are two network protocols used in the Internet protocol suite (TCP/IP)[13]. IPv6 is responsible for the transmission of data packets over a network, while TCP is responsible for ensuring that these packets are delivered reliably and in the correct order. In other words, TCP sits on top of IPv6 and uses it as the underlying protocol to transmit data between two devices. IPv6 provides a unique 128-bit IP address to every device on a network, while TCP uses a 16-bit port number to identify individual applications or services on a device. Together, IPv6 and TCP enable communication between devices on a network, allowing for the transmission of data in a reliable and efficient manner.

TCP is a connection-oriented transport layer protocol that provides reliable, ordered, and error-checked delivery of data between applications on different hosts. It uses a three-way handshake process to establish a connection and provides flow control, congestion control, and re-transmission mechanisms to ensure reliable data transmission.

IPv6 is the latest version of the Internet Protocol (IP), with IPv4 being its predecessor. This protocol serves as a set of rules for the transmission of data across the internet. The adoption of a 128-bit address space enables the allocation of a virtually limitless number of distinct addresses to networked devices. Consequently, This allows for more efficient routing of data packets, enhanced security, and better support for new devices and technologies than its predecessor.

For the experimental setup of this project, I work with a small TCP client and server implementation that is integrated with the Contiki-NG OS and uses IPv6 to transmit the packets sent to the network.

2.5 Data Compression

The focus of this project is on data compression, which is the process of reducing the size of digital data to save storage space or reduce the amount of data that needs to be transmitted over a network. The latter scenario is considered in this project. Compression works by removing redundant or unnecessary information from the data, allowing it to be represented in a more compact form. The implementation of the data compression modules of this project are evaluated by the metrics of compression ratio, execution times, and storage requirements.
In this section, an overview of the existing data compression techniques along with other approaches to decreasing the data size, is given in order to better understanding their traits and implementations.

2.5.1 Lossless and Lossy Data Compression

Data compression algorithms can be either lossless or lossy, depending on whether the decompressed data is identical to the original data or not. Whereas lossless data compression algorithms retain the data’s integrity and restore it completely, lossy data compression removes the data which is not noticeable. By discarding some of the information, the compression ratios for lossy data compression algorithms can be substantially better than when using lossless data compression. Regardless of this, given that the data for this project contains important traffic monitoring information of the IoT devices, lossless data compression techniques is utilized. Furthermore, A general-purpose algorithm, which cannot depend on any prior knowledge of the input data, must be lossless [14].

2.5.2 Statistical Algorithms

Statistical algorithms perform compression on each symbol in two stages when encoding and decoding data: modeling and coding. During the modeling stage, a statistical model predicts a probability distribution of each symbol, and during the coding stage a coder computes a codeword for the symbols.

The process of statistical modeling can be achieved through three approaches, namely constant models, semi-adaptive models, and adaptive models. Constant models are predetermined and agreed upon by the encoder and decoder prior to encoding and decoding the data, whilst adaptive models are constructed by the encoder and decoder on-the-run. The benefit of the latter approach is that there is no need to pack the model with the data. With the semi-adaptive models, the data is processed once for the encoder to construct the model.

The encoder uses the statistical model to compress the symbols so that their average code length approaches the information entropy. The information entropy, denoted as $H(X)$, for a random variable $X$ of order O is determined by the formula

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i,$$

where $n$ refers to the number of symbols in the alphabet, and $p_i$ represents the probability of the $i$-th symbol in the alphabet. It is worth noting that the information entropy sets an upper limit for the encoder’s effectiveness.

**Huffman Coding**

Huffman Coding [15] is a greedy statistical coding method that gives each symbol in the alphabet a value with a length of an integral number of bits. The lengths of the assigned values are based on the frequencies of corresponding symbols. Each value has a unique bit pattern prefix, so that the Huffman encoded data can be decoded unambiguously.
Algorithm 1 Huffman Encoder

**Input:** F: A forest of trees initially consisting of one node, with a unique symbol and its weight.

**Output:** T: A Huffman tree.

1: procedure HuffmanEncoder(F)
2:   while |F| > 1 do
3:     T₁ ← FindMinimum(F);
4:     RemoveTree(F, T₁);
5:     T₂ ← FindMinimum(F);
6:     RemoveTree(F, T₂);
7:     T_new ← CreateTree();
8:     SetLeftBranch(T_new, T₁);
9:     SetRightBranch(T_new, T₂);
10:    SetWeight(T_new, Weight(T₁) + Weight(T₂));
11:   InsertTree(T_new);
12:   end while
13: return F;
14: end procedure

In Algorithm 1, we see that input, denoted as F, is a forest of trees. Initially, the forest consists of a single node, representing a unique symbol along with its weight. The algorithm enters a loop. Inside the loop, the two trees with the minimum weights from the forest are found. The two selected trees, denoted as T₁ and T₂, are removes from the forest. A new tree, denoted as T_new, is then created, where the left branch is set to T₁ and the right branch is set to T₂. The weight of the new tree is set to the sum of the weights of its branches and is then inserted to the forest, F. The loop continues until there is only one tree remaining in the forest, which is the final Huffman tree, that gets returned.

Arithmetic Coding

Arithmetic Coding, which is another statistical coding method, encodes the entire message into a single number, rather than separating the input into component symbols and replacing each with a code, such as in Huffman Coding. This number represents a subinterval of a finite-length interval in the range [0, 1], in proportion to its probability of occurring in the input data.

Algorithm 2 Arithmetic Encoder

**Input:** M: Message to be encoded

**Output:** C: Encoded message

1: procedure ArithmeticEncoder(M)
2:   low ← 0;
3:   high ← 1;
4:   while MoreSymbols(M) do
5:     symbol ← NextSymbol(M);
6:     range ← high - low;
7:     low ← low + range × prob(symbol-1);
8:     high ← low + range × prob(symbol);
9:   end while
10: return low;
11: end procedure
As shown in Algorithm 2, the algorithm encodes a message, \( M \), by narrowing a range of possible values for the encoding of each symbol, based on the symbol’s probability distribution. The initial range for the encoding is \([0,1]\), with the lower bound initialized to 0 and the upper bound initialized to 1. The algorithm then encodes each symbol in the message, until there are no more symbols left, updating the range of possible values for the encoding for each symbol based on its probability distribution. Finally, the compressed output is the resulting value of the lower bound, \( \text{low} \), of the encoding range, which should fall within the range \([0,1]\).

Note that, the pseudo code in Algorithm 2, assumes that the symbol probability model is known both by the encoder and the decoder since the same probability model is needed for the decoder to be able to reconstruct the original message from the compressed form.

### 2.5.3 Lempel-Ziv Algorithms

The Lempel-Ziv Algorithms are data compression algorithms that were introduced and developed by Lempel and Ziv in the 1970s [16]. Since then, several variants of the Lempel-Ziv Algorithms have been developed, such as LZ77, LZ78 and LZFX, with LZ77 being the original.

**LZ77**

LZ77 algorithms use a *sliding-window* technique to achieve compression. They replace repeated occurrences of data with references to a single instance of that data located earlier in the uncompressed data stream. These references, known as length-distance tokens, consist of a pair of numbers \((\text{length}, \text{distance}, \text{symbol})\) or a length-distance pair. The encoder maintains a sliding window structure to track a portion of the most recent data and identify repeated patterns.

**LZ78**

On the other hand, the LZ78 algorithms utilize a *dictionary-based* approach for encoding repeated patterns in a data stream. These algorithms work by constructing a dictionary of encountered substrings in the data stream. The reference tokens, represented as \((\text{distance}, \text{length})\), are used to replace the repeated substrings by referencing their positions within the dictionary.

