University of Zurich^UZH

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Secure Data Transmission in Contiki-based Constrained Networks Offering Mututal Authentication

Severin Siffert
Zurich, Switzerland
Student ID: 14-720-536

Supervisor: Dr. Corinna Schmitt, Eder Scheid
Date of Submission: 26. August 2018

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

With the Internet of Things growing very rapidly and device diversity exploding, security solutions are not always keeping pace. In order to get to security by default, encryption has to become more efficent and easier to use. TinySAM is a security solution for constrained sensor networks and offers confidentiality and mutual authentication. It uses AES and a key server to dynamically establish sessions between any two peers that are allowed to communicate with each other, thus supporting in-network aggregation of data. An implementation already exists for the TinyOS operating system. This work is an improvement of it, called sTiki (short for *secure Contiki*) and implemented for the Contiki operating system. The implementation is done under Contiki 3.0 and runs on OpenMote hardware, which already uses the TinyIPFIX protocol to collect data. The key server implementation is in Java, integrated into CoMaDa, an interface to easily manage nodes and collect data with sensor networks.

Zusammenfassung

Wegen des sehr schnell wachsenden Internet of Things und der explodierenden Gerätediversität können Sicherheitslösungen nicht immer Schritt halten. Um standardmässig sichere Kommunikation zu haben, muss Verschlüsselung effizienter und einfacher einzubauen werden. TinySAM ist eine Sicherheitslösung für eingeschränkte Sensornetzwerke und bietet Vertraulichkeit sowie gegenseitige Authentifizierung. TinySAM benützt AES und einen Key Server, um dynamisch Sitzungen zwischen zwei beliebigen Partnern, die zusamen kommunizieren dürfen, aufzubauen. Dies ermöglicht es, bereits im Netzwerk die Daten zu aggregieren. Eine Implementation besteht bereits für das Betriebssystem TinyOS. Diese Arbeit ist eine optimierte Implementation namens sTiki (kurz für *secure Contiki*) für das Betriebssystem Contiki. Diese Version wurde für Contiki 3.0 geschrieben und läuft auf OpenMote-Geräten, die bereits das TinyIPFIX-Protokoll beherrschen, um Daten zu sammeln. Der Key Server ist in Java geschrieben und in CoMaDa, einer Schnittstelle für Knotenmanagement und Datensammlung mit Sensornetzen, eingebaut.

Acknowledgments

First, I would like to thank my supervisor Dr. Corinna Schmitt for her continuous support and valuable feedback. I also would like to thank my co-supervisor Eder Scheid for his help with technical problems and his great eye for details. Last but not least, I would like to thank Professor Dr. Burkhard Stiller, head of the Communication Systems Group at the University of Zurich for supporting the realization of this Bachelor thesis.

Contents

Chapter 1

Introduction

Due to the growth of the Internet and the device diversity, the Internet of Things (IoT) is gaining a lot of attention. IoT used to be to limited to Peer-to-Peer (P2P) networks and devices such as servers, computers, and routers. However, nowadays, it also includes wireless sensor devices that form an individual network, called a Wireless Sensor Network (WSN). Those devices present a challenge for developers because they are limited in memory, energy, and computational capacities. In order to connect them with the Internet, they must support IP communication, which is often realized using an IPv6 implementation called 6LoWPAN. [33]

The topology of WSNs can range from star topologies to pure-P2P topologies, but a combination of both topologies is employed in WSN deployments. This means that the network will consists of Full-Function Devices (FFD) and Reduced-Function Devices (RFD), both types can support different functionalities depending on the location within the WSN. This functionality can range from simple data collection and forwarding to pre-processing. The communication between the devices in a WSN is performed wirelessly and over UDP. Furthermore, the transmitted packet size is limited (e.g., 127 Bytes). However, due to existing IPv6 implementations, like 6LoWPAN, it is possible to support packet fragmentation and compression in order to connect such limited devices to the IoT. [15]

Many use cases for IoT involve the collection and transmission of sensitive data. Yet, many deployments currently do not protect this data through suitable security schemes [30]. Different end-to-end security schemes were build upon existing Internet stanards, specifically the Datagram Transport Layer Security (DTLS) protocol, but might not be applicable to WSNs due to the use of constrained devices with especially limited memory resource. By relying on an established standard existing implementations, engineering techniques, and security infrastructure can be reused that enables easy security uptake from application developers. [17]

1.1 Problem Statement

The challenge of this thesis is to bring standard compliant security to very resource constrained sensor nodes in an end-to-end security architecture. The intended solution must satisfy the paradigm of end-to-end security, should be standard based, and support mutual authentication (e.g., [17, 21, 22]). The intended solution will be based on symmetric cryptography and dynamically establish sessions between pairs of nodes in the network, which are used to protect later data transfer. The sessions will only be established between nodes that actually need to communicate with each other and, thus, no prior knowledge of the information flow in the network is necessary. It must be taken into account, that the designed and developed solution must be able to process in parallel the basic functionality of the existing implementation – gathering data, transmitting it using TinyIPFIX [29] format, and support aggregation in the network [31, 32]. Further the solution requires to be flexible concerning network updates (e.g., node addition or deletion) in order to establish efficient and secure communication ways in the network.

1.2 Thesis Outline

The rest of this thesis is structured as follows: In Chapter 2, this work's context is explained. Chapter 3 covers the constraints of developing software for sensor networks, the chosen architecture and the choice of AES. Chapter 4 discusses the implementation of sTiki in the two environments. Afterwards, in Chapter 5 the implementation is evaluated and compared to the existing implementation. Finally, in Chapter 6 conclusions are drawn and an outlook for possible future work is presented.

Chapter 2

Background

This chapter covers the most important technologies encountered in this work. First, Wireless Sensor Networks (WSNs) are introduced, showing the technological restrictions of the problem space. Afterwards, the utilized hardware and operating system is introduced. As a requirement for this thesis is to support the TinyIPFIX format used for efficient data transmission a brief explaination is presented. Further the existing security solution TinySAM implemented for constraint devices under another common operating system named TinyOS [19] is presented together with other selected lightweighted security solutions (e.g., MiniSec [22], TinyDTLS [17]) to which the developed solution will be compared to.

2.1 Wireless Sensor Networks

A WSN is a network consisting of just a few or up to thousands of small, very limited computers, normally called *nodes*. The nodes usually are battery powered and their memory is very limited, often measured in kilobytes. Their purpose is to gather data, for example about temperature, humidity or movement of certain objects. Due to the limited memory and and the high probability of failure, normally the nodes send collected data to a device (or device combination) called *sink* or *gateway*, which is more reliable and has more power and memory available (depicted in Figure 2.1). When a node is too far from the gateway, another node can forward the message to another node or to the sink. In certain use cases it is even feasible that a node processes the data (e.g., filtering out measurements that are not needed at the moment or computing an average value) in order to send less bytes or to send more useful data. The Aggregators in Figure 2.1 do exactly that, whereas the Collectors only produce measurements. Because the nodes are constrained in resoures, they require specialized tools such as the operating system or protocols shown in the following sections. [2, 37]

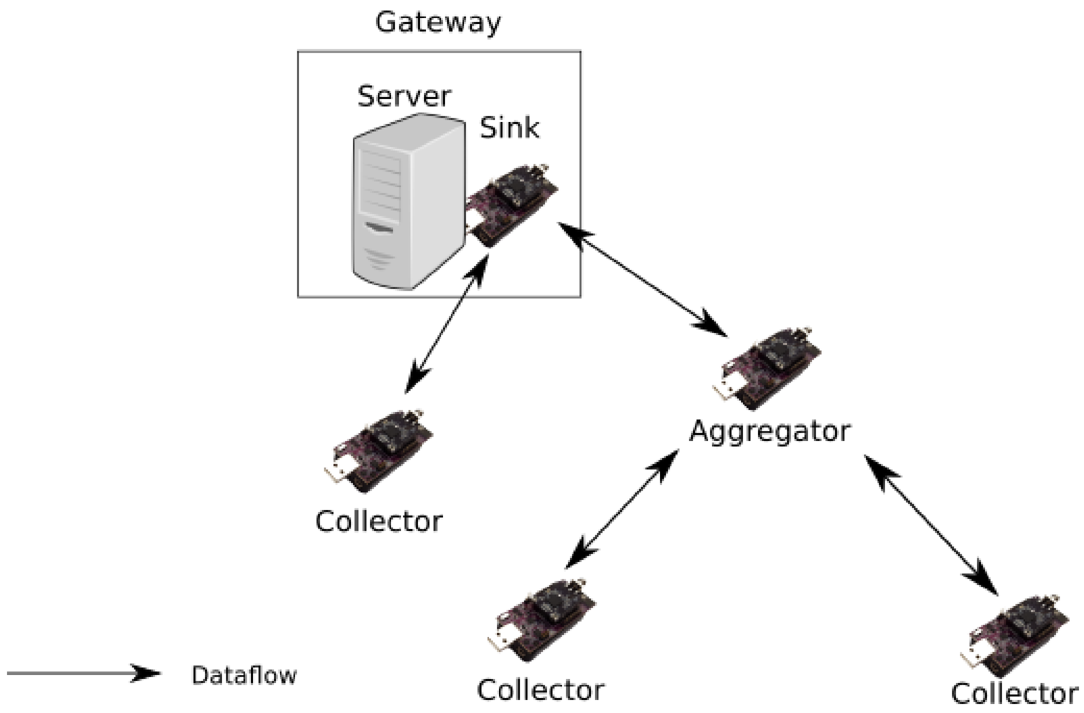


Figure 2.1: A typical WSN setup.

2.2 Hardware Platform

In this work, **OpenMote**s were used as constrained devices to build the network. They are very small and are a combination of the parts OpenMote-CC2538 Rev.E and OpenUSB Rev.B. The OpenMote-CC2538 mainly consists of the CC2538 processor from Texas Instruments, which has an ARM Cortex-M3 microcontroller with 512 KB storage and 32 KB RAM. It also has hardware implementations for AES-128 and AES-256. The OpenUSB has a USB port, space for two AA batteries and sensors for light, temperature, humidity, and acceleration on three axes. [32, 35, 36]

2.3 Operating System

Specialized operating systems exist for WSNs. Because the nodes have very limited Random Access Memory (RAM), Read-only Memory (ROM) and battery power, running Windows or Linux on them is either impossible or does not make any sense. Various minimal Operating Systems (OS) exist, for example Contiki [10], TinyOS [19], LiteOS [6] or MantisOS [5]. As Contiki is used in this thesis it is briefly characterized. For the others, please refer to the given references. Contiki is an open source minimal OS written in C, that was created by Adam Dunkels in 2004. It has a modular architecture which is built on top of an event-driven kernel [10]. Whenever an event is triggered, it runs to completion, but can be preempted if necessary. To keep the size as small as possible, threads are only implemented as a library that is included when needed [12]. In the same spirit of saving resources, Contiki offers no way to synchronize internal clocks, which makes it impossible to implement a multitude of protocols. Contiki can dynamically load and unload code, which allows to change the running code remotely and without recompiling [10].

Because Contiki is extremely constrained, there are only two ways to start code execution: code can run once the node has powered up or in reaction to an event happening. The most common event sources are a timer running out and the arrival of a packet. This event-driven nature of Contiki has a big impact on the coding of applications. The impact on this work is limited because an encryption protocol mostly works in response to requests of the application it supports. Those requests are usually to encrypt and send a message or to decrypt an incoming one.

2.4 TinyIPFIX

IPFIX is a protocol to transmit flow information in a network, specified by the Internet Engineering Task Force (IETF) in RFC7011 [8]. It is a push-based protocol where the sender separately sends *data records* and *template records*. The template records define what is contained in the data records, thereby eliminating the need to include what the data is in every data record. TinyIPFIX [28, 29] is an adapdation of IPFIX for wireless sensor networks. Compared to IPFIX, it omits a lot of information that is not used in most WSN contexts, thereby reducing the size of the packets.

The utilized implementation of TinyIPFIX also supports in-network data or message aggregation. Being able to aggregate the measurements is crucial to sending less data. This is important because sending data is one of the most power-intensive tasks a node can perform [21]. Depending on the desired form of aggregation, it is possible to save a substantial amount of messages and energy by, for example, sending the average temperature of an entire room over two minutes instead of sending five nodes' measurements every ten seconds to a far away sink. If every measurement has to arrive at the sink, it is at least possible to perform message aggregation. Message aggregation works by combining multiple payloads into one, which works because TinyIPFIX has a constant overhead per message, regardless of payload size. Before implementing TinyIPFIX into an application it should be considered whether or not it actually makes sense to use TinyIPFIX. When it takes more power and time to aggregate the measurements than to simply forward them to the sink, then TinyIPFIX should only be deployed if network congestion is a bigger problem than the additional power requirements.

Both forms of aggregation require that the messages can be read by the aggregators. If they cannot (e.g., due to encryption), the aggregators are no longer able to take advantage of aggregation. Being able to decrypt traffic on the aggregator nodes is therefore a key requirement for the encryption protocol discussed in the next section.

2.5 Encryption

As described in [21], there are four properties a system can gain that are related to encryption, but not all might be needed or desired.

