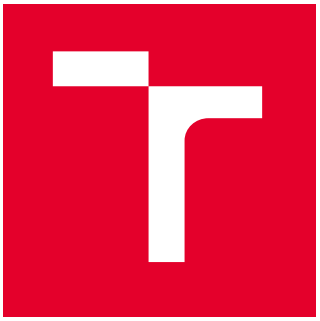


BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

OPTIMIZING OF CYBER SECURITY OF PRECISION TIME PROTOCOL (PTP) ON ETHERNET FOR ARM BASED CPU

OPTIMIZING OF CYBER SECURITY OF PRECISION TIME PROTOCOL (PTP) ON ETHERNET FOR ARM BASED CPU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Borivoje Latinovič**
AUTOR PRÁCE

SUPERVISOR — **prof. Ing. Zdeněk Smékal, CSc.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Communications and Networking (Double-Degree)**

Department of Telecommunications

Student: Bc. Borivoje Latinovič ***ID:*** 222948

Year of study: 2 ***Academic year:*** 2023/24

TITLE OF THESIS:

Optimizing of cyber security of Precision Time Protocol (PTP) on Ethernet for ARM based CPU

INSTRUCTION:

Test the behaviour of the PTP protocol on an Ethernet network in the mode of operation without applied cyber security and in the mode of applied cyber security (encryption) and compare the results. Subsequently, optimize the cyber security mode so that the differences in the behaviour of the PTP protocol are as small as possible. Consider in an Ethernet network with the operation of endpoints, built on the HW architecture of ARM.

RECOMMENDED LITERATURE:

[1] RFC 8173 Precision Time Protocol Version 2 (PTPv2)

[2] i.MX 8X SECO HSM vs. FIPS 140-2 Non-Proprietary Security Policy, NXP 02/22

Date of project specification: 5.2.2024 ***Deadline for submission:*** 21.5.2024

Supervisor: prof. Ing. Zdeněk Smékal, CSc.

Consultant: Ing. Rudolf Procházka

doc. Ing. Jiří Hošek, Ph.D.
Chair of study program board

ABSTRACT

This thesis provides the selection and implementation of different security protocols to be used with Precision Time Protocol (PTP). The thesis further tests the behavior of PTP under different security protocols, using an automatized measurement tool, and analyses the effect on PTP in order to determine the most optimal security protocol.

KEYWORDS

Precision Time Protocol (PTP), Synchronization, Timing, Encryption, Encapsulation, Linux

LATINOVIČ, Borivoje. *Optimizing of cyber security of Precision Time Protocol (PTP) on Ethernet for ARM based CPU*. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications, 2024, 112 p. Master's Thesis. Advised by prof. Ing. Zdeněk Smékal, CSc.

Author's Declaration

Author:	Bc. Borivoje Latinovič
Author's ID:	222948
Paper type:	Master's Thesis
Academic year:	2023/24
Topic:	Optimizing of cyber security of Precision Time Protocol (PTP) on Ethernet for ARM based CPU

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....

author's signature*

*The author signs only in the printed version.

Contents

List of Figures

List of Tables

Listings

Introduction

The cyber security is a very broad concept, which encapsulates many different technologies. In the context of the thesis, the cyber security is meant as a cryptographic encryption of *Precision Time Protocol* (PTP) data transported over the network. One of the big vulnerabilities of the PTP is that by itself it does not provide any possibility for data encryption. In order to make communication of PTP cryptographically secure and adhere to cyber security standards an external security protocol must be selected, configured for PTP and tested. However in order to optimize the cyber security of PTP to the furthest extent, multiple different protocols must be considered and it should be determined which is the most optimal. However, before continuing with the aforementioned topics, some introductory information should be provided.

–

The need for correct timing and synchronization of applications has become significantly important in many of today's use cases. With the increasing demands of various industries, such as monitoring systems, control systems, manufacturing devices, and many others, the complexity and the number of used devices have increased as well, and with that, the need to properly synchronize all the components. By ensuring that the timing is precise and correct in all the nodes in a given time domain, it can be ensured that all the specific requirements are fulfilled at the correct time, and correctly from the perspective of the whole system.

The PTP is a networking timing protocol that was developed specifically for this reason. With PTP applied, much higher time accuracy and precision can be achieved as compared to its predecessors. The first standardized version of PTP was described in IEEE 1588-2002. This standard was however surpassed by the IEEE 1588-2008, which is forward-compatible with the latest standardization IEEE 1588-2019.