**DEFLATE**

DEFLATE [17] is a lossless data compression algorithm that is widely used in various applications, including file compression formats such as ZIP and gzip. It was developed by Phil Katz and is based on a combination of LZ77’s sliding-window approach and Huffman coding for efficient encoding of the data. DEFLATE works by first using the LZ77 algorithm, which aids in reducing redundancy for encoding data more efficiently. After applying LZ77 compression, Huffman coding is applied for further compression.

Before encoding the input data, DEFLATE divides the data into blocks for compression. Each block can be compressed independently, allowing for efficient random access and partial decompression. Furthermore, a header and trailer are added to the compressed data for decompression purposes. The header contains information about the compression method and parameters used, whilst the trailer includes integrity checks, such as a cyclic redundancy check (CRC), to ensure the compressed data’s integrity.
2.5.4 Transformation Algorithms

Transformation algorithms are utilized in data compression to modify the structure of the data, rendering it more suitable for compression by other techniques, such as those described in Section 2.5.2 and Section 2.5.3. The purpose of the transformation algorithms is not to compress the data itself but rather to optimize its representation, which may increase the size of the data temporarily.

**Move-To-Front Transform**

Move-To-Front transform (MTF), originally published by B. Ryabko as a “book stack” [18], and rediscovered by J.K. Bentley et al.[19] in 1986, is a transformation algorithm that, when efficiently implemented, is fast enough that the advantages it provides justify being included as an extra step in data compression.

The main goal of MTF is to reduce the entropy of the input data. This is done by maintaining a list of symbols of the input alphabet, representing the recently used symbols in order. Each time a symbol is encountered, its corresponding index in the list of symbols is given as an output and moved to the front of the list.

The MTF transform is particularly effective when the input data contains repeated symbols, as the repeated symbols are more likely to be found towards the front of the list and therefore have lower indices. However, The algorithm is not always optimal and may result in increased complexity and compression time. This is particularly the case for large alphabets and is therefore often used in combination with other data transformation techniques and compression algorithms to achieve better compression performance.

**Burrows-Wheeler Transform**

The Burrows–Wheeler transform (BWT), invented by Michael Burrows and David Wheeler in 1994 [20], works by rearranging the characters of the input data into a new sequence, such that repeated patterns in the original data are grouped together in the new sequence.

The BWT starts by creating a matrix of all cyclic rotations of the input data, where each row of the matrix represents a different cyclic shift of the input string. If the input data is of length $N$, then the dimensions of the matrix will be $N \times N$. The rows of the matrix are then sorted in lexicographical order, and the last column of the sorted matrix is extracted as the transformed sequence along with the index of the row showing containing the original string.

One of the main benefits of using BWT is that the algorithm is reversible, meaning that the original input data can be reconstructed from the transformed sequence, without storing any extra data.

**Run-Length Encoding**

Run-Length Encoding (RLE) was patented in 1986 by Hitachi[21][22] and is a form of data compression that aims to reduce the size of data by representing consecutive repeated elements as a single instance followed by a count of repetitions. It is particularly effective when dealing with data that contains long sequences of repeated values.

The basic idea behind RLE is to replace repeated elements with a pair consisting of the element and the number of times it repeats. For example, if we have the sequence “AAAABB-BCCD”, RLE would encode it as “4A3B2C1D”, indicating that “A” repeats four times, “B” repeats three times, “C” repeats two times, and “D” appears once. However, For data with little repetition, RLE may not provide significant compression gains and can even result in increased storage requirements.
RLE is a lossless compression technique, meaning that the original data can be perfectly reconstructed from the compressed representation. It is simple to implement and computationally efficient, making it suitable for scenarios with limited computational resources. The technique can be applied to various types of data, including text, images, and binary data, and is often used as a component in more advanced compression algorithms or as a simple compression method in scenarios where simplicity and speed are prioritized over compression ratio.

The method of using Run-Length Encoding for compressing data was investigated and considered as a candidate for the implementation of the special-purpose compression module.

CBOR Encoding

Concise Binary Object Representation (CBOR) [23], is a compact binary serialization format designed for efficient encoding and decoding of structured data, in a compact way to support systems with very limited memory, processor power, and instruction sets. CBOR enables the representation of various data types, including integers, floats, booleans, strings, arrays, maps, and more. It serves as a means of data interchange between different systems, particularly in resource-constrained environments like embedded systems or network protocols. CBOR exhibits a self-descriptive nature that represents data in a structured way. The format defines a set of predefined major types (0-7) along with their corresponding data formats. Each major type may have specific fields associated with it, as outlined below.

<table>
<thead>
<tr>
<th>Major Type</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: Unsigned integers</td>
<td>Value (contains the actual integer value)</td>
</tr>
<tr>
<td>1: Negative integers</td>
<td>Value (contains the actual negative integer value)</td>
</tr>
<tr>
<td>2: Byte strings</td>
<td>Length (size of the byte string) Value (contains the byte string data)</td>
</tr>
<tr>
<td>3: Text strings</td>
<td>Length (size of the text string) Value (contains the text string data)</td>
</tr>
<tr>
<td>4: Arrays</td>
<td>Length (number of elements in the array) Value (contains the array elements)</td>
</tr>
<tr>
<td>5: Maps</td>
<td>Length (number of key-value pairs in the map) Value (contains the map key-value pairs)</td>
</tr>
<tr>
<td>6: Tagged data</td>
<td>Tag (indicates the specific data tag) Value (contains the tagged data)</td>
</tr>
<tr>
<td>7: Additional information and simple data types</td>
<td>Value (contains the specific data type)</td>
</tr>
</tbody>
</table>

For this project, I use CBOR encoding and decoding for the special-purpose compression implementation.

2.6 Related work

This sections presents some related work and aims to provide an overview of existing research and studies that closely align with the objectives of this thesis.
2.6.1 Efficient Sensor Network Reprogramming through Compression of Executable Modules

In a study by Tsiftes et al. [24], seven compression methods were tested to evaluate their performance in reducing the time and energy needed to update the software on the nodes in a sensor network. In the study, it was found that compressing code makes the update smaller, but the sensor nodes have to spend time and energy to decompress the code. Additionally, the decompression software itself can take up to 10 kilobytes of space to be stored on the sensor nodes.

The study highlights the effectiveness of compression in reducing the size of data being transmitted, which is an important aspect in networks with limited bandwidth. However, it also informs the importance of considering the processing time and energy required for decompression on the sensor nodes. When studying the impact of compression on data structures sent over TCP in sensor networks, it is relevant to consider both the benefits of reduced transmission size and the potential drawbacks of increased processing time, energy consumption, and storage requirements for both compression and decompression.

In the study, different compression methods are investigated for their compression ratios, energy consumption, and memory footprints. The results of these investigations are relevant to the studies made in this project as a basis for choosing what compression method to adopt for the general-purpose compression module. The following software was used for the investigations of the study:

<table>
<thead>
<tr>
<th>Name</th>
<th>ELF file size</th>
<th>Entropy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File A</td>
<td>7548</td>
<td>3.62</td>
<td>Driver for the dcf2411 sensor.</td>
</tr>
<tr>
<td>File B</td>
<td>2152</td>
<td>3.81</td>
<td>Radio connectivity testing.</td>
</tr>
<tr>
<td>File C</td>
<td>3560</td>
<td>4.13</td>
<td>Trickle [2].</td>
</tr>
<tr>
<td>File D</td>
<td>4240</td>
<td>4.71</td>
<td>ELF loader.</td>
</tr>
<tr>
<td>File E</td>
<td>8564</td>
<td>4.68</td>
<td>A convergent protocol.</td>
</tr>
<tr>
<td>File F</td>
<td>11704</td>
<td>4.67</td>
<td>The aIP TCP/IP stack.</td>
</tr>
</tbody>
</table>

![Figure 2.1: The software used to evaluate the compression algorithms. The sizes are in bytes.][24, Table I]

It is important to consider that the software and data used for the study is different from the investigation made in this paper, which may result in varying results by the different compression methods. The following are the results of the study made by Tsiftes et al.