Authentication: The identity of the sender can be determined with certainty.
Data Integrity: The recipient can be sure that the data has not been tampered with.
Freshness: The recipient can be sure that he is not receiving a copy of a previous message.
Confidentiality: Only the intended recipient can read the message, but noone else.

Authentication and data integrity can both be achieved by using a cryptographic checksum—commonly called Message Authentication Code (MAC). Freshness usually is provided by timestamps (which require a global clock), nonces or counters. Confidentiality is achieved by encryption.

One major choice is between symmetric and asymmetric encryption as there are arguments in favour of both. As in many decisions with WSNs, this one too is between efficiency and security. Asymmetric ciphers win in security, but take significantly more resources [23]. For example, ECC (asymmetric) is about 100-1000x slower than AES (symmetric) [11, 25]. For the planned use cases, symmetric will be fine and also allow a better comparison to [21].

The next decision is between block and stream ciphers. Block ciphers encrypt blocks of a fixed size whereas stream ciphers encrypt data of arbitrary length. Research about lightweight cryptography focuses mainly on block ciphers because they can also be used for computing MACs, and if necessary can be turned into stream ciphers by using CBC (Chain-Block Chaining) or Counter mode [23]. As a result of this, most high quality lightweight ciphers are block ciphers and when using one, the implementation of an additional cipher for computing MACs can be omitted, which is also why a block cipher is used here.

[23] compared many lightweight symmetric block ciphers for various criteria. In addition to the ciphers compared in [23], a new ultra lightweight block cipher called QTL [20] was considered. However, QTL was quickly dropped because [9] and [26] showed it to be insecure. Lightweight ciphers are a specialized category of encryption algorithms that try to find a good tradeoff between resource consumption and security. They are commonly used in WSNs, WBANs (Wireless Body Area Networks) and other medical devices [23]. Especially in implanted devices power consumption is a big concern [23]. Because of the various security requirements of different applications (compare for example controlling a pacemaker and a home temperature monitoring system), there is a place for many different algorithms or smaller variations thereof.

Comparing different ciphers is no easy task. There is no universally accepted metric for most possible measurements and security is no well-defined term [23]. In addition, new and innovative ways of attacking highly rated algorithms might be discovered and render the ratings useless. Even a metric that does not consider security like *efficiency* (e.g., defined as energy required per encrypted byte) fails at being a fair comparison because it can be gamed in various ways too [3]. Luckily, AES shows up close to the top for almost all used metrics in [23] and [25], which makes it a relatively obvious choice. Its only problem is the relatively big memory footprint. But because AES is very popular, many boards (including OpenMote) have a hardware-implementation of AES, which requires almost no additional memory. The most popular implementation is AES-128. Based on the results of [23], Tea/xTea would have been the second choice.

Even though the parameters of the hardware implementation of the OpenMote cannot be changed, it is still worth looking at them and their impact on energy consumption to judge potential alternatives. The main parameters of block ciphers are key-size, block-size and the number of rounds performed [23]. The increase in energy consumption is roughly linear for key-size and the number of rounds [25]. The main parameters, however, do not have the biggest impact on energy consumption. Instead, the mode of operation has a two to three times bigger impact [25]. This is because the different modes have different procedures to determine the key for the next encryption step, which can be a substantial effort.

2.6 TinySAM

This section covers the encryption protocol TinySAM [21]. First, a general overview is given, followed by more detailed information about the common header, the used encryption mode and error recovery. The protocol is explained in detail because sTiki relies heavily on TinySAM and mostly uses the same methods and mechanisms.

2.6.1 General Overview

TinySAM [21] is an application layer encryption protocol that uses any symmetric encryption (here and in [21], AES-128 is used) and a key server. There are multiple reasons for those choices. It is implemented as an application, running below other applications, because this makes it as platform independent as possible. The protocol would support asymmetric cryptography with only minor adjustments, but asymmetric cryptography uses significantly more RAM and ROM than symmetric cryptography [14], even though it would offer more security [23].

Using a key server is a tradeoff solution. Having a single key for the entire WSN is very susceptible to node capture and can compromise the network in its entirety. Because of that, every link should be encrypted with its own key. But storing the keys for interacting with every other node requires a lot of ROM, renders key distribution extremely complicated, and makes adding new nodes to the network very expensive [16]. Using a key server is not without problems either, because it forms a single point of failure and thus is a weak point [4], but in the planned use cases with a single sink node, the same problems exist, even if the key server was not used.

2.6.2 Common Header

Each TinySAM packet starts with a common header. The header may begin with a magic number, which serves to discriminate between packets that are encrypted with TinySAM from ones that are not. This is necessary in networks that have nodes that have no support for TinySAM. Also in the header is a number specifying the (TinySAM-internal) protocol (e.g., handshake, data transport, or alert) and sub-protocol (e.g., what type of alert) a packet should be forwarded to.

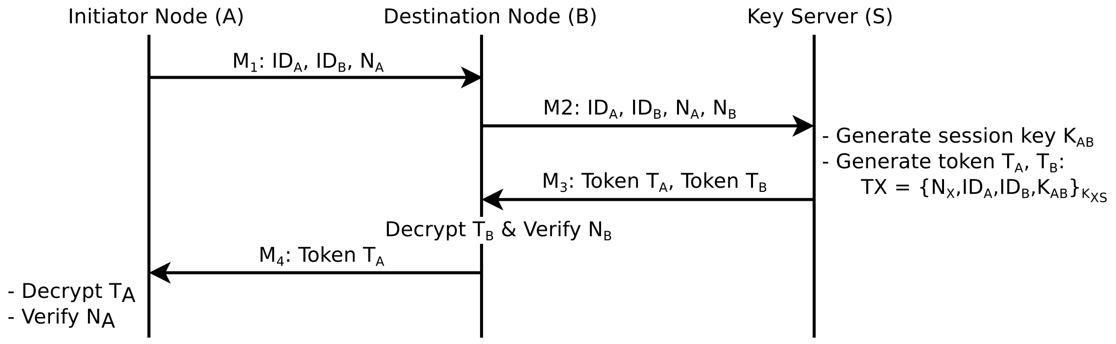


Figure 2.2: The messages sent in the ANOR handshake [21].

2.6.3 Session Keys and Message Encryption

To establish the session key, the ANOR-protocol (shown in Figure 2.2) is used. ANOR (AN Otway-Rees) [1] is an improvement over the Otway-Rees key establishment protocol [24]. Each node has a key it uses to communicate with and authenticate itself with the key server. To begin the handshake, Node A sends the first message M_1 (containing ID_A and a nonce N_A) to B. B forwards this message, ID_B and nonce N_B to the key server in the second message, M_2. If both nodes belong to the network and may communicate with each other, the key server generates a session key K_{AB}. Then, the session key is once encrypted with A's initial key K_{AS} and once with B's initial key K_{BS}. Both versions are sent to B in M_3, which decrypts the session key with its own initial key. To conclude the handshake, B sends M_4 with the encrypted session key to A. By using B as an intermediate station to send the key to A, A only receives the session key *after* B has received the session key. This proves to A that the handshake was successfully completed and signals that buffered messages can be sent securely, using the just established session key.

When working with static keys, sending the same data twice results in the same (encrypted) message being sent twice. This allows attackers to draw conclusions from repetition patterns. To protect against such an attack, counter mode is used for encryption with a changing initialization vector (IV), resulting in a new key for every message. Because sending data draws a lot of power, not the whole initialization vector is transmitted with every message. Instead, certain bytes of it act as a message counter and only this counter is sent along with the payload. This counter can also be used to prevent replay attacks, where an attacker captures a message and later sends it again. This will be detected because the counter is not increasing from one message to the next. To ensure the data (for example the message counter) has not been tampered with, a MAC is added to almost all messages. To calculate the MAC, the session key has to be known and changing only one bit will result in a completely different MAC, making tampered or incorrectly transmitted data easy to detect.

The way TinySAM uses encryption results in all four properties being covered: using MACs produces Authentication and Data Integrity, the counters offer Freshness and encrypting the data during transmission achieves Confidentiality. TinySAM does not prescribe a specific cipher. With some minor modifications, even asymmetric ciphers would be feasible.

2.6.4 Error Recovery

In TinySAM, there are two common exceptional states. In the first case, a node has lost the initialization vector or has detected a packet with the wrong message counter. In that case, it instructs the other node to send a new initialization vector along with the next message. The other problem occurs when the session key was lost on a node, most likely because it lost power for a short amount of time or because it crashed and had to reboot. If the node sending the message has lost the key, it will initiate a new handshake. The other node will simply overwrite the old session data. If the receiving node has lost the session key, it will relay the command to have the other node initiate a new handshake via the key server. This message is sent via the key server because the key server is the only node that has a key to communicate securely with the other node. If the instruction would not need to be secured, then a malicious node could premanently send 'missing session' alerts in the name of any node in the network, shutting down any communication.

2.7 MiniSec

MiniSec [22] is a security solution for WSNs with a version for unicasting and a version for broadcasting messages. Its main goal is low energy consumption, but as little compromise in security as possible. In MiniSec, each node pair shares two encryption keys—one for each direction. This requires that the network layout is already known in advance and does not allow new nodes to be added easily. The encryption is done with *Skipjack* in *Offset CodeBook* mode, which has the advantage of only requiring one pass over messages to produce the cipher text as well as the integrity protection, thereby saving a lot of expensive encryption calculations. Message counters are included in the encryption keys to produce differing cipher texts even when sending the same content repeatedly and to protect against replay attacks. [22]

MiniSec achieves all four features discussed in the Section 2.5 about encryption: Confidentiality is provided by using encryption, Authentication and Data Integrity are a result of integrity protection and Freshness is achieved by using counters that alter encryption keys and initialization vectors. The main disadvantages of MiniSec are the pre-shared keys and its integration in the networking stack of the operating system. The pre-shared keys require a lot of planning before deploying and no easy way to switch out a single node in a network. The deep integration into the operating system makes it very convenient to use once it is set up, but the implementation is difficult and very hard to reuse in case it should be ported to a different operating system.

2.8 TinyDTLS

TinyDTLS [17] is an implementation of the Datagram Transport Layer Security (DTLS) protocol on TinyOS for OPAL nodes with a Trusted Platform Module (TPM). The DTLS protocol is an adaption of SSL/TLS, altered to support unreliable communication such as

User Datagram Protocol (UDP). In DTLS, the communication partners *can* authenticate each other, but do not have to. To do so, they present their X.509 certificates to each other, which will be verified by the certificate authority. Authenticity, therefore, is voluntary, but can be forced by the application. Confidentiality is provided by encrypting the payload and integrity is guaranteed through the use of MACs. [17]

The fact that DTLS is a standard protocol is a huge advantage for interoperability with other systems. But this compatibility has a steep price: The overhead (compared to TinySAM [21] and MiniSec [22]) is very big and messages are even padded because DTLS only works with block ciphers. The next expensive part is in code size and/or hardware support: The handshake uses RSA encryption, the payload encryption AES-128, the MAC computation SHA1, and to securely store the certificate, a TPM is required. Implementing and performing three different ciphers is very expensive for sensor nodes, which makes DTLS only useful for nodes with plenty spare resources. Finally, introducing a certificate authority adds a single point of failure to the system, which can be a high risk when working in harsh environments or with severely limited power supply. The cost of generating, storing and distributing the certificates should also not be disregarded. The implementation of TinyDTLS uses about 20 kilobytes of RAM and 67 kilobytes of ROM, which is about ten times the amount TinySAM uses (RAM and ROM) or four times the amount of ROM and 25 times the amount of RAM used by MinySec [17, 21, 22].

Chapter 3

Design Decisions

This section covers how sTiki is composed. In the section Protocol Description is the description of the sTiki protocol. The section Use Cases explains for which situations sTiki is designed and when it should not be used. Finally, in the sections Architecture and Memory Constraints, the platform will be examined to decide about coding style and tradeoffs that will be a theme at some point during implementation.

3.1 Protocol Description

sTiki is mostly inspired by TinySAM because the requirements fit very well. TinySAM covers end-to-end security, is standards based and supports mutual authentication. It uses the requested symmetric encryption by default and can dynamically establish sessions between the nodes. The requirements about network updates and supporting in-network aggregation are also fulfilled. Therefore sTiki is an improvement over TinySAM and works in a very similar fashion.

The following decisions were made to adapt TinySAM:

- The magic number in the header is included.
- As discussed in the section about encryption, AES-128 was chosen for encryption.
- The number of ongoing handshakes per node is limited to only one.
- A failed handshake will not be retried.

The magic number in the header serves to discriminate sTiki packets from ones that are not encrypted, thereby allowing mixed networks with nodes that do not support sTiki, for example because they do not have enough resources left over by operating system and running applications.

The maximum concurrent handshakes were limited to just one because they each take up 50 bytes plus the overhead of searching through them during handling. The cost of not being able to set up an entire network at once is also very limited when the handshake timeout is set to a reasonable timeframe. Handshakes also are not retried in case of a handshake timeout because it is deemed more useful to retry with a more current message. Otherwise it is possible that a node will try to send a measurement for multiple hours instead of trying to send a current measurement. Also, when continuing to retry, it is possible that the destination node malfunctioned and will never be reachable again.