The network security protocols generally work by implementing cryptographic algorithms, in order to encrypt the traffic and prevent the unauthorized elements from accessing the confidential information. There are many options in regard to security protocols and each may function differently and may have different cryptographic properties. Each of the security protocols may also have a different effect on the efficiency of the network and the underlying encrypted applications. With that in mind, the correct security protocol must be selected, in consideration of the use scenario and the effect on the PTP precision.

All implementations in the thesis shall be done on the devices running on the Linux operating system and all the utilized software shall be open-source.

1 Precision Time Protocol (PTP)

1.1 Overview of timing protocols

Before diving into specifics about PTP protocol, it would be advisable to provide the reader with a brief explanation and meaning behind the network timing protocols. All the computers keep track of time using an internal oscillator. Computer oscillators commonly use internal crystals, to which the voltage is applied and thus the required frequency is produced [2]. This frequency is used to generate the clock signal in the computer system. However, due to various reasons, such as internal crystal degradation or temperature effects, the internal computer time will inevitably start to shift. In scenarios, where the computers are required to provide timing precision in nanoseconds, the correct timing is crucial. If per se, the computers were running unsynchronized in an interconnected system, many errors and inconsistencies would occur, due to the fact that each computer would rely solely on its own time, which would inevitably start to drift from the reference time, especially in cases where cheaper oscillators are used. This is why the timing protocols are used.

The basics of the timing protocols can be explained with the help of *Network Time Protocol* (NTP). The NTP protocol, which belongs to one of the oldest internet protocols, was initially designed to provide an accuracy of about several hundred milliseconds of *Coordinated Universal Time* (UTC) time [3]. The goal of network timing protocols is to ensure synchronized time across all participating network nodes. This application is especially crucial in many practical domains such as telecommunications, finance, industrial automation, monitoring devices, and many others. In all these cases, it is essential to provide correct timing with great precision, since many devices in these domains are bound to the clock in order to fulfill the required tasks within the broader system and provide correctly timestamped outputs. The core function of the timing protocols is often similar in that, that reference time is sent over the network, from some sort of reference clock device to the end nodes.

1.2 Description

The PTP, is a network protocol for the first time defined in the IEEE 1588 standard, of which the implementation was led by John Eidson. As described in the *Request for Comments* (RFC) document, with the serial number 8173, the main idea behind

the development of the IEEE 1588 was to provide clock accuracies not feasible by Network Time Protocol (NTP), to devices with restricted or no GPS access [1].

The PTP protocol uses a master-slave system to provide time synchronization to the end nodes in packet-based network systems. The PTP standard was designed in order to allow multicast or unicast communication, or both at the same time [4]. The synchronization to the Grandmaster clock can be achieved within the microsecond to nanosecond range while using up a minimum amount of network and computing resources. The information provided in the individual data frames allows the end nodes to derive specific UTC time, as well as other time scales that are typically used.

1.2.1 Concepts

In this section, there shall be described various definitions that are provided in the IEEE 1588-2019 standard as a way to outline the PTP.

Accuracy is a commonly referenced parameter when describing timing protocols. This parameter determines the mean difference between the value obtained at the end node and the reference clock value. The lower the difference between the measured time and the reference time, the higher the accuracy of the provided time. Considering that, the formula for PTP accuracy can be specified as seen in 1.1.

$$Accuracy = 100 - |\frac{Ref_t - Mes_t}{Ref_t} \cdot 100| \quad [\%] \tag{1.1}$$

The *Ref_t* stands for a reference time in an arbitrary time unit, *Mes_t* stands for a measured time in an arbitrary time unit. From this formula, the percentage accuracy of the measured time can be determined, considering that the measurement itself is accurate.

The **clock**, generally in the timing protocol context, can be specified as a time that has passed since the beginning of a defined epoch. The clock can be either physical or mathematical, where each of them is modeled differently [4]. There are other clocks in this case related to the concept of PTP, however, these types of clocks are not the same as the one specified above. All of the described clocks, however, operate with the clock described above. The **Master clock** is the main source of time in the network. This is the type of clock that usually obtains a signal from an external time source, which is for example the Grandmaster clock [5]. The Master clock is used to synchronize Slave clocks that are part of the same network or domain. The **Slave clock** is a type of clock that synchronizes with the Master clock and is not meant to provide timing to any other clock on the side where it

communicates with the Master clock. Both the Master and the Slave clock are concepts used in all timing protocols. It should also be noted that Master and Slave are relative concepts, meaning that the same node may have multiple functionality based on its configuration.