<table>
<thead>
<tr>
<th>File</th>
<th>Original size</th>
<th>AC</th>
<th>GZIP</th>
<th>LZARI</th>
<th>LZOIX</th>
<th>S-LZW</th>
<th>SBRZIP</th>
<th>VCDIFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>File A</td>
<td>1208</td>
<td>740</td>
<td>645</td>
<td>641</td>
<td>844</td>
<td>816</td>
<td>664</td>
<td>777</td>
</tr>
<tr>
<td>File B</td>
<td>2176</td>
<td>1146</td>
<td>1037</td>
<td>1066</td>
<td>1447</td>
<td>1274</td>
<td>1020</td>
<td>1316</td>
</tr>
<tr>
<td>File C</td>
<td>3960</td>
<td>1981</td>
<td>1542</td>
<td>1694</td>
<td>2200</td>
<td>2203</td>
<td>1708</td>
<td>1977</td>
</tr>
<tr>
<td>File D</td>
<td>4240</td>
<td>2617</td>
<td>2014</td>
<td>2255</td>
<td>2860</td>
<td>2873</td>
<td>2260</td>
<td>2551</td>
</tr>
<tr>
<td>File E</td>
<td>8564</td>
<td>5147</td>
<td>3789</td>
<td>4494</td>
<td>5720</td>
<td>5681</td>
<td>4583</td>
<td>4853</td>
</tr>
<tr>
<td>File F</td>
<td>11704</td>
<td>6492</td>
<td>4754</td>
<td>5623</td>
<td>7345</td>
<td>7379</td>
<td>5714</td>
<td>6236</td>
</tr>
<tr>
<td>Average</td>
<td>5259</td>
<td>3017</td>
<td>2297</td>
<td>2639</td>
<td>3403</td>
<td>3371</td>
<td>2674</td>
<td>2952</td>
</tr>
<tr>
<td>Savings</td>
<td>42.6%</td>
<td>56.3%</td>
<td>50.0%</td>
<td>35.2%</td>
<td>36.0%</td>
<td>49.1%</td>
<td>43.9%</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 2.2: Compressed file size (in bytes) of the different algorithms. The VCDIFF files have been generated with the XDELTA3 software.][24, Table III]

The investigation of the effectiveness of the compression with the different algorithms shows that GZIP, in general, provides the highest compression ratio.
Looking at the memory footprint and code size, AC consumes the smallest amount of space compared to the other algorithms.

In contrast to the memory footprint and storage requirement, AC has the longest execution time, with LZO1X being the fastest method.

These results provide an insightful picture of the effectiveness of the different compression algorithms that give an understanding of how the methods are affected by the varying file formats.

2.6.2 Efficient and Flexible Sensornet Checkpointing

Another study by Lösch et al. [25], presents work done on sensornet checkpointing, which includes extensions such as compression, binary diffs, selective checkpointing, and checkpoint inspection. These extensions, including compression, aim to reduce the time required for checkpointing operations and improve the granularity of system state examination. Through experimental evaluation, it is demonstrated that these improvements result in significant reductions in checkpoint sizes and time, with checkpoint sizes reduced by 70%-93% and time reduced by at least 50%.

These findings can be directly related to the objectives of this thesis as they focus on data compression techniques to enhance the efficiency of transmitting data in IoT networks. By shrinking the file size of full checkpoints, the proposed improvements contribute to more efficient checkpointing and rollback operations, thus reducing transmission time and resource consumption.
2.6.3 Data Compression Algorithms for Energy-Constrained Devices in Delay Tolerant Networks

A further study, by Sadler et al. [26], introduces the challenges faced in sensor networks regarding the difficulty and energy consumption associated with delivering information from sensors to the sink. It emphasizes the importance of energy improvements achieved through computationally-efficient lossless compression algorithms implemented on the source nodes. By reducing the amount of data transmitted through the network and to the sink, these algorithms offer multiplicative energy benefits based on the number of hops the data travels.

The text also highlights the limitations in existing approaches, where sensor system designers either develop application-specific compression algorithms or use off-the-shelf algorithms not optimized for resource-constrained sensor nodes. This paper addresses the design issues related to implementing, adapting, and customizing compression algorithms specifically tailored for sensor nodes. The mention of the Sensor LZW (S-LZW) algorithm and its variations demonstrates the efforts made to develop effective compression techniques for energy savings, considering factors such as compression levels and radio hardware.

Furthermore, the work is validated and evaluated using datasets from real-world deployments, showing energy consumption reductions of up to a factor of 4.5X across the network. This empirical evidence reinforces the significance of the proposed approaches and their potential for achieving efficient data transmission in IoT networks. Overall, this related work provides valuable insights into the development and application of compression algorithms for energy-efficient data transfer, which aligns closely with the objectives of this thesis.
Chapter 3

Design and Implementation

This chapter details the design and implementation of a set of known compression algorithms and a special purpose compression algorithm. To evaluate the performance of the different data compression algorithms on wireless sensor networks using dynamically linkable modules, I integrated the modules in Contiki-NG. This chapter primarily addresses the design decisions when adapting and implementing the data compression modules along with the resulting implementation.

3.1 Integrating New Software into Contiki-NG

Integrating new software modules into Contiki-NG involves a systematic process that includes module selection and implementation, source code integration, Makefile configuration, defining configuration options, API definition and usage, compilation and build process, and testing and debugging. Careful selection, implementation, and integration ensure compatibility and efficient utilization within the Contiki-NG framework. Configuration options and API provide customization capabilities, while testing and debugging verify the functionality of the integrated software modules. This systematic approach enhances the functionality and customization possibilities within Contiki-NG.

3.2 TCP Connection and Modification

For this project, as mentioned in Section 2.4, I use an existing implementation of a TCP socket, that establishes a connection using IPv6. The choice of using this TCP implementation is mainly based on its arrangement of a complete solution of a reliable connection, which is able to send a larger amount of data. Though it creates more overhead than, for instance, CoAP and UDP, TCP gives a high data integrity and delivery guarantee, which is especially beneficial when sending critical or sensitive data.

In order to improve the performance and reliability of the TCP socket implementation, I add several new functionalities to the existing code base. These additions are listed below.

- Adding a further checksum calculation on the client side and validation on the server side.
- Changing the transmission to send smaller sections of the data rather than one big chunk.
- Adding a header to the data containing the size of the data being sent the type of compression used (or indicates if no compression is used).
- Adding functionality to integrate the compression modules.
The first step is to add a checksum integrity verification to the TCP socket. This is achieved by calculating a checksum value of the data on the client side. For this, a simple CRC (Cyclic Redundancy Code) implementation is used. Before sending the data to the server, the calculated checksum is added to the checksum field of the structure. On the server side, the received checksum data is checked against the computed checksum to ensure that the data had not been corrupted during transmission.

In order to keep track of the size and type of the compression used, a header is added to the data being sent. The header contained information about the size of the data, as well as information about the type of compression used (or indicating if no compression is used). This allows the server to correctly handle and decompress the received data and also simplifies switching between the different data compression methods.

The next step is to send the data in sections, to improve the reliability of the system by reducing the chances of data loss due to network issues. This is achieved by allowing the client to send only smaller, more manageable sections of the data, until the entire buffer is sent.