3.2 Use Cases

Since sTiki uses a key server to manage all communication lines, it suffers from having a single point of failure. Because the key server is only used to establish a secure connection, a key server failure does not mean an immediate, complete failure of all communication, however, once connections start to fail (e.g., due to lost keys from rebooting), transmitting data to the sink becomes impossible.

Having a single point of failure does not mean that a protocol is useless. WSNs commonly contain only a single sink node and the whole network fails if the sink fails in some fashion. If the sole sink node is the key server (or the only connection to it), it is the same single point of failure and no additioal problems are introduced by implementing sTiki.

Another possible solution is to extend sTiki to allow multiple key servers. This is feasible because any procedure involving the key server requires only a single message to and from it, without considering any dynamic information. Having mutliple key servers requires no communication or awareness between them because they only need to checke whether a node is part of the network and whether the two nodes are allowed to communicate with each other, which is little enough information to keep synchronized. The extension would encompass cycling through key servers on handshake timeouts and maybe using different keys for different key servers.

If passing information around as quickly as possible is a requirement (e.g., firefighters deploying nodes to collect information about a fire), the time it takes to perform a handshake probably makes sTiki useless. The problem in this case is not that there are too many messages passed around but using UDP and a timeout to restart the handshake could mean that time is wasted by waiting for a message that will never come. Additionally, time can be saved by using a handshake that does not involve a key server which might be many hops away. For such a scenario, it would make more sense to have a shared key for the entire network which saves the entire handshake but trades long-term security for it.

3.3 Architecture

sTiki is written in a modular fashion for multiple reasons. Contiki itself is written in modular style so the style is kept consistent. Second, having modules makes it easier to switch out parts (such as the encryption algorithm) or make small changes if it should

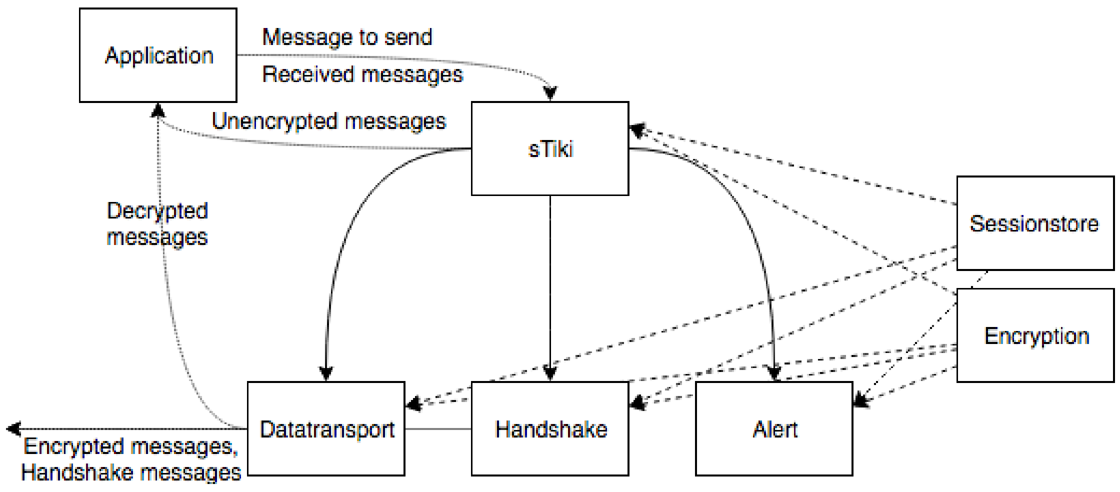


Figure 3.1: The chosen architecture of sTiki.

become necessary. Last, a modular architecture tries to keep code local, which makes it significantly easier to maintain, adapt and test.

As depicted in Figure 3.1, the starting point for every use case is the *sTiki* module. When receiving messages, its job is to filter out packets that are not in sTiki format and to redirect the messages to the appropriate protocol. It does that by acting as a dispatcher and delegates the handling to the correct module. When sending messages, the main module does the necessary checks to determine whether a handshake has to be initiated or the data can be sent and, again, delegates this action to the responsible module.

In addition to the main module and the protocol handlers, there are the helper modules such as the *session store* or the *encryption* module. Those are used in most source files because they perform common critical functions or carry state information that is used throughout sTiki. In Figure 3.1 they are presented on the side because they are used by almost everything.

3.4 Memory Constraints

A big part of working with WSNs is juggling resources. The limited RAM and ROM makes it necessary to keep in mind how much of the two resources is still available on the devices as many decisions favour RAM, ROM or power savings for the cost of taking more of one or both other resources. An example are the fixed keys in each node: In sTiki, each node has a master key from which two keys are derived in order to secure the communications with the key server. One for encryption purposes, the other one to compute the MACs. It is either possible to store the master key in ROM and derive the two keys on startup, or they can be precomputed and stored in ROM so the power to compute them can be saved on the expense of storing double the amount of keys. If the node has very little ROM left, it probably will be decided to spend the energy to recompute the keys.

TinySAM requires a rather sizeable chunk of ROM and RAM. In [21], TinySAM required about 6.5 kilobytes of ROM and 1.5 kilobytes of RAM, which makes the entire code running on the nodes 15–20% bigger, which can be a problem on certain devices. It should be noted that the implementation also contained the AES encryption algorithm, which is not necessary on devices with a hardware implementation. Additionally, some variations should be expected since different programming languages and programming styles can produce fairly different compiled code.

The implementation for the OpenMote need not be too concerned with resources. After the implementation of TinyIPFIX in [32], there were 13.3 kilobytes of RAM and 462.5 kilobytes of ROM left over.

Chapter 4

Implementation

This chapter covers the implementation of sTiki, first for the nodes and then the key server. For the key server, managing the master keys was not implemented because this functionality already exists in a different version but has not yet been ported over to the used version.

4.1 Nodes

The interface for other modules to use is in `stiki.h`. It is a Facade [13] to make interacting with sTiki simpler. It contains only the four methods one has to understand to use sTiki and is commented heavily. The decision to create a facade was made because a user of sTiki only needs to know about those four methods, whereas `stiki-main` contains too much detail that is not important or even confusing for users of sTiki. The four methods in `stiki.h` are used as follows:

On system startup, two methods have to be called when using sTiki: `init_stiki()` and `set_stiki_packet_handler(packet_handler ph)`. This allows sTiki to set up everything necessary (e.g., the `handshake` module has to reset the handshake status to `IDLE` and register the connection to the key server. See `handshake.c`, line 13) and tells it what method to use when receiving data for the application. `stiki_udp_sendto(...)` is the securer replacement for `simple_udp_send_to(...)` and encrypts the data before sending. It transparently handles handshakes and error conditions. It is, however, not guaranteed that the message will arrive because the whole protocol is built on top of UDP. Finally, `stiki_receive(...)` is the function to be registered as a callback when setting up connections over which one plans to send data with `stiki_udp_sendto`. Listing 4.1 shows an example how to set up sTiki. After those steps are complete, messages can be sent with `stiki_udp_sendto`.

The core implementation of sTiki resides in `stiki-main.c` and `stiki-main.h`. Any action involving sTiki passes through it, either by using it to send something (`stiki_udp_sendto`) or by using it to receive a packet (`stiki_receive`). Both methods act as a dispatcher to the correct submodule, for example the `datatransport` or `alert` module. The receiver

```
init_stiki();
//tell sTiki to call receiver(...) when receiving a packet for the application
set_stiki_packet_handler(&receiver);
//register stiki_receive(...) as message processor
simple_udp_register(&connection, 0, NULL, UDP_PORT, stiki_receive);
```

Listing 4.1: An example on how to set up sTiki.

simply directs the packet to the correct submodule whereas the sender first has to check for existing sessions or potential errors to decide to which submodule the packet has to be directed to.

The files `stiki-main.h` and `stiki-main.c` also contain structs and their helper functions that are used throughout sTiki in addition to some configurations. Looking back, it might have been more useful to extract those into its own file, thus making `stiki.h` unnecessary. The configurations consist of useful constants used throughout the implementation (e.g., encryption key length or the size of nonces) and the user-defined values (e.g., when a handshake times out or how many sessions a node can keep active at a time).

When receiving a packet in `stiki_receive`, it should only be processed by sTiki if it actually is an sTiki packet. This is done by looking for the magic number as the first byte of the received data. If the packet is meant for sTiki, it has to be processed by the correct submodule. The modules `alert`, `datatransport` and `handshake` are reached by a function `handle_<modulename>(...)`, which is called by the dispatcher (see Listing 4.2) in `stiki-main` when a fitting packet was received. It decides which module to use by examining the packet header.

4.1.1 Handshake

The module `handshake` has two responsibilities: Initiating a new handshake and responding to handshake packets. Those duties are performed by the functions `initiate_handshake(...)` and `handle_handshake(...)`. The handshake works according to the ANOR protocol as described in [21] and shown in Figure 2.2.

To keep track of the state of an ongoing handshake, a struct named `stiki_handshake_context` is used. Because sTiki only supports one handshake at a time, there is just one such context stored at any time. It contains the following information:

`state` Keeps track of the progress. It is used to make sure that no messages are skipped and to determine if a new handshake can be initiated.

`target_id` The id of the handshake partner. It could be calculated from the partner's IP address for every use but it is used so frequently that it makes sense to keep it in memory.

```c
void stiki_receive_main(struct simple_udp_connection *c,
        const uip_ipaddr_t *sender_addr,
        uint16_t sender_port,
        const uip_ipaddr_t *receiver_addr,
        uint16_t receiver_port,
        const uint8_t *data,
        uint16_t datalen) {

       if(!is_stiki_packet(data)) {
           //handle data received
           (*data_processor)(c, sender_addr, sender_port, receiver_addr,
               receiver_port, data, datalen);
       }

       uint8_t protocol = read_protocol(data);
       //dispatch message to correct protocol handler
       switch(protocol) {
           case PROTOCOL_DATA_TRANSPORT:
               handle_data_transport(c, sender_addr, sender_port,
                   receiver_addr, receiver_port, data, datalen);
               break;
           case PROTOCOL_HANDSHAKE:
               handle_handshake(c, sender_addr, sender_port, receiver_addr,
                   receiver_port, data, datalen);
               break;
           case PROTOCOL_ALERT:
               handle_alert(c, sender_addr, sender_port, receiver_addr,
                   receiver_port, data, datalen);
               break;
           default:
               ;
               //no correct protocol found
       }
}
```

Listing 4.2: The implementation of the dispatcher in `stiki-main`.

`partner_conn` The (UDP) connection to the handshake partner. sTiki is not responsible for setting up connections to a specific node because it could occupy the same ports the application wants to use. Therefore, the connection the user wants to send the packet with is used for any communication. This variable keeps track of the connection.

`target_ip` The IP address of the handshake partner. Used to address messages to the partner.

`my_nonce` The current handshake's nonce. It is used to make sure the response is not meant for a previous handshake that timed out.

`my_token` The secret part of the last handshake message. It needs to be copied to somewhere because en-/decryption happens in place and the received message may not be altered.

`timestamp` Timestamp of the last activity concerning the ongoing handshake. Used to determine when a handshake has timed out.

`stored_msg` **and** `stored_msg_len` A handshake is initiated when a message is sent to a node with which no session has been established yet. This message is stored until the handshake has completed and is then sent securely. These two variables store this message.

When a new handshake should be started, the function `initiate_handshake` is called from `stiki-main`. This will only work if no other handshake is currently going on. This can be determined by looking at the handshake context's attribute `state` (desired value: `IDLE`) and the timestamp (to check for a timeout). If no handshake is going on, the context is reset and the new handshake is started by sending M1 (see Figure 2.2 for reference).

Received handshake packets are fed into `handle_handshake`, which in turn feeds the message to the appropriate handler. There are handlers for messages M_1, $|M_3$, and M_4. There is no handler for M_2 because M_2 should always be sent to the key server. Once the handshake is completed, the freshly established session will be added to the `sessionstore`, where the session keys, IVs and message counters are stored.

4.1.2 Datatransport

The `datatransport` module is responsible for sending and receiving data for the application using sTiki. For sending data, `datatransport` offers the method `datatrans-port_sendto(...)` and for receiving `handle_data_transport(...)`.

For sending data, the `datatransport` module expects that a valid session already exists. It then fetches the session and, depending on whether or not an IV was already sent, creates a datatransport packet with or without a new IV. The packet is then sent to the appropriate node.

When receiving a packet into `handle_data_transport`, it is not assumed that a session exists. Therefore, it first is checked whether a session exists, and if not, then an alert message is sent to the key server. The key server will inform the sender node of the missing session. If a session exists but the IV is missing (only for packets without an IV included), then an alert message is sent to the sender node so it can include an IV in its next message. If none of these checks failed, the message is decrypted and forwarded to the application using sTiki.

4.1.3 Alert

The `alert` module is responsible for handling alert messages. It makes sure the alert messages are valid and then invalidates the necessary session parameters. Depending on the alert, it can be enough to invalidate the local IV, or when the server reports a missing session, an entire session has to be invalidated.