Moving on, to more PTP specific clocks, the Transparent clock shall be discussed. The **Transparent clock** can be described as a device that can be used to route the timing messages within the network [5]. One of its main functionalities is to correct a timestamp in the transmitted message, on its way to the end nodes. This is one of the ways for network delay to be calculated. Transparent clocks may have different functionality in the E2E and P2P delay modes. They are usually implemented within network switches or any other devices, which must support PTP and be non-blocking. A non-blocking switch is a type of switch that can handle a number of ports without interrupting the rest of the traffic. The network properties of the PTP shall be discussed more deeply in the later sections of the thesis.

The **Boundary clock** is a type of PTP clock that can take the functionality of both the Slave and Master clock [5]. It obtains a Sync message from the Master or Grandmaster clock, which contains the time. It then calculates the delay and adjusts the timing, creates a new packet, and sends it to a network. This way the number of nodes directly talking to the Master or Grandmaster clock is greatly reduced while the Sync message is recreated the same way as it would come directly from the Grandmaster clock.

The **Grandmaster clock** could be described as a main distributor of timing, to which other clocks in the network are synchronized. Grandmaster clock receives time directly from the external source, such as GNSS or GPS, and should be therefore considered as the first and only timing source in a network.

The **PTP domain** is described as a "logical grouping of PTP instances" [4, p. 17], where PTP is used to synchronize all the clocks in this domain to "Grandmaster Clock of the domain, but are not necessarily synchronized to the Local PTP Clocks in another domain" [4, p. 17]. Domains are a very important concept in PTP and other timing protocols, as they allow the specification of all PTP devices within a given network. This can be very useful when designing time-synchronized applications. A **PTP port** or **PTP interface** is a communication endpoint for PTP-related messages on a given device. A single device can have multiple PTP ports and it is possible for each PTP port on a single device to fulfill the role of different types of PTP clocks.

The following concepts are not strictly PTP-related but shall be mentioned throughout the work, it is therefore important to explain them. A **bit** is defined as

the smallest unit used as an increment of data in a computer [6]. The value of bit can be either 0 or 1, which from the electronics point of view means either on or off. Bits may also be mentioned when referring to a certain position within a byte. A **byte** is the second smallest unit after bit. A byte is simply a group of eight bits. An **octet** is a term that can be used interchangeably with a byte. The usage of both terms can be seen in different literary sources. The indexing of bits in byte/octet starts with 0 and ends with 7.

Another important concept that is commonly used in this thesis is the **TCP/IP** five-layer network model. This model describes and differentiates technologies used for data transmission in physical and logical networks. The original TCP/IP model, as opposed to the older OSI model, consists of only four layers. However for the sake of clarity, a new version of this model shall be used in this thesis, which contains fiver different layers. The layers are hierarchically lined up based on their functionality. Starting from the lowest layer to the highest layer, the layers are as follows: **Physical Layer, Data Link Layer, Network Layer, Transport Layer, Application Layer** [7]. The Physical Layer is concerned with signal transmission using either physical wire or a wireless medium. The Data Link Layer is concerned with creating frames with *Medium access control* (MAC), that are always transmitted between adjacent devices or in the *Local Area Network* (LAN) with the help of Data-Link switches [7]. The Network Layer creates logical groupings called networks, where the connected nodes are differentiated by IP address, which determines how should be the messages sent to the destination nodes. The Transport Layer is concerned with the management of the message transfer itself. The Application Layer is the upper layer which is used by the applications requiring the network connection [7]. Each of these layers is encapsulated into each other, starting from the lowest to the highest, and each interacts differently in the process of communication. Knowledge of all TCP/IP layers is important for understanding the described concepts. Throughout the thesis, more concepts may be mentioned, which are either very specific for given applications or are considered to be general knowledge.

1.2.2 Operations

The principle of PTP operations is based on sending special timestamp messages, to and between entities communicating within a specific PTP domain. Synchronization in networks is based on the measurement of time shift caused by message propagation delay [8]. However, before the synchronization process itself is described, it is important to explain how the Slave clock determines its time source in case there is more than one Master or Grandmaster clock available.