Finally, functionality to handle and integrate the different data compression modules is added. This includes adding configurations for choosing which method to use or if none should be used, variables for handling the data, and error checking to ensure that the compression and decompression are successful.

Overall, these additional functionalities greatly improves the performance and reliability of the TCP socket implementation, making it more robust and efficient for transmitting data over the network.

### 3.3 Data Structure

For the evaluation of this project, the data compression modules are tested by applying them to a data structure, entitled packet_stats, that correlates to the traffic statistics information sent from the nodes within a wireless sensor network to the edge node.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>checksum</td>
<td>The checksum value of the packet.</td>
</tr>
<tr>
<td>magic</td>
<td>The signature or identifier for the packet statistics.</td>
</tr>
<tr>
<td>version</td>
<td>The version of the packet statistics.</td>
</tr>
<tr>
<td>packet_count</td>
<td>The total number of packets.</td>
</tr>
<tr>
<td>too_large_packets</td>
<td>The number of packets that were too large to be processed.</td>
</tr>
<tr>
<td>packet_lengths</td>
<td>The lengths of the packets.</td>
</tr>
<tr>
<td>icmpv6</td>
<td>Statistics related to Internet Control Message Protocol version 6 (ICMPv6).</td>
</tr>
<tr>
<td>ipv6</td>
<td>Statistics related to Internet Protocol version 6 (IPv6).</td>
</tr>
<tr>
<td>rpl</td>
<td>Statistics related to Routing Protocol for Low-Power and Lossy Networks (RPL)</td>
</tr>
<tr>
<td>tcp</td>
<td>Statistics related to Transmission Control Protocol (TCP).</td>
</tr>
<tr>
<td>udp</td>
<td>Statistics related to User Datagram Protocol (UDP).</td>
</tr>
<tr>
<td>src_addr_count</td>
<td>The number of packets sent from each source address.</td>
</tr>
<tr>
<td>dst_addr_count</td>
<td>The number of packets sent to each destination address.</td>
</tr>
<tr>
<td>src_addr</td>
<td>The source IP addresses of the packets.</td>
</tr>
<tr>
<td>dst_addr</td>
<td>The destination IP addresses of the packets.</td>
</tr>
</tbody>
</table>

The data structure, depicted in Figure 3.1 is of size 8896 bytes, and contains information on various packet statistics, such as protocol and address information along with some meta-
data, that is used to analyze the performance and behavior of a network. For the evaluation experiments in this project, the data structure is populated with artificially generated values. For some experiments, the data contains random values, while for others, it consists solely of the integer 1.

3.4 Adapting Existing Compressors for Sensor Nodes

In this section, I discuss the process of adapting existing compressors that meet the requirements of the resource-constrained sensor nodes. The initial considerations when adapting an existing compression method for a data structure sent over a TCP connection in Contiki-NG, involved understanding the data structure, analyzing the existing compressing module, modifying the module to make it compatible with the data structure (if needed), and integrating the modified module with the TCP connection.

3.4.1 Analysing the Information Entropy of the Data

The information entropy of the data provides insightful information on the upper limits of statistical compression methods. Therefore, An information entropy analysis of the “packet_stats” structure is done with non-random and random values and percentage filled, as shown in Table 3.2 and Table 3.3.

<table>
<thead>
<tr>
<th>Struct filled (%)</th>
<th>Entropy (b/B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>5</td>
<td>0.193</td>
</tr>
<tr>
<td>25</td>
<td>0.565</td>
</tr>
<tr>
<td>50</td>
<td>0.829</td>
</tr>
<tr>
<td>75</td>
<td>0.999</td>
</tr>
<tr>
<td>100</td>
<td>0.999</td>
</tr>
</tbody>
</table>

Table 3.2: Entropy of the “packet_stats” structure with percentages filled and non-random values.

<table>
<thead>
<tr>
<th>Struct filled (%)</th>
<th>Entropy (b/B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>5</td>
<td>0.733</td>
</tr>
<tr>
<td>25</td>
<td>2.813</td>
</tr>
<tr>
<td>50</td>
<td>4.962</td>
</tr>
<tr>
<td>75</td>
<td>7.977</td>
</tr>
<tr>
<td>100</td>
<td>7.971</td>
</tr>
</tbody>
</table>

Table 3.3: Entropy of the “packet_stats” structure with percentages filled and random values.

For the case where non-random values are used, in this case the structure is filled with the integer 1, the entropy values increase gradually as the structure is filled with more values. At 100% of structure filled with data, the entropy reaches its maximum value of 0.999 bits per byte, suggesting that with an effective compression method the data could be reduced to 0.125% of its original size.

In contrast, when random values are used, that were generated with Contiki-NGs built in function random(rand), the entropy values increase more rapidly as the structure is filled. This indicates that the random values introduce greater unpredictability and higher entropy.
At 100% structure filled, the information entropy reaches a higher value of 7.971 bits per byte, indicating a higher degree of information contained within the data. As the relationship between the information entropy and the data is nearly one-to-one, it suggests that even with an effective compression module, the reduction in size would be minimal.

With this being said it is worth noting than in real-life scenarios, the structure would very unlikely be 100% filled and would exhibit some patterns in the data. Therefore, It is important to interpret the results with caution and a level of skepticism.

3.4.2 Investigating the Different Compression Methods

The selection of compressors involved an evaluation of various algorithms based on their compression performance. Furthermore, When deciding on what data compression algorithm to use, multiple factors were considered to see what type of method would be most efficient. These included the information entropy of the data, considering that it has to be lossless compression, and also the complexity of the method and its availability to integrate with the TCP connection.

From the aspect of the complexity of the module, I opted to use a statistical algorithm, as these algorithms generally tend to have more simple implementations compared to the Lempel-Ziv algorithms. Statistical algorithms excel in situations where the data exhibits predictable patterns typically found in low entropy data. These algorithms analyze the statistical properties of the data and adapt their encoding scheme accordingly, enabling effective compression. On the other hand, Lempel-Ziv algorithms, while still capable of compressing low entropy data, may not achieve the same level of compression efficiency as statistical algorithms, as the dictionary and sliding-window approaches may not be as effective in capturing and exploiting the statistical regularities present in low entropy data. However, It is important to note that the performance of compression algorithms can vary depending on the specific characteristics of the data being compressed, and these properties should be considered as general guidelines.

The consideration was given to employing data transformation techniques that can rearrange the data, making it more suitable for compression in the best possible cases. For example, the bzip2 compressor uses the Burrows-Wheeler transform before compressing the data with MTF and Huffman coding. As the given data altogether has a low entropy when containing a smaller amount of data, which corresponds to real-life scenarios, the approach of using transformations algorithms would be predictably unnecessary and more beneficial on a structure with higher entropy. The implementation of a data transformation is nevertheless dismissed due to time constraints and restrictions concerning the the project scope.

3.4.3 Adapting Arithmetic Coding

Arithmetic coding is chosen as the general-purpose compression module for adaptation in the network due to several considerations. One of the primary reasons for selecting arithmetic coding is the availability of an existing arithmetic coding module within the Contiki-NG system. This module provided a foundation that is leveraged for the specific requirements of the sensor nodes. This facilitated the development process, as it eliminated the need to start from scratch and provided a solid foundation to extend and optimize for energy efficiency and memory usage.

Arithmetic coding’s ability to accurately represent and assign shorter codes to frequent symbols or patterns in the data makes it effective in compressing data with low entropy, which is appropriate for the given data when it contains less than 50% data or the data is non-random.

The adaptability of arithmetic coding to the constraints of resource-constrained sensor nodes is another factor in its selection. The adaptive order-0 model used by the arithmetic coder chosen aligns well with the limited memory capacity of sensor nodes. This feature allows for efficient
memory utilization, ensuring that the compression module can operate within the constrained memory resources available in the sensor nodes.