4.1.4 Sessionstore

Once a handshake is completed, two nodes share a session, of which they keep track of through the module `sessionstore`. The `sessionstore` module stores a predefined amount of sessions and offers ways to access and manipulate them. The amount of sessions stored is defined by setting the compiler flag `STIKI_MAX_SESSION_COUNT`. For the testing setup, only three sessions were stored. Every session occupies 80 bytes of RAM. To ask for a specific session, the other modules simply need to know the ID of the other node. A session contains the following information:

`node_id` ID of the partner node.

`my_iv` The initialization vector used for messages sent *to* the other node.

`remote_iv` The initialization vector used for messages sent *from* the other node.

`crypt_key` The key used to encrypt messages.

`mac_key` The key used to compute the MAC.

`last_event` Timestamp of the last message between the two nodes. Used to determine a session timeout.

Pointers to the sessions are stored in an array named `sessions`. To get a pointer to a specific session, `sessions[search_session_idx(node_id)];` can be used. The following methods are available to interact with sessions:

`search_session_idx` Returns the index in `sessions[]`, -1 if no session exists for the requested node ID.

`is_session_available` Returns whether or not a session with the specified node ID is active.

`update_last_active` Sets the `last_active` timestamp to the current time. Used when interacting with another node to keep the session from timing out.

`create_session` Creates a new session with the specified node ID and returns the index in `sessions[]`. If `sessions[]` has no free slots, the oldest session is discarded in favor of the new session.

`set_session_key` Sets the session key for the specified session and does the key derivation.

`invalidate_session` Sets the session with the specified node ID as timed out so that it can not be used anymore.

4.1.5 Encryption

All cryptography related functions are in the module `stiki-crypto`. For encryption, it uses the AES-128 implementation already present in Contiki, implemented in [18]. The function `encrypt_message` en- and decrypts messages with a given key and IV. It does both de- and encryption because in the used mode of encryption, the encryption operation consists of applying bitwise XOR to the message. When XOR is applied twice, the original input is the result.

The second function the encryption module fulfills is integrity checking. The functions `compute_mac` and `mac_is_valid` compute and verify the MAC of a message according to RFC 4493 [34].

The last responsibility of the encryption module is generating random data. It is used to generate nonces for use in the handshake (`generate_nonce`) and to generate IVs whenever needed (`generate_iv`).

4.2 Server

While the nodes are programmed in C, the server side is in Java, built into CoMaDa [27]. CoMaDa is a modular framework to run on the computer connected to the sink node of the network. It collects the data to allow advanced processing and is capable of exporting the desired information in any form. CoMaDa also allows for easy node management by facilitating compilation and optimizing power usage.

The Server and Node implementation share a lot of surface features: The functionality implemented in CoMaDa is almost the same as on the nodes, plus some key server capabilities. The big picture overview of the code is almost the same: in both versions is a main dispatcher module, a module for each subprotocol, one for encryption and one for session handling. The object-oriented nature of Java makes certain code a lot more comprehensible. Because objects can inherit code from each other, procedural commonalities become visible quite easily. In the case of the main processing function (see Listing 4.3), it becomes clear that a packet only gets processed when it has a valid MAC (lines 14–16). In the C implementation, it is difficult to grasp such information because different protocols are programmed separately.

```
 1 @Override
 2 public WSNProtocolPacket process(WSNProtocolPacket packet) throws
      WSNProtocolException {
 3 
 4    if(!isSTikiPacket(packet)) {
 5       return packet; //not an sTiki packet, do not touch
 6    }
 7 
 8    STikiPacket tikiPacket = STikiPacket.parse(packet);
 9    if(tikiPacket.missesSession()) {
10       AlertPacket.sendInvalidSession(tikiPacket.getSourceId(), 1/*Key server
            ID*/);
11       return null;
12    }
13 
14    if(!tikiPacket.macIsValid()) {
15       return null;
16    }
17 
18    return tikiPacket.process();
19 }
```

Listing 4.3: The main processing function on the server (Logging/debugging statements removed).

4.2.1 Packet Processing

In CoMaDa, packet processing is done by stacking protocols on top of each other. When a packet arrives, it gets parsed into a `WSNProtocolPacket`. This packet then gets fed into the first protocol, in this case already sTiki. Its output then gets fed into the follwing protocols one after another, until the last protocol has done its job. If a protocol returns `null`, the packet will not be forwarded to the next protocol. This is useful when receiving control messages that have no content for the following protocols.

As shown in Listing 4.3 on the last line, the processing in sTiki is done by the function `process()`, returning a `WSNProtocolPacket`. This function manipulates session information and packet contents according to the sTiki protocol and (if it is a datatransport packet) returns a new packet with the decrypted content.

To choose the right `process` function, the correct subclass of `STikiPacket` is determined when parsing the packet (line 8 in Listing 4.3) by looking at the packet header. As part of parsing, the various parts of the packet get split into variables, while also performing correctness checks (e.g., packet size). The `process` function then does the necessary manipulation: For handshake and alert packets, the session information gets updated, for datatransport packets, additionally, the content gets decrypted and returned in a new packet. As an example of session information getting updated (line 11) and data getting returned (line 17), see Listing 4.4.

```
 1 @Override
 2 public WSNProtocolPacket process() throws WSNProtocolException {
 3    if(!getSession().remoteIvIsSet()) {
 4       //send alert MISSING_IV
 5       missingIv();
 6       return null;
 7    }
 8
 9    int dataStart = 4;
10    byte[] iv = getSession().getRemoteIV();
11    getSession().updateLastAction();
12    int messageCounter = ((_payload[2]&0xff)<<8)|(_payload[3]&0xff);
13
14    byte[] encryptedMessage = Arrays.copyOfRange(_payload, dataStart,
          _payload.length-16);
15    byte[] decryptedMessage = AES.ctrCrypt(getSession().getEncryptionKey(),
          encryptedMessage, iv, messageCounter);
16
17    return new WSNProtocolPacket(_id, decryptedMessage, _source);
18 }
```

Listing 4.4: The `process` functino of `DataTransportPacketCtr` (Debugging statements removed).

4.2.2 Utility Classes

Because most operations in the sTiki protocol are heavily repeated in many places, a few utility classes were created. The classes for cryptographic functions are all located in the `Crypto` package:

`AES.java` Responsible for encrypting a single block with AES (function `crypt`) or an entire message in counter mode (function `ctrCrypt`).

`AES_CMAC.java` Responsible for computing the checksum of a message (function `computeMAC`) or checking a MAC for correctness (function `checkMAC`). The algorithm used, AES-CMAC, is described in RFC 4493 [34].

`CryptoUtils.java` Contains functions to generate new random data, to convert byte arrays to strings, and strings to byte arrays. The conversion functions are mainly used for debugging.

`KeyStore.java` Interacts with the file `conf/keystore.properties`, which contains the initial keys for the nodes. Also contains the logic to determine if two nodes may talk to each other.

As part of sTiki, the class `STikiUtils` was created to help with the common tasks of sTiki. It contains functions to convert IP addresses to node IDs and backwards, logging on different granularities, and some common data manipulation like setting a message header.

4.2.3 Key Server Capabilities

The server side implementation is not only a node implementation in Java, it is also the key server. The additional functionality is not very complicated, but it is crucial for the protocol to work. To become the key server, three additional functions have to be performed: The nodes' initial master keys have to be stored, handshake message M_2 has to be processed (thereby allowing handshakes between nodes to be completed), and in case of a lost session key, the key server has to relay the `MISSING_SESSION` message to the right node.

Chapter 5

Evaluation

This chapter first shows that the developed solution works, then evaluates the sTiki implementation by analyzing power and memory consumption, discussing the capabilities and finally compares it to the previously discussed security solutions.

5.1 Testing

While coding it turned out that the chosen architecture makes testing in many places hardly possible. Actions are triggered by a function like `process()` or `handle_packet()`, which then takes the necessary actions without returning much information. Most often, the reaction is sending another message, which is hard to test for.

The few methods suited to testing are tested with the methodology of QuickCheck [7]. QuickCheck is a library originally made for the Haskell programming language which tests methods for mathematical properties. The programmer specifies what properties a method has to fulfill and QuickCheck then generates a couple hundred random cases to check whether those properties are fulfilled. This makes working with assumptions about a method's output easy since it is experimentally proven that those assumptions are correct. Counter mode encryption, for example, as seen in Listing 5.1, should not change the data's length (Line 17) and it should be reversable by applying the same operation again (Line 19). Those properties are checked with a few hundred random cases (Line 5) and each one that fails the test will be noted.

5.2 Proof of Operability

The goal of implementing sTiki was to have all the previous functionality, but having encrypted communications. Because of that, the implementation is barely visible in the application's code and not at all in the interfaces to work with the nodes, as can be seen in the figures 5.2 and 5.3. This sections shows that the messages are actually encrypted for

```java
/*variable declarations omitted*/

//generate test cases
cases.add(new byte[] {}); //test empty case
for(int i = 0; i < TestConfig.RANDOM_CASES; i++) {
   cases.add(CryptoUtils.randomBytes(rand.nextInt(AES.CTR_MODE_MAX_LENGTH)));
}

//perform tests
cases.forEach(new Consumer<byte[]>() {
   @Override
   public void accept(byte[] data) {
      byte[] encrypted = AES.ctrCrypt(key, data, iv);
      byte[] decrypted = AES.ctrCrypt(key, encrypted, iv);

      //encryption does not change the length
      assertTrue(data.length==encrypted.length &&
         encrypted.length==decrypted.length);
      //applying counter mode twice reverts the first time
      assertTrue(Arrays.equals(data, decrypted));
   }
});
```

Listing 5.1: Testing that encryption in counter mode does not change the length and is reversable by applying counter mode encryption a second time.

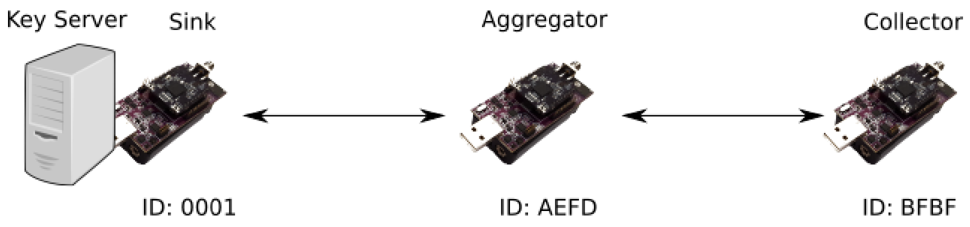
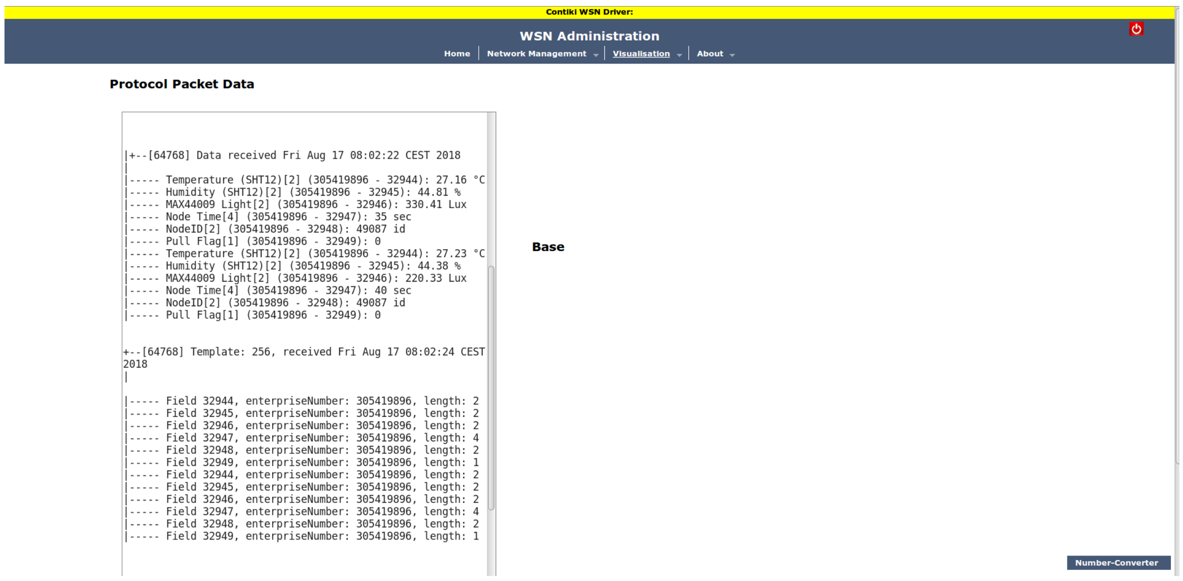


Figure 5.1: Collection setup with an Aggregator and a Collector as used in the Proof of Operability section.