Tab. 1.1: PTP BMCA Parameters

Parameter	Explanation
priority1	The primary parameter for selecting the clock source based on priority [9]. With a lower value, the priority is higher.
clock class	Quality of the clock source. With a lower value, the quality is better.
clock accuracy	Accuracy of the clock source in parts per billion (ppb) unit. With a lower value, the accuracy is better.
PTP variance	Variation in the clock source's frequency over time. With a lower value, the stability is better.
priority2	The secondary parameter for selecting a clock source based on priority. With a lower value, the priority is higher.
clock identity	Identifies the clock source. Distinguishes between the clock in the network.
distance (number of boundary clocks)	Number of boundary clocks that are between the Master clock and the Slave clock. With a lower value, the distance is smaller

1.2.2.1 Best master clock algorithm

The hierarchical position of the Master clock as a time source in the network topology can be either configured in a statical way or it can be generated using *Best Master Clock Algorithm* (BMCA) as defined in IEEE 1588. This algorithm runs in a separate manner on each node in a PTP domain. This means that nodes do not negotiate the hierarchical position of each PTP instance between each other, instead, they only compute the state for their own PTP port [9]. It is also important to note that the Best Master Clock Algorithm runs on each node continuously so that eventual changes within the PTP network can be reflected to PTP instances immediately, in case they were to occur [9].

For the purpose of Best Master Clock Algorithm, the special type of source ad-

vertisement message is defined, which is called the *Announce message*. This message contains data used for advertising the Grandmaster clock in regards to preferability [10]. Based on the information, received from this advertisement message, it is up to the end node to determine which clock is to be the Master clock. The preference settings for the PTP clock, when selecting the best potential Grandmaster clock, as defined in IEEE 1588, is based on these parameters: *priority1, clock class, clock accuracy, PTP variance, priority2, clock identity, distance (number of boundary clocks)* [11]. An explanation of each parameter can be seen in the table 1.1. The selection of the parameters may however be modified depending on the specific needs of the PTP clock.

After the PTP node receives multiple *Announcement message*s from different Master clocks, it looks at the values as specified in the table 1.1. As described in the IEEE 1588, the data set comparison in the BMCA works on the principle, that values provided by two Announce messages are sequentially compared. If the values in parameters match or are within a tolerance, the comparison function moves to another value. Once one attribute in the Announce message has a better value, than the ones provided by different Master clocks, the clock that sent that specific message is chosen by the PTP node as the Master clock. The lower value usually means the better quality of the clock.

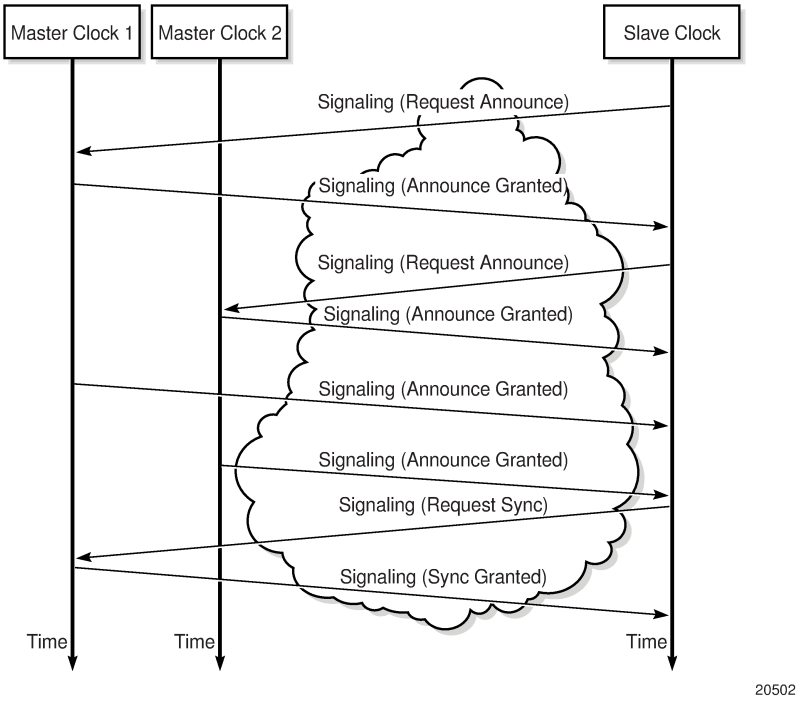


Fig. 1.1: Messaging between PTP Slave and master clocks [11]

In figure 1.1, the situation can be seen where the Slave clock communicates with two different Master clocks. As can be seen, the PTP node sends a multicast request for an announcement, which is basically just a Sync message, to which are the Master clocks required to reply with the *Announce* message. Alternatively, the Slave clock simply listens on a specific port until some Master clock advertises itself. The Master clock can either periodically send *Announce*, *Sync*, and *Follow_Up* messages to multicast address, or it can wait for a request from the Slave clock. When the Slave clock obtains from the Master clock the *Announce* message, which contains all the specific parameters needed for the BMCA, a PTP node can run the algorithm and determine the best Master clock as a time source. Once the best time source is determined, the synchronization process can begin.