Additionally, integrating the arithmetic coding module into the TCP client and server in Contiki-NG required careful consideration of the system architecture and communication protocols. The compression module needed to seamlessly fit into the existing network infrastructure without disrupting the normal flow of data transmission.

During the integration process, various modifications were made to the TCP client and server to support compression. This involved incorporating the necessary function calls and hooks to enable compression at the appropriate stages of the communication process. The integration also required establishing a mechanism for negotiating and signaling the use of compression between the client and server, ensuring compatibility and synchronization.

As mentioned in Section 3.2, a header is added and sent together with the data. The header is necessary for the synchronization between the client and server for the compression and decompression stages. The header is used to send information, such as what compression method is used or if none is initiated, and the size of the compressed data. This information is crucial for allowing the server to decompress the data correctly and initiating the decompression when all data is received.

To test the implemented compression module and its compatibility with the TCP connection, a simulation of the connection, configured to use the module, is run in Cooja. The simulation is run multiple times, with differently allocated “packet_stats” structures being compressed. The results of the compression and decompression is logged to see if any disturbances occurred during the simulation.

In conclusion, the adaptation of arithmetic coding to the system involved leveraging existing modules, ensuring compatibility with the data structure, integrating the code seamlessly into the TCP client and server, and thorough testing.

3.5 Implementing a Special-Purpose Compressor for Sensor Nodes

In this section, I discuss the process of implementing and investigating a special-purpose data compression method to meet the requirements of resource-constrained sensor nodes. The main considerations for finding and choosing a suitable method is to analyze the data structure, to study if any data is redundant and thereafter identify a method to arrange the data to make it smaller in a way that is lossless.

3.5.1 Investigating the Data Contents

To be able to create a special-purpose data compression module that is tailored around the “packet_stats” structure, an examination is done on the given data structure. The analysis revolved around the types of data being collected and their characteristics, such as the size, structure, and distribution of the data. This information is then used to inform the design and implementation of the special-purpose compressor and also aided in evaluating the performance of the compression method.

By investigation of the data structure, it is evident that some of the content is redundant. Fields in the structure, that are represented with an array, are not always filled. This implies that sending the whole array is redundant and therefore causes unnecessary data to be sent. Organizing the data, to send only the non-empty elements of the arrays, can be done in multiple ways.
Figure 3.1: Visualization of memory distribution of the “packet_stats” structure.

Figure 3.1 shows a visual representation of the memory distribution of the packet_stats structure. In the figure, we can observe that multiple of the fields contain arrays of long lengths. These are the fields that may cause redundancies.

Figure 3.2: Zoomed in visualization of memory distribution of the “packet_stats” structure.

An example of this is the “packet_lengths” field represented by an array with a length of 1281 elements used for storing the lengths of the packets sent, with each element occupying 2 bytes of memory. In the case where only a single element of this array contains meaningful data, the rest of the 1280 elements would be redundant. Considering this, it becomes apparent
that transmitting the entire “packet_lengths” array with 1280 redundant elements results in unnecessary data transmission and inefficient use of network resources. Instead, by only sending the elements containing useful data, the “packet_lengths” data could become, in best case scenario, as small as the “version” data, which occupies only 2 bytes of the memory, as can be seen in Figure 3.2.

By addressing such redundancies and implementing appropriate strategies for data compression and transmission, the special-purpose compressor can effectively eliminate unnecessary data and ensure efficient utilization of network resources, leading to improved data transmission performance.

3.5.2 Investigating Methods of Reducing Data Size

As specified in the previous section, a significant concern regarding the data is the possibility of redundancy in certain fields. The majority of unnecessary data that is transmitted originates from arrays not being filled. Redundancies in non-filled arrays can be handled by implementing efficient data structures and algorithms that only transmit and store the non-empty elements of the arrays. Instead of sending the entire array, only the relevant elements containing data are transmitted or stored.

There are multiple methods for accomplishing this, with one approach being to use a dynamic data structure, such as a linked list or dynamic array, where the size of the structure adjusts dynamically based on the number of non-empty elements. This way, only the necessary elements are stored, reducing the overall storage requirements. Using a linked list allows efficient memory utilization, as it is possible to adjust their size dynamically, allocating memory only to the necessary elements. Nevertheless, Accessing elements in a linked list requires traversing through the list, which can be slower than accessing elements directly in a compact array. Furthermore, Linked lists require additional memory to store the pointers or references to the next element, resulting in increased memory overhead compared to a compact array representation.

CBOR encoding is a further method that can be used for efficiently encoding complex data structures and various data types concisely. As CBOR is designed to be used on resource-constrained systems and offers a compact binary format and efficient encoding, it is appropriate for IoT devices. Another preference for using CBOR is its ability to handle different data types, including integers, strings, arrays, and maps, in a unified manner. This allows you to represent and compress diverse data elements within an array efficiently. CBOR’s built-in mechanisms can also be leveraged for representing arrays of varying lengths and handling sparse data, which is suitable for the given data structure.

However, It is necessary to consider the trade-offs when using CBOR. Encoding and decoding data using CBOR may introduce additional processing overhead, as it requires converting data into its binary representation, applying specific encoding rules, and potentially adding additional metadata, all of which can incur computational costs.

Another method is to use a run-length encoding. This method may not provide substantial compression gains in the case of arrays with non-filled elements, as the non-filled elements are essentially treated as individual unique elements, which would not result in significant savings in terms of storage space. Instead, RLE can be applied to compress other parts of the data that exhibit repetition and be used in combination with another method that handles the arrays. This approach is not employed, as it would create a larger memory footprint, and repetitiveness in the structures data is not very likely.
3.5.3 Adapting CBOR

CBOR encoding is chosen as the preferred method for handling redundancy within the structure and reducing unnecessary data transmission for the special-purpose compression module. By leveraging CBOR’s ability to efficiently encode complex data structures and various data types, including arrays, concisely, storage requirements could be minimized. CBOR’s unified handling of different data types, such as integers, strings, arrays, and maps, allows for the representation and compression of diverse data elements within the structure.

Although CBOR encoding introduces some processing overhead due to the conversion of data into binary representation and application of encoding rules, this trade-off is deemed acceptable considering the significant reduction in redundant data transmission. A minimalistic, open-source CBOR implementation made by Kyunghwan Kwon [27], and licensed by MIT, is chosen for adoption to the code base. The implementation adheres to the C99 standard and retains a small code footprint, which corresponds well with Contiki-NG.

As the chosen CBOR code base is a custom implementation, it differs a bit from the standard CBOR format. When encoding with Kwon’s CBOR implementation, you encode each field of the data structure accordingly to the specific datatype of the data. The data is then encoded into a CBOR item, which in this specific implementation contains meta data such as the size and data type together with the encoded data. The CBOR item is automatically inserted to a CBOR writer, that is initialized beforehand and will, after the encoding process, hold all CBOR items to be sent.

This process is done for each field of the “packet_stats” structure, with some fields handled in specific sub-processes. These are fields that do not hold a single integer as data, such as “packet_lengths”, “tcp_stats” and “src_addrs”, to name a few. As these hold arrays, it is critical to only encode elements of the arrays which hold important data, to ensure compression.