Protocol Packet Data

```
|+--[64768] Data received Fri Aug 17 08:02:22 CEST 2018
|
|----- Temperature (SHT12)[2] (305419896 - 32944): 27.16 °C
|----- Humidity (SHT12)[2] (305419896 - 32945): 44.81 %
|----- MAX44009 Light[2] (305419896 - 32946): 330.41 Lux
|----- Node Time[4] (305419896 - 32947): 35 sec
|----- NodeID[2] (305419896 - 32948): 49087 id
|----- Pull Flag[1] (305419896 - 32949): 0
|----- Temperature (SHT12)[2] (305419896 - 32944): 27.23 °C
|----- Humidity (SHT12)[2] (305419896 - 32945): 44.38 %
|----- MAX44009 Light[2] (305419896 - 32946): 220.33 Lux
|----- Node Time[4] (305419896 - 32947): 40 sec
|----- NodeID[2] (305419896 - 32948): 49087 id
|----- Pull Flag[1] (305419896 - 32949): 0


+--[64768] Template: 256, received Fri Aug 17 08:02:24 CEST
2018
|

|----- Field 32944, enterpriseNumber: 305419896, length: 2
|----- Field 32945, enterpriseNumber: 305419896, length: 2
|----- Field 32946, enterpriseNumber: 305419896, length: 2
|----- Field 32947, enterpriseNumber: 305419896, length: 4
|----- Field 32948, enterpriseNumber: 305419896, length: 2
|----- Field 32949, enterpriseNumber: 305419896, length: 1
|----- Field 32944, enterpriseNumber: 305419896, length: 2
|----- Field 32945, enterpriseNumber: 305419896, length: 2
|----- Field 32946, enterpriseNumber: 305419896, length: 2
|----- Field 32947, enterpriseNumber: 305419896, length: 4
|----- Field 32948, enterpriseNumber: 305419896, length: 2
|----- Field 32949, enterpriseNumber: 305419896, length: 1
```

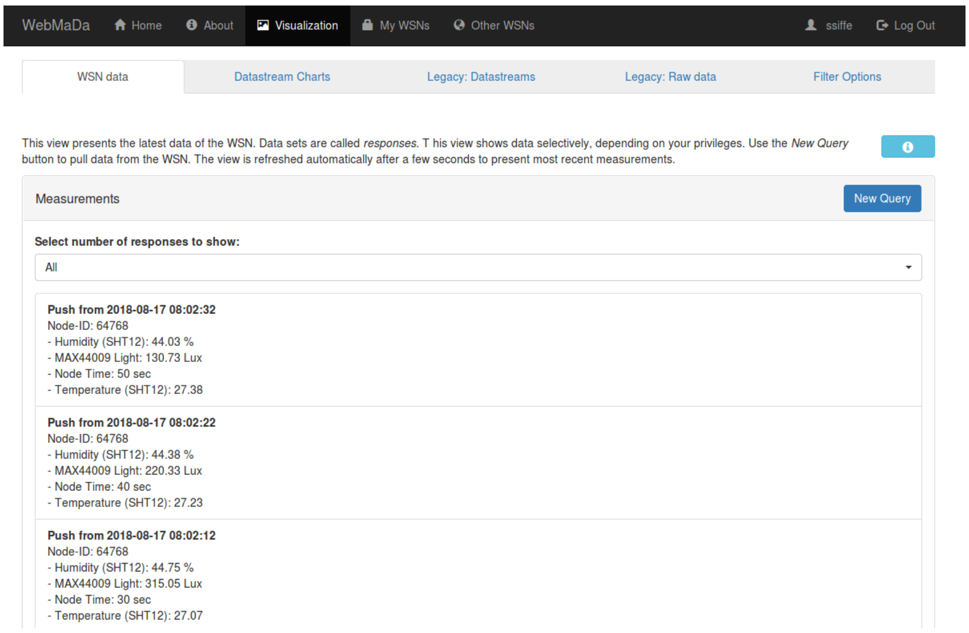
Figure 5.2: Measurements getting collected in CoMaDa. sTiki is used to deliver the measurements securely.

the sample network depicted in Figure 5.1. This network architecture was chosen because it consists of the simplest possible setup that still covers all the discussed cases.

5.2.1 Aggregator to Server

Before the aggregator can send data to the sink/server, it has to establish a session with it so that it can send the data securely. Because the sink is the receiving party (in the code/protocol description named Node B) and the key server at the same time, the messages M2 and M_3 can happen at the sink internally and only M1 and M_4 are needed.

To begin a handshake, the aggregator sends M1 (see Figure 5.4, the message starts on the fourth line with the header) to the sink. Because it is the initiator, it is called Node A in the code and protocol definition. M_1 contains `EF41AEFD0001F3F98422`. The first two bytes (`EF41`) are the sTiki header and contain the magic number (`EF`) that shows that the packet is an sTiki packet as well as the protocol information (`41`). The first three bits of the protocol byte denote the protocol (here: `010`, showing that it is a handshake packet) and the following five bits denote the subprotocol (here: `00001`, showing that it is M_1). After the header follows the ID of the initiating node (node A), `AEFD`, followed by the ID



This view presents the latest data of the WSN. Data sets are called *responses*. T his view shows data selectively, depending on your privileges. Use the *New Query* button to pull data from the WSN. The view is refreshed automatically after a few seconds to present most recent measurements.

Measurements

Select number of responses to show:

All

Push from 2018-08-17 08:02:32
Node-ID: 64768
- Humidity (SHT12): 44.03 %
- MAX44009 Light: 130.73 Lux
- Node Time: 50 sec
- Temperature (SHT12): 27.38

Push from 2018-08-17 08:02:22
Node-ID: 64768
- Humidity (SHT12): 44.38 %
- MAX44009 Light: 220.33 Lux
- Node Time: 40 sec
- Temperature (SHT12): 27.23

Push from 2018-08-17 08:02:12
Node-ID: 64768
- Humidity (SHT12): 44.75 %
- MAX44009 Light: 315.05 Lux
- Node Time: 30 sec
- Temperature (SHT12): 27.07

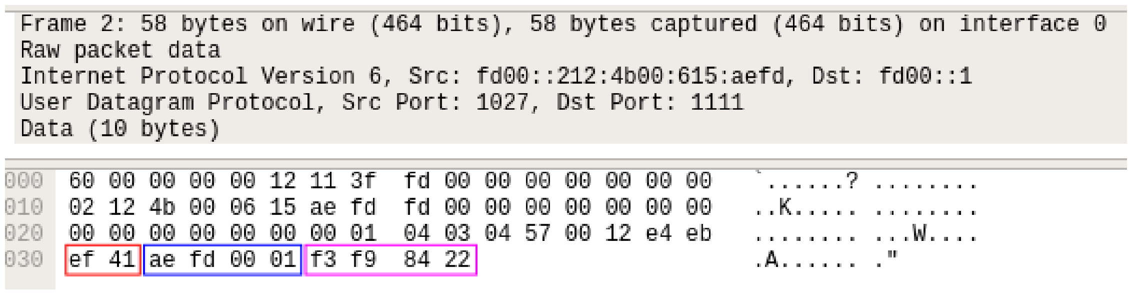
Figure 5.3: After being collected in CoMaDa, the measurements are stored in a database and are visible in the web interface.

of the receiving node (node B), `0001`. The sink's ID is always `0001`. The last four bytes are a random nonce which will be sent back to A, so that it can be sure to which request the response belongs to.

When M_1 is a valid request, the sink will respond with M_4. The process to calculate M_4 can be seen in Listing 5.2. When M_1 arrives, it has to be parsed (Lines 2–4). The MAC validation (line 5) will always succeed because M_1 is not protected by a MAC. The token (line 7) is the most important part of the response as it consists of the session details: First come the involved nodes, nodes `AEFD` and `0001`. The nonce follows, so that node A can match the response to the started handshake. The last 16 bytes of the token are the session key. To keep the session key a secret, the entire token is encrypted with the node's initial master key and a random IV (line 6). To prove that the packet has not been tampered with, a MAC (calculated again with the node's initial master key) is added to the message (Line 9). Then, M_4 is sent back to the initiating node, as can be seen in Figure 5.5. M_4 consists of the sTiki header (`EF44`, the `44` shows that it is handshake packet M_4), the IV (used to decrypt the token), the encrypted token and the MAC, totaling 57 bytes.

Once the handshake is complete, the aggregator can begin sending data. The first message with data (see Figure 5.6 and Listing 5.3) has to contain an initialization vector because none has been set up to this point (the one used for the handshake is discarded). A message with data that includes an IV has the header `EF21` (protocol `001` and subprotocol `00001`). Right after the header come 13 bytes that describe the IV (bytes 14–16 are `000000` because the counters start at 0), followed by the data. The last 16 bytes of the message are the MAC. The data is decrypted by using the just-received initialization vector and the session key.

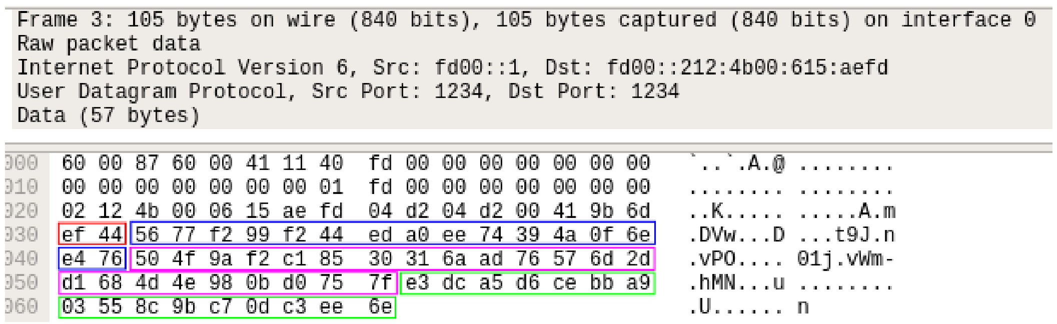
Once an IV was set, the next messages only need to include the message counter, thereby



```
Frame 2: 58 bytes on wire (464 bits), 58 bytes captured (464 bits) on interface 0
Raw packet data
Internet Protocol Version 6, Src: fd00::212:4b00:615:aefd, Dst: fd00::1
User Datagram Protocol, Src Port: 1027, Dst Port: 1111
Data (10 bytes)

000  60 00 00 00 00 12 11 3f  fd 00 00 00 00 00 00 00   `......? ........
010  02 12 4b 00 06 15 ae fd  fd 00 00 00 00 00 00 00   ..K..... ........
020  00 00 00 00 00 00 00 01  04 03 04 57 00 12 e4 eb   ........ ...W....
030  ef 41 ae fd 00 01 f3 f9  84 22                     .A...... ."
```

Figure 5.4: Handshake message M_1 captured by Wireshark. Red: Header, Blue: Node IDs, Pink: Nonce.



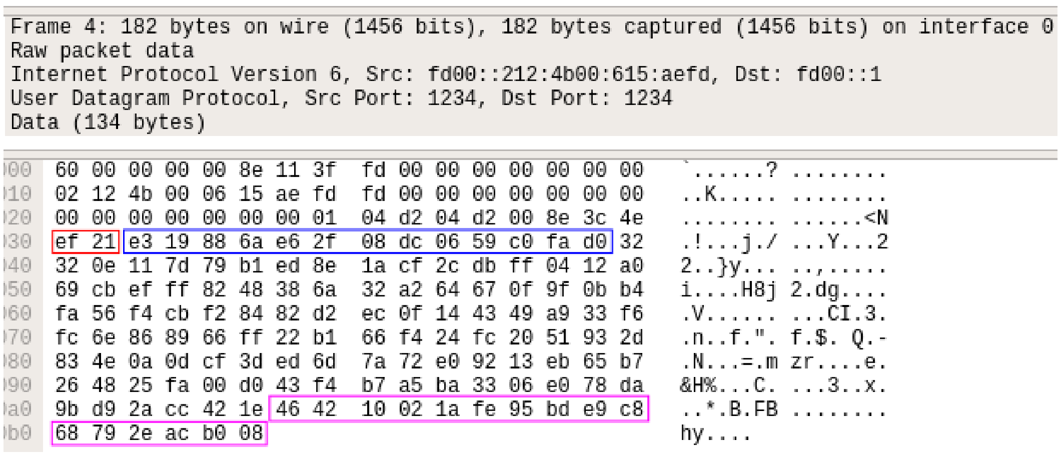
```
Frame 3: 105 bytes on wire (840 bits), 105 bytes captured (840 bits) on interface 0
Raw packet data
Internet Protocol Version 6, Src: fd00::1, Dst: fd00::212:4b00:615:aefd
User Datagram Protocol, Src Port: 1234, Dst Port: 1234
Data (57 bytes)

000  60 00 87 60 00 41 11 40  fd 00 00 00 00 00 00 00   `..`.A.@ ........
010  00 00 00 00 00 00 00 01  fd 00 00 00 00 00 00 00   ........ ........
020  02 12 4b 00 06 15 ae fd  04 d2 04 d2 00 41 9b 6d   ..K..... .....A.m
030  ef 44 56 77 f2 99 f2 44  ed a0 ee 74 39 4a 0f 6e   .DVw...D ...t9J.n
040  e4 76 50 4f 9a f2 c1 85  30 31 6a ad 76 57 6d 2d   .vPO.... 01j.vWm-
050  d1 68 4d 4e 98 0b d0 75  7f e3 dc a5 d6 ce bb a9   .hMN...u ........
060  03 55 8c 9b c7 0d c3 ee  6e                        .U...... n
```

Figure 5.5: Handshake message M_4 captured by Wireshark. Red: Header, Blue: IV, Pink: Token, Green: MAC.

```
Starting tunnel
[sTiki] Received data: EF41AEFD0001F3F98422
[sTiki] Header details: Protocol 2, Subprotocol: 1
[sTiki] [STiki.HandshakePacket$M_1$: IDa: 44797 IDb: 1 nonce: F3F98422 ]
[sTiki] MAC is valid!
[sTiki] IV is 5677F299F244EDA0EE74394A0F6EE4
[sTiki] token is AEFD0001F3F98422E8D1D612190E9E75590016E5DABC73AC
[sTiki] encrypted token is 76504F9AF2C18530316AAD76576D2DD1684D4E980BD0757F
[sTiki] MAC is E3DCA5D6CEBBA903558C9BC70DC3EE6E
[sTiki] sending EF445677F299F244EDA0EE74394A0F6EE476504F9AF2C18530316AAD765
    76D2DD1684D4E980BD0757FE3DCA5D6CEBBA903558C9BC70DC3EE6E to
    /fd00:0:0:0:212:4b00:615:aefd
```

Listing 5.2: Debugging output from CoMaDa when receiving a handshake packet M_1.