Algorithm 3 Encode Array

<table>
<thead>
<tr>
<th>Input:</th>
<th>A: Array to be encoded</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>procedure ENCODEARRAY(A)</td>
</tr>
<tr>
<td>2:</td>
<td>size ← GetArraySize(A);</td>
</tr>
<tr>
<td>3:</td>
<td>count ← 0;</td>
</tr>
<tr>
<td>4:</td>
<td>for i ← 0 to size − 1 do</td>
</tr>
<tr>
<td>5:</td>
<td>if A[i] &gt; 0 then</td>
</tr>
<tr>
<td>6:</td>
<td>count ← count + 1</td>
</tr>
<tr>
<td>7:</td>
<td>end if</td>
</tr>
<tr>
<td>8:</td>
<td>end for</td>
</tr>
<tr>
<td>9:</td>
<td>EncodeInteger(count)</td>
</tr>
<tr>
<td>10:</td>
<td>for i ← 0 to size − 1 do</td>
</tr>
<tr>
<td>11:</td>
<td>if A[i] &gt; 0 then</td>
</tr>
<tr>
<td>12:</td>
<td>EncodeInteger(i)</td>
</tr>
<tr>
<td>13:</td>
<td>EncodeInteger(A[i])</td>
</tr>
<tr>
<td>14:</td>
<td>end if</td>
</tr>
<tr>
<td>15:</td>
<td>end for</td>
</tr>
<tr>
<td>16:</td>
<td>end procedure</td>
</tr>
</tbody>
</table>

The sub-process responsible for handling arrays follows the algorithm outlined in Algorithm 3. Initially, the number of non-zero elements of the array is counted. This number is then encoded to the writer, and serves as metadata for the CBOR decoder to be able to decode the data properly. Then the process iterates over the array, encoding each each index-value pair.
By including both the index and its corresponding value in the encoding process, the integrity of the decoding process is ensured.

This means that some extra data is added to the resulting data to be transmitted as the non-zero elements count and array indices are not included in the original data. This may cause the encoded data to be bigger than its original size, and is therefore handled as a special case where the original data is sent instead.

The decoding process for CBOR follows a similar approach to the encoding process. It begins by initializing a CBOR reader, which holds the decoded data in the form of CBOR items. The reader is then parsed, populating an array specifically designed to store the CBOR items. Iterating through the items array, each item is matched and decoded onto its corresponding field in the “packet_stats” structure. Additionally, the metadata gathered during the encoding of the data is used in the process of decoding the arrays and correctly storing their values at their respective indices.

In conclusion, the adaptation of CBOR encoding and decoding for the special-purpose compression module involved several steps, including selecting a suitable CBOR implementation, ensuring compatibility with the “packet_stats” structure, seamlessly integrating the code into the TCP client and server with a focus on the data structure, and addressing any arising issues or flaws.
Chapter 4

Evaluation

This chapter covers the evaluation of the project, which includes the compression ratios, the execution times, and the storage requirements of the implementations. The various experiments made to investigate the performance of the general-purpose compression module and the special-purpose compression module is also described.

4.1 Experiment Setup

When testing the programs, they are run in Cooja, within a Cooja Script (CSC) file, that contains configuration settings and parameters for a simulation scenario. These files provide a way to script and automate simulations by specifying the behavior of simulated nodes, their interactions, and the environment in which they operate. This allows for definition of various simulation parameters, such as the network topology, node properties, mobility patterns, radio settings, and event scheduling.

The simulated scenario used for the project experiments defines a TCP socket test, including settings for motes, radio communication, logging, visualization, and plugins used in Cooja. The socket includes two mote types: the client and server motes; each with their respective
configurations. The client mote of the simulations represents the IoT device of the traffic monitoring module, while the server mote serves as the edge device. In the simulations, one server mote and one client mote are active, as seen in Figure 4.1.

4.2 Evaluation Measures

When examining the impact of compressing the data intended for transmission over the wireless network, aspects such as compression ratio, execution time, and storage requirements are considered. The objective of designing the compression modules include the following:

- High compression ratio
- Fast execution times
- Small storage requirements

The compression ratio serves as a crucial metric for assessing the effectiveness of the compression algorithm in reducing the size of the data. In the context of wireless networks, achieving a high compression ratio is of utmost importance. A higher compression ratio minimizes the bandwidth usage required for transmission by significantly reducing the data size.

Fast execution times play a vital role in data compression algorithms. Quick compression and decompression processes contribute to reducing latency in wireless network communication.

Furthermore, the storage requirement of the compressed data is a factor to consider. Compressed data occupies less storage space compared to the original uncompressed data. This size reduction is valuable in situations where storage capacity is limited. It allows for longer data retention, efficient data archiving, and improved scalability.

4.3 Compression Ratio

The compression ratio is used to measure the effectiveness of reducing the size of the given data of a compression algorithm. The following formula can be used to determine the compression ratio.

\[
\text{Compression Ratio} = \frac{\text{Uncompressed Size}}{\text{Compressed Size}}
\]

For instance, assuming an initial file of size 1,000 kilobytes (kB) becomes 500 kB after compression, the compression ratio would be 50%, or in explicit notation 2:1, since the compressed file is half the size of the original file.

Table 4.1: Compressed file sizes (in bytes) of the “packet_stats”-structure, using Arithmetic encoding, with varying percentage of structured filled with non-random values.

<table>
<thead>
<tr>
<th>Filled (%)</th>
<th>AC</th>
<th>Savings (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>216</td>
<td>97.57</td>
</tr>
<tr>
<td>5</td>
<td>431</td>
<td>95.16</td>
</tr>
<tr>
<td>25</td>
<td>844</td>
<td>90.51</td>
</tr>
<tr>
<td>50</td>
<td>1137</td>
<td>87.22</td>
</tr>
<tr>
<td>75</td>
<td>1326</td>
<td>85.09</td>
</tr>
<tr>
<td>100</td>
<td>1326</td>
<td>85.09</td>
</tr>
</tbody>
</table>
Table 4.2: Compressed file sizes (in bytes) of the “packet_stats”-structure, using CBOR encoding, with varying percentage of structured filled with random values.

<table>
<thead>
<tr>
<th>Filled (%)</th>
<th>CBOR</th>
<th>Savings (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>474</td>
<td>94.67</td>
</tr>
<tr>
<td>5</td>
<td>1156</td>
<td>87.01</td>
</tr>
<tr>
<td>25</td>
<td>3824</td>
<td>57.01</td>
</tr>
<tr>
<td>50</td>
<td>7143</td>
<td>19.71</td>
</tr>
<tr>
<td>75</td>
<td>8896</td>
<td>0.00</td>
</tr>
<tr>
<td>100</td>
<td>8896</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 4.1 and Table 4.2 illustrate the performance of the arithmetic compressor and CBOR encoding techniques when applied to the “packet_stats” structure with varying degrees of the data structure filled with non-random values. The arithmetic compressor demonstrates a reduction in file size, with savings percentages ranging from 97.57% for 0% of the data structure filled to 85.09% for 100% of the data structure filled. This shows that AC’s ability to reduce the data size levels out after the structure is more than 50% filled, but still provides substantial savings.

On the other hand, the results of the CBOR encoding technique differ from those of the arithmetic compressor. As the percentage of the structure being filled increases, the compressed file sizes fluctuate. The savings percentage starts at a high value of 94.67% for 0% of the data structure filled, but gradually decreases to 0% for percentages higher than 50%. This implies that CBOR is unable to reduce the size of the data structure when it is more than 50% full, which is a result of CBOR adding additional metadata for decoding purposes. These findings indicate that CBOR encoding is less effective in compressing structured data with higher percentages of the structure filled with non-random values compared to the arithmetic compressor.

![Figure 4.2: Comparison of the compression ratios when the structure is filled with non-random data.](image)

The arithmetic compressor consistently achieves a reduction in file size and higher savings percentages across the range of the percentages of the structure being filled with data, while the effectiveness of CBOR encoding diminishes with increasing percentages. These findings
can be observed in Figure 4.2, and emphasize the differing capabilities of the two compression techniques in handling structured data with varying characteristics.

Table 4.3: Compressed file sizes (in bytes) of the “packet_stats”-structure, using Arithmetic coding, with varying percentage of structured filled with non-random values.

<table>
<thead>
<tr>
<th>Filled (%)</th>
<th>AC</th>
<th>Savings (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>216</td>
<td>97.57</td>
</tr>
<tr>
<td>5</td>
<td>980</td>
<td>88.98</td>
</tr>
<tr>
<td>25</td>
<td>3253</td>
<td>63.43</td>
</tr>
<tr>
<td>50</td>
<td>5627</td>
<td>36.75</td>
</tr>
<tr>
<td>75</td>
<td>8896</td>
<td>0.00</td>
</tr>
<tr>
<td>100</td>
<td>8896</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 4.4: Compressed file sizes (in bytes) of the “packet_stats”-structure, using CBOR encoding, with varying percentage of structured filled with random values.

<table>
<thead>
<tr>
<th>Filled (%)</th>
<th>CBOR</th>
<th>Savings (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>474</td>
<td>94.67</td>
</tr>
<tr>
<td>5</td>
<td>1621</td>
<td>81.78</td>
</tr>
<tr>
<td>25</td>
<td>5959</td>
<td>33.01</td>
</tr>
<tr>
<td>50</td>
<td>8896</td>
<td>0.00</td>
</tr>
<tr>
<td>75</td>
<td>8896</td>
<td>0.00</td>
</tr>
<tr>
<td>100</td>
<td>8896</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 4.3 and Table 4.4 present the results of the experiment conducted on the “packet_stats” structure with varying percentages of structured filled with random values. For the arithmetic compressor, as the percentage of random values in the structure increases, the compressed file sizes gradually increase, while the savings percentages decrease. At 0% of the structure filled, the arithmetic compressor achieves a savings percent of 97.57%, which decreases to 0% for percents higher than 50% of the structure filled. This indicates that the arithmetic compressor is unable to effectively compress the structured data with a high percentage of random values.

Similarly, for CBOR encoding, the compressed file sizes and savings percentages vary with the percent of random values. The savings percentage starts at 94.67% for 0% of the structure filled and decreases to 0% for percents higher than 25%, as the compressed size exceeds the original data size. These results suggest that CBOR encoding is less effective in compressing the structured data with a higher proportion of random values.
The findings demonstrate that the arithmetic compressor and CBOR encoding face challenges in compressing structured data with a significant presence of random values, shown in Figure 4.3. The compressed file sizes increase, and the savings percentages decrease as the randomness in the data increases. This indicates the limitations of these compression techniques when applied to structured data containing a considerable amount of random values.

Furthermore, a small experiment was done where a combination of the compression methods was used. It shows that using arithmetic coding on data that is already encoded with CBOR, where the structure is 50% filled with non-random values, resulted in 89.53% of saved space. This is a substantially better result than using CBOR encoding alone and also results in a slightly increased savings of space than using only arithmetic coding. Doing the same double compression where the structure is 5% filled with non-random values, the result shows a saved space of 98.29%. The approach of combining multiple compression methods is a practice that could be investigated further, with a focus on the trade-offs regarding the execution times.

The results of the compression experiments reveal that as the proportion of random values increases in the structured data, the performance of both the arithmetic compressor and CBOR encoding techniques deteriorates. With more random values and larger data sizes, the compressed file sizes tend to increase, while the savings percentages decrease. These findings indicate that the compressors have limitations in effectively compressing structured data with a high presence of random values. However, it is important to note that in real-life scenarios, the structured data is often not filled with completely random values, and it is also not consistently fully filled. In practical applications, the data within the structure tends to exhibit patterns, correlations, or specific characteristics. This means that the limitations observed in the second compression experiment may not significantly impact real-life scenarios, as the structured data is likely to contain non-random and predictable elements.

4.4 Execution times

The execution times of the TCP socket operations are measured using a timer, that begins when a TCP connection is established and continues until all data has been transmitted over the socket. This enables the monitoring and analysis of the time required for the additional code
involved in compressing and decompressing the data. By using the timer, it becomes possible to assess the impact of the compression and decompression processes on the overall execution time of the TCP socket operations.

Table 4.5: Comparison of execution times (in clock seconds) of the different compression methods with varying percentage of structured filled with non-random values.

<table>
<thead>
<tr>
<th>Filled (%)</th>
<th>AC</th>
<th>CBOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>238</td>
<td>375</td>
</tr>
<tr>
<td>5</td>
<td>363</td>
<td>883</td>
</tr>
<tr>
<td>25</td>
<td>627</td>
<td>2618</td>
</tr>
<tr>
<td>50</td>
<td>835</td>
<td>4591</td>
</tr>
<tr>
<td>75</td>
<td>948</td>
<td>5528</td>
</tr>
<tr>
<td>100</td>
<td>948</td>
<td>5528</td>
</tr>
</tbody>
</table>

Table 4.6: Comparison of execution times (in clock seconds) of the different compression methods with varying percentage of structured filled with random values.

<table>
<thead>
<tr>
<th>Filled (%)</th>
<th>AC</th>
<th>CBOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>238</td>
<td>375</td>
</tr>
<tr>
<td>5</td>
<td>630</td>
<td>1075</td>
</tr>
<tr>
<td>25</td>
<td>2167</td>
<td>3730</td>
</tr>
<tr>
<td>50</td>
<td>3471</td>
<td>5251</td>
</tr>
<tr>
<td>75</td>
<td>5539</td>
<td>5539</td>
</tr>
<tr>
<td>100</td>
<td>5539</td>
<td>5539</td>
</tr>
</tbody>
</table>

Table 4.5 and Table 4.6 present two tables displaying the comparison of execution times, measured in clock seconds where one clock second corresponding to a frequency of 1000, for different compression methods with varying percentages of structured data filled. Two scenarios are considered: one with non-random values filled in the structure, and the other with random values filled.

For the scenario with non-random values, shown in Table 4.5, the arithmetic coding compression method consistently demonstrates lower execution times compared to the CBOR method. As the percentage of structured data filled increases, the execution times for both compression methods also increase, which is expected as more data need to be processed. This suggests that compressing structured data with non-random values incurs additional processing overhead.

In the case of the scenario with random values filled, shown in Table 4.6, the AC compression method generally exhibits lower execution times compared to the CBOR method, even though it is more affected by the random data than CBOR. As the percentage of structured data filled increases, the execution times for both compression methods continue to rise. However, It is worth noting that the execution times of the CBOR implementations increase more rapidly than those of the Arithmetic coder.
Looking at Figure 4.4 and Figure 4.5, it is evident that both compression methods increases in execution time with higher amount of data in the structure, though the Arithmetic coder provides much better execution times compared to CBOR, especially when the structure contains non-random data.

4.5 Storage Requirements

The 'size' command is used for analysing the storage requirements. The 'size' command displays the size of object files or executables. On Linux and MacOS systems, the 'size'
command can be invoked from the terminal to obtain information about the memory usage of compiled code.

Table 4.7: The sections and sub-sections of the ‘size’ command and their description.