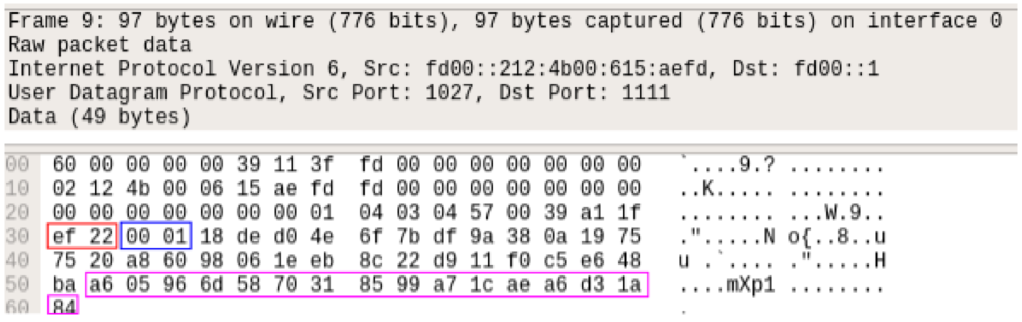
```
Frame 4: 182 bytes on wire (1456 bits), 182 bytes captured (1456 bits) on interface 0
Raw packet data
Internet Protocol Version 6, Src: fd00::212:4b00:615:aefd, Dst: fd00::1
User Datagram Protocol, Src Port: 1234, Dst Port: 1234
Data (134 bytes)

00  60 00 00 00 00 8e 11 3f  fd 00 00 00 00 00 00 00   `......? ........
10  02 12 4b 00 06 15 ae fd  fd 00 00 00 00 00 00 00   ..K..... ........
20  00 00 00 00 00 00 00 01  04 d2 04 d2 00 8e 3c 4e   ........ ......<N
30  ef 21 e3 19 88 6a e6 2f  08 dc 06 59 c0 fa d0 32   .!...j./ ...Y...2
40  32 0e 11 7d 79 b1 ed 8e  1a cf 2c db ff 04 12 a0   2..}y... ..,.....
50  69 cb ef ff 82 48 38 6a  32 a2 64 67 0f 9f 0b b4   i....H8j 2.dg....
60  fa 56 f4 cb f2 84 82 d2  ec 0f 14 43 49 a9 33 f6   .V...... ...CI.3.
70  fc 6e 86 89 66 ff 22 b1  66 f4 24 fc 20 51 93 2d   .n..f.". f.$. Q.-
80  83 4e 0a 0d cf 3d ed 6d  7a 72 e0 92 13 eb 65 b7   .N...=.m zr....e.
90  26 48 25 fa 00 d0 43 f4  b7 a5 ba 33 06 e0 78 da   &H%...C. ...3..x.
a0  9b d9 2a cc 42 1e 46 42  10 02 1a fe 95 bd e9 c8   ..*.B.FB ........
b0  68 79 2e ac b0 08                                  hy....
```

Figure 5.6: Datatransport packet with an IV, captured by Wireshark. Red: Header, Blue: IV, Pink: MAC.

```
[sTiki] Received data: EF21E319886AE62F08DC0659C0FAD032320E117D79B1ED8E1ACF
    2CDBFF0412A069CBEFFF8248386A32A264670F9F0BB4FA56F4CBF28482D2EC0F144349A93
    3F6FC6E868966FF22B166F424FC2051932D834E0A0DCF3DED6D7A72E09213EB65B7264825
    FA00D043F4B7A5BA3306E078DA9BD92ACC421E464210021AFE95BDE9C868792EACB008
[sTiki] Header details: Protocol 1, Subprotocol: 1
[sTiki] [STiki.DataTransportPacket: Source: 212:4b00:615:aefd:fd00:0:0:0 MAC:
    464210021AFE95BDE9C868792EACB008 subprotocol: 1 ]
[sTiki] MAC is valid!
[sTiki] new remote IV is E319886AE62F08DC0659C0FAD03232
[sTiki] Encrypted data is 32320E117D79B1ED8E1ACF2CDBFF0412A069CBEFFF8248386
    A32A264670F9F0BB4FA56F4CBF28482D2EC0F144349A933F6FC6E868966FF22B166F424FC
    2051932D834E0A0DCF3DED6D7A72E09213EB65B7264825FA00D043F4B7A5BA3306E078DA9
    BD92ACC421E
[sTiki] Decrypted data is 0467FF0100000C80B000021234567880B100021234567880B
    200021234567880B300041234567880B400021234567880B500011234567880B000021234
    567880B100021234567880B200021234567880B300041234567880B400021234567880B50
    00112345678
```

Listing 5.3: Debugging output from CoMaDa when receiving a datatransport packet with an IV.



```
Frame 9: 97 bytes on wire (776 bits), 97 bytes captured (776 bits) on interface 0
Raw packet data
Internet Protocol Version 6, Src: fd00::212:4b00:615:aefd, Dst: fd00::1
User Datagram Protocol, Src Port: 1027, Dst Port: 1111
Data (49 bytes)

00  60 00 00 00 00 39 11 3f  fd 00 00 00 00 00 00 00   `....9.? ........
10  02 12 4b 00 06 15 ae fd  fd 00 00 00 00 00 00 00   ..K..... ........
20  00 00 00 00 00 00 00 01  04 03 04 57 00 39 a1 1f   ........ ...W.9..
30  ef 22 00 01 18 de d0 4e  6f 7b df 9a 38 0a 19 75   ."....N o{..8..u
40  75 20 a8 60 98 06 1e eb  8c 22 d9 11 f0 c5 e6 48   u .`.... ."....H
50  ba a6 05 96 6d 58 70 31  85 99 a7 1c ae a6 d3 1a   ....mXp1 ........
60  84                                                 .
```

Figure 5.7: Datatransport packet with a message counter, captured by Wireshark. Red: Header, Blue: Message Counter, Pink: MAC.

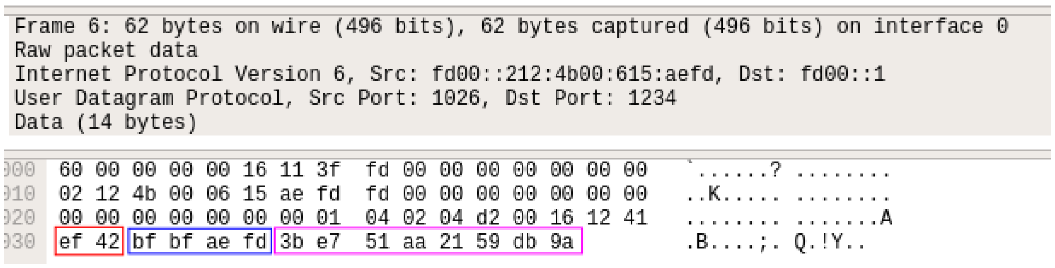
saving eleven bytes per message. Those messages have the header `EF22`. A message with message counter `0001` can be seen in Figure 5.7 and how it is processed in Listing 5.4. The third and fourth byte make up the message counter. Once again, the MAC is in the last 16 bytes.

```
[sTiki] Received data: EF22000118DED04E6F7BDF9A380A19757520A86098061EEB8C2
    2D911F0C5E648BAA605966D5870318599A71CAEA6D31A84
[sTiki] Header details: Protocol 1, Subprotocol: 2
[sTiki] [STiki.DataTransportPacket: Source: 212:4b00:615:aefd:fd00:0:0:0 MAC:
    A605966D5870318599A71CAEA6D31A84 subprotocol: 2 ]
[sTiki] MAC is valid!
[sTiki] decrypted data is 081D000BC50DE1E10D0000000FBFBF000BC30DCBCB0D00000
    014BFBF00
Upload response...
```

Listing 5.4: Debugging output from CoMaDa when receiving a datatransport packet with an message counter.

5.2.2 Collector to Aggregator

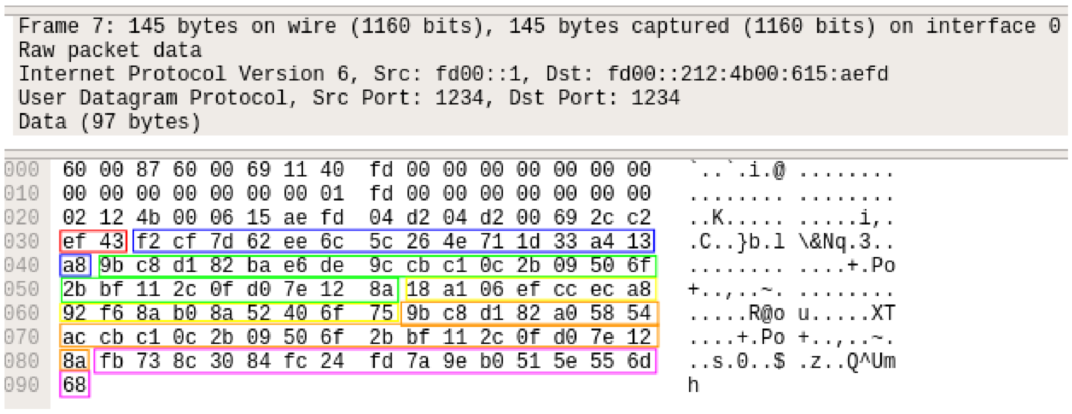
Sending data from collectors to aggregators works exactly the same way as when sending data from an aggregator to the sink. The only difference is in the handshake because none of them is the key server. For a handshake between collector and aggregator, the collector sends M_1 to the aggregator. The aggregator then sends M2 to the key server (see Figure 5.8). The key server recognizes M2 from the header `EF42` and processes it (see Listing 5.5). If the nodes are allowed to talk, the key server chooses a session key and makes the token for both nodes, which includes the IDs of the nodes (`BFBF` and `AEFD`), their respective nonces (`3BE751AA` and `2159DB9A`) and the session key. The token gets encrypted with an IV and the nodes' initial master keys. The key server also precomputes the MAC for M_4 because node B does not have node A's master key. Everything is then sent as M_3 (see Figure 5.9) to node B, which then forwards the token to A, which then can begin sending data.



```
Frame 6: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface 0
Raw packet data
Internet Protocol Version 6, Src: fd00::212:4b00:615:aefd, Dst: fd00::1
User Datagram Protocol, Src Port: 1026, Dst Port: 1234
Data (14 bytes)

000  60 00 00 00 00 16 11 3f  fd 00 00 00 00 00 00 00   `......? ........
010  02 12 4b 00 06 15 ae fd  fd 00 00 00 00 00 00 00   ..K..... ........
020  00 00 00 00 00 00 00 01  04 02 04 d2 00 16 12 41   ........ .......A
030  ef 42 bf bf ae fd 3b e7  51 aa 21 59 db 9a         .B....;. Q.!Y..
```

Figure 5.8: Handshake message M2 captured by Wireshark. Red: Header, Blue: Node IDs, Pink: Nonces.



```
Frame 7: 145 bytes on wire (1160 bits), 145 bytes captured (1160 bits) on interface 0
Raw packet data
Internet Protocol Version 6, Src: fd00::1, Dst: fd00::212:4b00:615:aefd
User Datagram Protocol, Src Port: 1234, Dst Port: 1234
Data (97 bytes)

000  60 00 87 60 00 69 11 40  fd 00 00 00 00 00 00 00   `..`.i.@ ........
010  00 00 00 00 00 00 00 01  fd 00 00 00 00 00 00 00   ........ ........
020  02 12 4b 00 06 15 ae fd  04 d2 04 d2 00 69 2c c2   ..K..... .....i,.
030  ef 43 f2 cf 7d 62 ee 6c  5c 26 4e 71 1d 33 a4 13   .C..}b.l \&Nq.3..
040  a8 9b c8 d1 82 ba e6 de  9c cb c1 0c 2b 09 50 6f   ........ ....+.Po
050  2b bf 11 2c 0f d0 7e 12  8a 18 a1 06 ef cc ec a8   +..,..~. ........
060  92 f6 8a b0 8a 52 40 6f  75 9b c8 d1 82 a0 58 54   .....R@o u.....XT
070  ac cb c1 0c 2b 09 50 6f  2b bf 11 2c 0f d0 7e 12   ....+.Po +..,..~.
080  8a fb 73 8c 30 84 fc 24  fd 7a 9e b0 51 5e 55 6d   ..s.0..$ .z..Q^Um
090  68                                                 h
```

Figure 5.9: Handshake message M_3 captured by Wireshark. Red: Header, Blue: IV, Green: Token for A, Yellow: MAC for M_4, Orange: Token for B, Pink: MAC.

```
[sTiki] Received data: EF42BFBFAEFD3BE751AA2159DB9A
[sTiki] Header details: Protocol 2, Subprotocol: 2
[sTiki] [STiki.HandshakePacketM2: IDa: 49087 IDb: 44797 nonceA: 3BE751AA
    nonceB: 2159DB9A ]
[sTiki] MAC is valid!
[sTiki] session key will be AE2D8DC5CE10B9FC02DBD4A70BC7B98F
[sTiki] IV will be F2CF7D62EE6C5C264E711D33A413A8
[sTiki] Token for A is BFBFAEFD3BE751AAAE2D8DC5CE10B9FC02DBD4A70BC7B98F
[sTiki] Encryped token for A is
    9BC8D182BAE6DE9CCBC10C2B09506F2BBF112C0FD07E128A
[sTiki] MAC for A is 18A106EFCCECA892F68AB08A52406F75
[sTiki] Token for B is BFBFAEFD2159DB9AAE2D8DC5CE10B9FC02DBD4A70BC7B98F
[sTiki] Encryped token for B is
    9BC8D182A05854ACCBC10C2B09506F2BBF112C0FD07E128A
[sTiki] MAC is FB738C3084FC24FD7A9EB0515E556D68
[sTiki] sending EF43F2CF7D62EE6C5C264E711D33A413A89BC8D182BAE6DE9CCBC10
    C2B09506F2BBF112C0FD07E128A18A106EFCCECA892F68AB08A52406F759BC8D182A05854
    ACCBC10C2B09506F2BBF112C0FD07E128AFB738C3084FC24FD7A9EB0515E556D68 to
    /fd00:0:0:0:212:4b00:615:aefd
```

Listing 5.5: Debugging output from CoMaDa when receiving a handshake packet M2.

5.2.3 Error Cases

Within sTiki, two things can go wrong: an IV can get lost or even an entire session. An IV can get lost whenever a new one is transmitted. If the message does not arrive at its destination, when the next message is sent, the receiver has no valid IV to decrypt the message. The other case is when a node crashed or had its power supply interrupted for a moment. Then the node reboots but has lost all session keys and recollection of ever connecting to other nodes.

Lost Initialization Vector (IV)

Listing 5.6 shows what happens when a missing IV is simulated (it is intentionally dropped, see line 3). An alert packet is sent (see line 6) to the communication partner, indicating that there is no valid IV present. This is done with the packet header `EF63`, followed by the two nodes' IDs (`AEFD`, the aggregator and `0001`, the key server). The MAC is computed with the *session key* because there is a live session between the two nodes. The packet that could not be decrypted because of the missing IV is lost. The nodes do not keep their last few messages in memory. The next message after sending the alert packet includes a new IV (line 16), just like the first message after the handshake.

Lost Session

When a node receives a message from a node it has no recollection of establishing a session to, it tries to notify the sender node of this problem. But because the two nodes have no common session, there is no way for the receiver node to do so securely. If an unsecured packet would suffice to initiate a new handshake, a denial of service attacke would be trivial to perform by sending missing session packets to the whole network continuously.

Because a node has no secure way to notify another node of a missing session, the message has to be relayed by the key server, which can then notify the node using its initial key. Such a case is shown in Listing 5.7: The key server receives a packet with the header `EF64`, indicating that it is about a missing session. In the packet included are the two nodes' IDs, `BFBF` and `AEFD`, protected by a MAC. The MAC is computed using the initial key of the sender node. The server then sends the two node IDs to the destination node with the header `EF65` (to tell the node it is about a missing session, but meant for a node and not for the server, which would be `EF64`). This message is protected by the initial key of the destination node. As the listing shows with the handshake packet (line 11), after sending the message to the destination node, it initiated a new handshake between the two nodes.

5.3 Power Consumption

Power consumption was measured by connecting a multimeter in between one side of a battery and the power socket on the node, as shown in Figure 5.10. Multiple attempts

```
[sTiki] [STiki.DataTransportPacket: Source: 212:4b00:60d:aefd:fd00:0:0:0 MAC: B06593DDC09F65DE3758EE6F100FFF4B subprotocol: 2 ]
[sTiki] MAC is valid!
[sTiki] Randomly dropping IV!
[sTiki] Invalidating remote IV for node aefd
[sTiki] Missing IV!
[sTiki] sending EF63AEFD0001F795121581BF74E9995BDED87EE14786 to /fd00:0:0:0:212:4b00:60d:aefd

[sTiki] Received data: EF218256E3FEC82056183CCA91572FA1A80553F06E6F13763CD7E6EB4A8BD63C
   6B7B6F2F2F6C83145B2FF8BB23FE66855476096A78C05629899544845A3910E0
   F86155CCE4858B3027E448C26B8743A85650CECC5D35C25F1FF8F84BEFF68AE5
   B8DB152ECB072A5ED3C8AE51EFE154C69AE7A457B10352A0DA268A9D9FC51FED
   AE227ABF1AC0
[sTiki] Header details: Protocol 1, Subprotocol: 1
[sTiki] [STiki.DataTransportPacket: Source: 212:4b00:60d:aefd:fd00:0:0:0 MAC: 52A0DA268A9D9FC51FEDAE227ABF1AC0 subprotocol: 1 ]
[sTiki] MAC is valid!
[sTiki] new remote IV is 8256E3FEC82056183CCA91572FA1A8
[sTiki] Encrypted data is A1A80553F06E6F13763CD7E6EB4A8BD63C6B7B6F2F2F6C83145B2FF8BB23FE66
   855476096A78C05629899544845A3910E0F86155CCE4858B3027E448C26B8743
   A85650CECC5D35C25F1FF8F84BEFF68AE5B8DB152ECB072A5ED3C8AE51EFE154
   C69AE7A457B103
[sTiki] Decrypted data is 0467FF0100000C80B000021234567880B100021234567880B200021234567880
   B300041234567880B400021234567880B500011234567880B000021234567880
   B100021234567880B200021234567880B300041234567880B400021234567880
   B5000112345678
```

Listing 5.6: Debugging output from CoMaDa when simulating a missing IV. The next packet contains a new IV.

```
[sTiki] Received data: EF64BFBFAEFD30702F211FF9A603062733412ABA4B1C
[sTiki] Header details: Protocol 3, Subprotocol: 4
[sTiki] [STiki.AlertPacket: sourceId: bfbf targetId: aefd MAC:
   30702F211FF9A603062733412ABA4B1C subprotocol: 4 from: aefd ]
[sTiki] MAC is valid!
[sTiki] Handling invalid Session between bfbf and aefd
[sTiki] MAC is 47459EBCA77754E16FF082F4B1E9C9B7
[sTiki] sending EF65BFBFAEFD47459EBCA77754E16FF082F4B1E9C9B7 to
   /fd00:0:0:0:212:4b00:60d:bfbf

[sTiki] Received data: EF42BFBFAEFDF9B04254BCA332F9
[sTiki] Header details: Protocol 2, Subprotocol: 2
[sTiki] [STiki.HandshakePacketM2: IDa: bfbf IDb: aefd nonceA: F9B04254 nonceB:
   BCA332F9 ]
[sTiki] MAC is valid!
[sTiki] session key will be 8F9D087D1BD8D950DEB9AD71215E01E8
[sTiki] IV will be 25FC32BC1669D02D5419AFF2D86189
[sTiki] Token for A is BFBFAEFDF9B042548F9D087D1BD8D950DEB9AD71215E01E8
[sTiki] Encryped token for A is
   52E07F609CCC32EE3683CE899824C5CCAF7E61005435A65C
[sTiki] MAC for A is 9497B128F5BF0C0554A37114F2361938
[sTiki] Token for B is BFBFAEFDBCA332F98F9D087D1BD8D950DEB9AD71215E01E8
[sTiki] Encryped token for B is
   52E07F60D9DF42433683CE899824C5CCAF7E61005435A65C
[sTiki] MAC is 4E9D4C05A948EEC2EC258F53CFC0E1BD
[sTiki] sending
   EF4325FC32BC1669D02D5419AFF2D8618952E07F609CCC32EE3683CE899824C5
   CCAF7E61005435A65C9497B128F5BF0C0554A37114F236193852E07F60D9DF42
   433683CE899824C5CCAF7E61005435A65C4E9D4C05A948EEC2EC258F53CFC0E1BD to
      /fd00:0:0:0:212:4b00:60d:aefd
```

Listing 5.7: Debugging output of CoMaDa when receiving an alert packet about a missing session. After relaying the missing session to the sender node it initiates a new handshake.

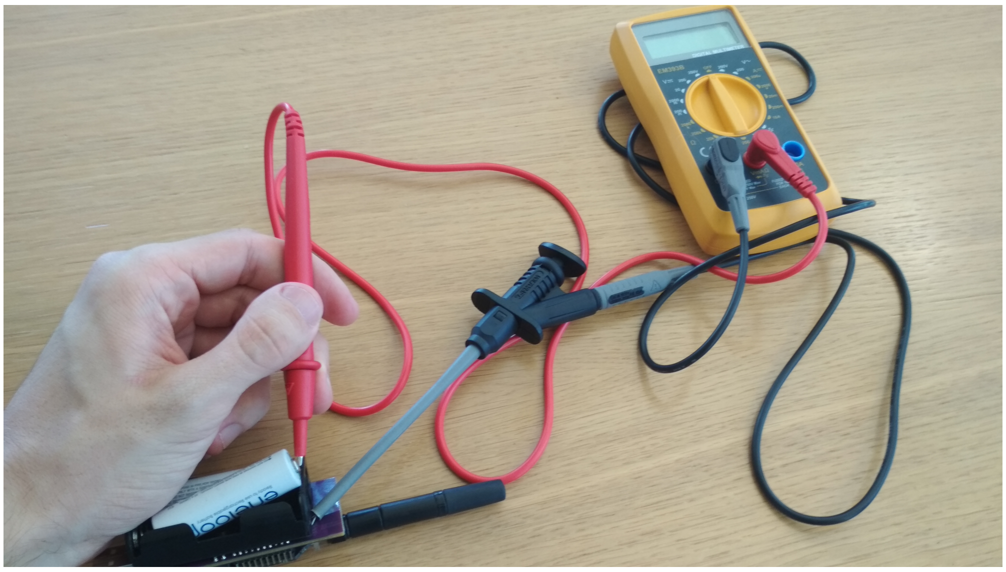


Figure 5.10: Power consumption measuring setup. The multimeter is set to 200mA.

were made and produced very similar results. During standby the nodes had a power consumption of 29.5mA ± 0.1mA. When sending messages, the current increased to 31.5mA ±0.2mA. This was the same on nodes with and without sTiki, which is to be expected because sending and computing data is the same process on both versions. The durations of those phases should be longer on the version with sTiki because the messages have some overhead because of header and MAC, and the calculations also should take longer because the version with sTiki also has to compute the MAC and manage sessions. On the nodes with sTiki the multimeter occasionally showed lower current flowing around 28mA. Where this comes from is unclear, one possibility is that the hardware implementation of AES takes less power and the processor pauses calculations while the encryption is going on. However, the drops are not frequent enough to match with every encryption operation, but this might be caused by a low sampling rate of the multimeter which misses short encryption sequences.

5.4 Memory Consumption

By analyzing the binary file generated by the `make` command with the utility `size`, it is possible to find out the ROM and RAM requirements of the code running on the nodes. Table 5.1 shows the measurements. It shows a RAM usage of 368 bytes (reported as `data` and `bss`) and a ROM usage of 4548 and 4556 bytes (reported as `text`) for sTiki. This difference in ROM usage most likely results from differing amounts of code that calls sTiki and/or differing compiler optimizations.

	Collector		Aggregator	
	ROM	RAM	ROM	RAM
Without sTiki	48248	15999	48357	17073
With sTiki	52796	16367	52913	17441
sTiki	4548	368	4556	368

Table 5.1: sTiki size according to `size` utility (all numbers in bytes).

5.5 Comparison to Earlier Work

The implementation of TinySAM on TinyOS in [21] uses almost 6.5 kilobytes of ROM and about 1.5 kilobytes of RAM with similar configurations, which makes sTiki 2 kilobytes smaller in ROM usage and 1.1 kilobytes smaller in RAM usage. Some of the smaller size comes from the limit of only one ongoing handshake at a time. Where the rest of the difference comes from is not quite clear. It might stem from the AES implementation, but tracking the precise reason down is outside of the scope of this work.