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_TEXT</td>
<td>Contains program’s executable code, including functions and instructions.</td>
</tr>
<tr>
<td>_text</td>
<td>Displays the size of the executable code or text segment of the object file.</td>
</tr>
<tr>
<td>_literal16</td>
<td>Stores 16-bit literal data, which includes small constant values used by the program.</td>
</tr>
<tr>
<td>_cstring</td>
<td>Stores program’s constant string literals, including error messages, format specifiers and other string constants.</td>
</tr>
<tr>
<td>_DATA</td>
<td>Contains initialized global or static variables, represents modifiable program data.</td>
</tr>
<tr>
<td>_data</td>
<td>Displays the size of the initialized data segment.</td>
</tr>
<tr>
<td>_const</td>
<td>Contains read-only constant data, such as configuration values or lookup tables.</td>
</tr>
<tr>
<td>_bss</td>
<td>Reserves memory for uninitialized global or static variables, typically zero-filled at startup. Used for variables without explicit initialization.</td>
</tr>
<tr>
<td>others</td>
<td>Other sections present in the object file that are not explicitly mentioned, such as exception handling.</td>
</tr>
</tbody>
</table>

For example, running ‘size -m myfile.o’ provides a breakdown of the memory sections within the object file. The sections explored for the evaluation of the compression modules can be seen in Table 4.7. More details about the usage and available options of the ‘size’ command can be found in the command’s manual page (‘man size’).

Table 4.8: Comparison of the storage requirements (in bytes) of the executables for the compression methods.

<table>
<thead>
<tr>
<th>File</th>
<th>_TEXT</th>
<th>_DATA</th>
<th>others</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac.o</td>
<td>1956</td>
<td>0</td>
<td>64</td>
<td>2020</td>
</tr>
<tr>
<td>cbor.o</td>
<td>10137</td>
<td>0</td>
<td>256</td>
<td>10393</td>
</tr>
</tbody>
</table>

An investigation of the compression module files show results indicating that the executable file of the CBOR implementation requires significantly more storage space compared to the arithmetic coder, almost 5 times as much space, which can be observed in Table 4.8. The larger size segment is primarily attributed to the _TEXT section, which contains the executable code. This gives a indication that the CBOR implementation requires more storage space as it has more extensive instructions than AC. This is understandable, as the CBOR implementation needs to iterate the data structure and encode each of its field independently, while AC consists of a compact algorithm.

Table 4.9: Original storage requirements (in bytes) of the executables for the TCP client and server codes.

<table>
<thead>
<tr>
<th>File</th>
<th>_TEXT</th>
<th>_DATA</th>
<th>others</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp-client.o</td>
<td>1774</td>
<td>616</td>
<td>96</td>
<td>2486</td>
</tr>
<tr>
<td>tcp-server.o</td>
<td>993</td>
<td>416</td>
<td>96</td>
<td>1505</td>
</tr>
</tbody>
</table>

Table 4.9, provide insights into the original storage requirements of the TCP client and
server executables before any change is made for the TCP socket to manage the compression of the data being sent. Looking at the table, it is notable that the storage requirements are relatively small, which indicates that the TCP client and server executables have compact storage requirements.

Table 4.10: Comparison of the storage requirements (in bytes) of the executable for tcp client and server codes, depending on compression methods.

<table>
<thead>
<tr>
<th>File</th>
<th>Compression</th>
<th>_TEXT</th>
<th>_DATA</th>
<th>others</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp-client.o</td>
<td>None</td>
<td>2386</td>
<td>27746</td>
<td>96</td>
<td>30228</td>
</tr>
<tr>
<td></td>
<td>AC</td>
<td>2753</td>
<td>36666</td>
<td>96</td>
<td>39515</td>
</tr>
<tr>
<td></td>
<td>CBOR</td>
<td>2739</td>
<td>36666</td>
<td>96</td>
<td>39501</td>
</tr>
<tr>
<td>tcp-server.o</td>
<td>Any</td>
<td>2084</td>
<td>18232</td>
<td>96</td>
<td>20412</td>
</tr>
</tbody>
</table>

Comparing the AC and CBOR compressed versions, it can be observed in Table 4.10, that both methods result in a similar total storage requirement for the “tcp-client.o” file, with the “tcp-server.o” file not being affected depending on the method used. Although the sizes of the individual sections vary slightly, these variations do not significantly impact the overall storage of the file. However, It is worth noting that both compressed versions require more storage compared to the uncompressed version, indicating the additional data introduced by the compression methods.

Comparing the size difference between the original storage requirements of the TCP socket files and the files after additional code for the compression handling, one can observe a significant increase in global and static variables. Looking closer at the storage distribution, most of the storage in the _DATA section falls under the subsection __bss.

The increase in storage can be primarily attributed to the incorporation of additional variables, such as the “packet_stats” structure, which alone consumes nearly 9kB of data. Furthermore, The “traffic_monitoring” structure holds the size of the “packet_stats” structure and the header size, resulting in an additional 9kB of data. Another notable contributor is the buffer designated for storing the transmitted data, its size is determined by the “traffic_monitoring” structure. These variables are modifiable and uninitialized, serving the role of data storage for transmission over the socket.

The introduction of these variables results in a substantial increase in the storage requirements of the TCP socket, particularly within the _DATA section. The added data plays a crucial role in facilitating compression handling and the efficient transmission of data over the socket, as they hold the data to be transmitted.
Chapter 5

Conclusion and Future Work

From the analysis carried out in this report, several significant discoveries are emphasized. The assessment centered on evaluating the efficiency of the compression modules through the examination of compression ratios, execution duration, memory usage, and storage capacities of the implementations. The experiments aimed in investigating the effectiveness of both the general-purpose compression module and the special-purpose compression module on artificially generated traffic monitoring data.

5.1 Results

Regarding compression ratios, the results showed that both compressing modules were effective on data with low information entropy and non-random data, which fits well with the real-life data of the traffic monitoring module in the IoT network. It is also demonstrated that the arithmetic compressor consistently achieved higher compression ratios across a range of structured data with varying characteristics compared to the CBOR implementation.

In terms of execution times, the experiments measured the impact of compression and decompression processes on the overall execution time of TCP socket operations. The results indicated that the arithmetic coding compression method generally exhibited lower execution times compared to the CBOR method, especially for structured data with non-random values. However, as the percentage of structured data filled increased, the execution times for both compression methods also increased. This suggests that compressing structured data, particularly with random values, incurs additional processing overhead.

Lastly, the storage requirements of compressed data were highlighted as an important factor to consider. The experiments demonstrated that by including the compression modules in the TCP socket, the code space increased a lot, which is a result of the usage of uninitialized global variable used for handling and storing the large data. It is shown that the code is effected likewise by the two modules. Looking at the storage requirements of the executable of the compression modules, it is shown that the CBOR encoder implementation demanded about five times the amount of storage space as the arithmetic coder.

Overall, the evaluation results provide insights into the performance characteristics of the compression modules. The arithmetic compressor showcased higher compression ratios, lower execution times (especially for structured data with non-random values), and less storage space compared to the CBOR encoding. These findings suggest that the arithmetic compressor is well-suited for scenarios requiring high compression ratios, real-time or near-real-time data transmission, efficient resource utilization, and reduced storage requirements. The general-purpose compression module is therefore the preferred method for compressing the data transmitted in a sensor network.
5.2 Future Work

A consideration for the project was to explore more compression algorithms and encoding techniques. Combining multiple methods of compression on the data, instead of only executing a singular method could also be interesting to evaluate further. This was examined in a small scale and could be further investigated as it proved intriguing results, where the combination provided a higher compression ratio.

In addition to compression techniques, data transformation techniques can be and interesting aspect to investigate and discuss its potential benefits and implications for future research. The usage of such techniques could potentially reform the data to be more fitting for compression, creating higher compression ratios.

It is important to note that the evaluation experiments focuses on specific scenarios and data. Some of the limitations observed for both compression modules in compressing structured data with a higher presence of random data may not significantly impact real-life scenarios, where data often exhibit patterns, correlations, or specific characteristics. Therefore, It would be interesting to do tests with more realistic traffic monitoring data and networks.
Bibliography