In comparison with TinyDTLS, sTiki requires about 15 times less ROM and 50 times less RAM. This big difference was expected for the most part because DTLS requires three different encryption algorithms. Just the binding to RSA takes about the same amount of ROM as the entirety of sTiki [17]. The price for using such a small implementation is non-compliance with a standard such as DTLS, which might be more important in some use cases.

When comparing with MiniSec, sTiki's strength is again its size: it uses about half as much RAM and a little less than four times the ROM MiniSec uses. The biggest tradeoff during operation in comparison with MiniSec is in encryption speed because MiniSec uses Offset CodeBook mode to compute ciphertext and MAC in a single pass. sTiki uses a separate algorithm for those two operations, thereby spending more time and energy to compute.

Comparing the power consumption is not feasible with the collected measurements as precise timings could not be collected for the Contiki implementation.

Chapter 6

Conclusion and Future Work

The security solution sTiki was developed out of TinySAM [21] and implemented for Contiki 3.0. A key server offers authentication and dynamic node management. The key server responsibilities were integrated into the CoMaDa framework. Symmetric encryption provides data integrity and a message counter guarantees freshness of received messages. For encryption, a multitude of algorithms were considered and compared against each other with AES-128 coming out on top. Confidentiality is achieved by using a Message Authentication Code (MAC).

The implementation consumes about 4.5 kilobytes of ROM and around 400 bytes of RAM, which is an improvement of about 2 kilobytes of ROM and about 1.1 kilobytes of RAM in comparison to the closely related TinySAM. sTiki was demonstrated to work and was tested successfully in multiple setups, with some scenarios missing sTiki support from certain nodes. Mixing sTiki with unencrypted nodes does work, even when not every device uses the same hardware. It is, however, not possible to move from encrypted nodes back to unencrypted ones.

The biggest weakpoint from a practical perspective is key management. Even though not every node pair requires its own pre-shared key, managing the node keys is not yet convenient enough. An management interface on the CoMaDa framework exists in a different version, but the process is still manual and should be automated furter. A method with QR codes or RFID tags was suggested in [21] but has unsolved security problems.

Another useful expansion would be to give sTiki some access to the routing protocol used in the TinyIPFIX implementation. Right now, there is no way for the nodes to know whether the handshake timed out because a message got lost or because the two nodes are not allowed to talk to each other. Because of that, they probably will never try to send data to a different node and will never get to send their data.

Bibliography

[1] M. Abadi, R. Needham: *Prudent engineering practice for cryptographic protocols*, in: *Proceedings of 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, IEEE, May 1994, pp. 122–136. URL `http://ieeexplore.ieee.org/document/296587,lastvisited14/08/2018`.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci: *A survey on sensor networks*, IEEE Communications Magazine, volume 40, no. 8, pp. 102–114, Aug 2002, ISSN 0163-6804, doi:10.1109/MCOM.2002.1024422.

[3] S. Badel, N. Dağtekin, J. Nakahara, K. Ouafi, N. Reffé, P. Sepehrdad, P. Sušil, S. Vaudenay: *ARMADILLO: A Multi-purpose Cryptographic Primitive Dedicated to Hardware*, in: S. Mangard, F. Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, Berlin, Heidelberg: Springer, 2010, pp. 398–412, ISBN 978-3-642-15031-9.

[4] W. Bechkit, Y. Challal, A. Bouabdallah: *A new class of Hash-Chain based key pre-distribution schemes for WSN*, Computer Communications, volume 36, no. 3, pp. 243–255, 2013, ISSN 0140-3664, doi:10.1016/j.comcom.2012.09.015.

[5] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, R. Han: *MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms*, Mobile Networks and Applications, volume 10, no. 4, pp. 563–579, Aug 2005, ISSN 1572-8153, doi:10.1007/s11036-005-1567-8.

[6] Q. Cao, T. Abdelzaher, J. Stankovic, T. He: *The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks*, in: *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, IEEE, April 2008, pp. 233–244, doi:10.1109/IPSN.2008.54.

[7] K. Claessen, J. Hughes: *QuickCheck: a lightweight tool for random testing of Haskell programs*, Acm sigplan notices, volume 46, no. 4, pp. 53–64, 2011, doi: 10.1145/357766.351266.

[8] B. Claise, B. Trammell, P. Aitken: *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information*, STD 77, RFC Editor, September 2013. URL `http://www.rfc-editor.org/rfc/rfc7011.txt,lastvisited14/08/2018`.

[9] Mu. Çoban, F. Karakoç, M. Özen: *Cryptanalysis of QTL Block Cipher*, in: Andrey Bogdanov, editor, *Lightweight Cryptography for Security and Privacy*, Cham: Springer International Publishing, 2017, pp. 60–68, ISBN 978-3-319-55714-4. URL `https://link.springer.com/chapter/10.1007%2F978-3-319-55714-4_5,lastvisited14/08/2018`.

[10] A. Dunkels, B. Gronvall, T. Voigt: *Contiki - a lightweight and flexible operating system for tiny networked sensors*, in: *29th Annual IEEE International Conference on Local Computer Networks*, IEEE, Nov 2004, pp. 455–462, ISSN 0742-1303, doi: 10.1109/LCN.2004.38.

[11] T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, L. Uhsadel: *A Survey of Lightweight-Cryptography Implementations*, IEEE Design Test of Computers, volume 24, no. 6, pp. 522–533, Nov 2007, ISSN 0740-7475, doi:10.1109/MDT.2007.178.

[12] M. Omer Farooq, T. Kunz: *Operating Systems for Wireless Sensor Networks: A Survey*, Sensors, volume 11, no. 6, pp. 5900–5930, 2011, ISSN 1424-8220, doi:10.3390/s110605900.

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design patterns: Elements of Reusable Object-Oriented Software*, Pearson Education India, 1995, ISBN 0201633612.

[14] R. Hummen, H. Shafagh, S. Raza, T. Voig, K. Wehrle: *Delegation-based Authentication and Authorization for the IP-based Internet of Things*, in: *2014 Eleventh Annual IEEE International Conference on Sensing, Communication and Networking (SECON)*, IEEE, December 2014, pp. 284–292, ISBN 978-1-4799-4657-0, doi: 10.1109/SAHCN.2014.6990364.

[15] H. Karl, A. Willig: *Protocols and architectures for wireless sensor networks*, John Wiley & Sons, 2007.

[16] S. U. Khan, C. Pastrone, L. Lavagno, M. A. Spirito: *An Authentication and Key Establishment Scheme for the IP-Based Wireless Sensor Networks*, Procedia Computer Science, volume 10, pp. 1039–1045, 2012, ISSN 1877-0509, doi:10.1016/j.procs.2012.06.144. ANT 2012 and MobiWIS 2012.

[17] T. Kothmayr, C. Schmitt, W. Hu, M. Brünig, G. Carle: *DTLS based security and two-way authentication for the Internet of Things*, Ad Hoc Networks, volume 11, no. 8, pp. 2710–2723, 2013.

[18] K. Krentz, H. Rafiee, C. Meinel: *6LoWPAN security: adding compromise resilience to the 802.15.4 security sublayer*, in: *Proceedings of the International Workshop on adaptive security*, ACM, 2013, ASPI '13, pp. 1–10, ISBN 9781450325431, doi:10.1145/2523501.2523502.

[19] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, D. Culler: *TinyOS: An operating system for sensor networks*, in: *Ambient Intelligence*, Springer Berlin Heidelberg, pp. 115–148, 2005, ISBN 3540238670, doi:10.1007/3-540-27139-2_7.

[20] L. Li, B. Liu, H. Wang: *QTL: A new ultra-lightweight block cipher*, Microprocessors and Microsystems, volume 45, pp. 45–55, 2016, ISSN 0141-9331, doi: 10.1016/j.micpro.2016.03.011.

[21] P. Lowack: *Key Management in Wireless Sensor Networks with Support for Aggregation Nodes*, Masters thesis, Lehrstuhl für Netzwerkarchitekturen und Netzdienste, Fakultät für Informatik, Technische Universität München, München, Germany, May 2013.

[22] M. Luk, G. Mezzour, A. Perrig, V. Gligor: *MiniSec: A Secure Sensor Network Communication Architecture*, in: *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, New York, NY, USA: ACM, 2007, IPSN '07, pp. 479–488, ISBN 978-1-59593-638-7, doi:10.1145/1236360.1236421.

[23] B. J. Mohd, T. Hayajneh, A. V. Vasilakos: *A survey on lightweight block ciphers for low-resource devices: Comparative study and open issues*, Journal of Network and Computer Applications, volume 58, pp. 73–93, 2015, ISSN 1084-8045, doi:10.1016/j.jnca.2015.09.001.

[24] D. Otway, O. Rees: *Efficient and Timely Mutual Authentication*, SIGOPS Oper. Syst. Rev., volume 21, no. 1, pp. 8–10, January 1987, ISSN 0163-5980, doi:10.1145/24592.24594.

[25] N. R. Potlapally, S. Ravi, A. Raghunathan, N. K. Jha: *Analyzing the energy consumption of security protocols*, in: *Proceedings of the 2003 international symposium on Low power electronics and design*, ACM, 2003, pp. 30–35, doi:10.1145/871506.871518.

[26] S. Sadeghi, N. Bagheri, M. A. Abdelraheem: *Cryptanalysis of reduced QTL block cipher*, Microprocessors and Microsystems, volume 52, pp. 34–48, 2017, ISSN 0141-9331, doi:10.1016/j.micpro.2017.05.007.

[27] C. Schmitt, A. Freitag, G. Carle: *CoMaDa: An adaptive framework with graphical support for Configuration, Management, and Data handling tasks for wireless sensor networks*, in: *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*, IEEE, October 2013, pp. 211–218, ISSN 2165-9605, doi: 10.1109/CNSM.2013.6727839.

[28] C. Schmitt, T. Kothmayr, B. Ertl, W. Hu, L. Braun, G. Carle: *TinyIPFIX: An efficient application protocol for data exchange in cyber physical systems*, Computer Communications, volume 74, pp. 63–76, 2016, ISSN 0140-3664, doi:10.1016/j.comcom.2014.05.012. Current and Future Architectures, Protocols, and Services for the Internet of Things.

[29] C. Schmitt, B. Stiller, B. Trammell: *TinyIPFIX for Smart Meters in Constrained Networks*, RFC 8272, RFC Editor, November 2017. URL `https://www.rfc-editor.org/rfc/rfc8272.txt,lastvisited14/08/2018`.

[30] J. Sen: *A Survey on Wireless Sensor Network Security*, International Journal of Communication Networks and Information Security (IJCNIS), volume 1, pp. 55–78, August 2010. URL `https://arxiv.org/abs/1011.1529,lastvisited14/08/2018`.

[31] L. Sgier: *Optimization of TinyIPFIX Implementation in Contiki and Realtime Visualization of Data*, Software project, Communication Systems Group, Department of Informatics, University of Zurich, Zurich, Switzerland, October 2016.

[32] L. Sgier: *TinyIPFIX Aggregation in Contiki*, Internship, Communication Systems Group, Department of Informatics, University of Zurich, Zurich, Switzerland, May 2017.

[33] Z. Shelby, C. Bormann: *6LoWPAN: The wireless embedded Internet*, volume 43, John Wiley & Sons, 2011.

[34] JH. Song, R. Poovendran, J. Lee, T. Iwata: *The AES-CMAC Algorithm*, RFC 4493, RFC Editor, June 2006. URL `https://tools.ietf.org/html/rfc4493,lastvisited14/08/2018`.

[35] Texas Instruments: *CC2538 A Powerful System-On-Chip for 2.4-GHz IEEE 802.15.4-2006 and ZigBee Applications*, December 2012. URL `http://www.ti.com/product/CC2538,lastvisited14/08/2018`.

[36] Xavier Vilajosana, Pere Tuset, Thomas Watteyne, Kris Pister: *OpenMote: Open-Source Prototyping Platform for the Industrial IoT*, in: N. Mitton, M. E. Kantarci, A. Gallais, S. Papavassiliou, editors, *Ad Hoc Networks*, Cham: Springer International Publishing, 2015, pp. 211–222, ISBN 978-3-319-25067-0, doi:10.1007/978-3-319-25067-0_17.

[37] J. Yick, B. Mukherjee, D. Ghosal: *Wireless sensor network survey*, Computer Networks, volume 52, no. 12, pp. 2292–2330, 2008, ISSN 1389-1286, doi:10.1016/j.comnet.2008.04.002.

Abbreviations

AES	Advanced Encryption Standard
ANOR	AN Otway-Rees
CBC	Chain-Block Chaining
DTLS	Datagram Transport Layer Security
ECC	Elliptic Curve Cryptography
IETF	Internet Engineering Task Force
IoT	Internet of Things
IV	Initialization Vector
MAC	Message Authentication Code
OS	Operating System
P2P	Peer-to-Peer
RAM	Random Access Memory
ROM	Read-only Memory
SSL/TLS	Secure Socket Layer/Transport Layer Security
TPM	Trusted Platform Module
UDP	User Dataram Protocol
WBAN	Wireless Body Area Network
WSN	Wireless Sensor Network

List of Figures

List of Tables

Listings