1.2.2.2 Time synchronization in End-to-end delay mode

As was described in the Concepts (1.2.1) subchapter, clocks in the PTP network, have a predefined hierarchy where each type of clock has a different role in the time-synchronized network. This subchapter focuses specifically on the End-to-end delay mode, where the delay is calculated between the starting and ending devices, that is between the Grandmaster clock, which is here denominated as a Master clock and the PTP end-node, that is to be synchronized, which is here described as a Slave clock. In E2E delay mode, all the possible delays between the Master clock and the Slave clocks are accounted for. It can be even assumed that devices, which are between the Master and Slave clock, may not support PTP. In comparison to Peer-to-Peer mode, the End-to-end delay mode may however suffer from lower precision regarding offset determination.

The most important is, for clock synchronization, the *Sync* message. This message is sent to multicast from the Master clock to all Slave clocks which are part of the given PTP domain and are to be synchronized with the Master clock. The interval of *Sync* messages is determined in configuration, but usually, it is something between 1 to 65 messages per second [10]. Synchronization begins when the *Sync* message is sent from the Master clock to the Slave clock. Hardware or software created timestamp *t1*, is used to mark down the time when the *Sync* message is transmitted. The moment the *Sync* message is received on the Slave clock side, the timestamp *t2* is created, which contains the specific time when the message arrived [4].

If a **one-step** synchronization mode is to be used, just a *Sync* message would be sent and other messages would be disregarded on the receiver side. In this case, the timestamp *t1*, would be part of the *Sync* message itself. This would

require very high precision hardware processing, as it would require the master clock to create timestamp *t1* at the time of sending and at the same time insert it into a *Sync* message [4]. In **two-step** synchronization mode *Follow_up* messages are sent right after the *Sync* message. The Follow_up message contains time *t1*, which was previously created when *Sync* was sent. This way, *t1* is conveyed with the same accuracy but two different messages are used. The issue with two-step communication may be the greater communication overhead, caused by the greater number of messages.

Communication continues by the Slave clock sending the *Delay_Req* message to the Master Clock. This message is usually sent to the Master clock in periodic intervals [13]. Similarly to the *Sync* message, the timestamp *t3* is created on the Slave clock side when the *Delay_Req* message is sent to Master. After receiving the message, the Master clock creates timestamp *t4*, which notes the time when was the *Delay_Req* message received. Finally the *Delay_Resp* message, which contains the timestamp *t4*, is sent from a Master clock side to the Slave clock [9]. The whole process of the above-described communication can be seen on the figure 1.2. Note that "t-ms" as seen in the figure means the time from Master to Slave and "t-sm" means the time from Slave to Master.

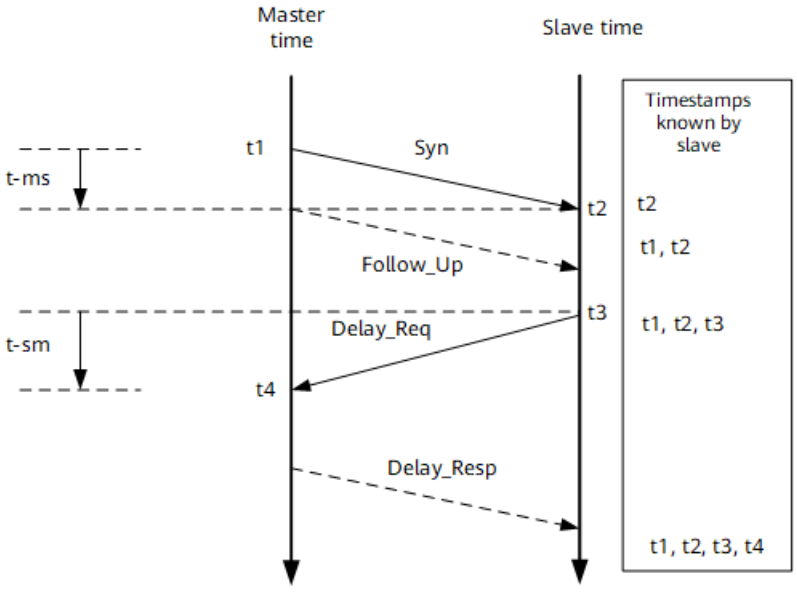


Fig. 1.2: PTP synchronization process in E2E[13] (t is same as *t*)

After all messages are exchanged, the end PTP node has all the previously saved timestamps, *t1*, *t2*, *t3* and *t4*. These can be afterwards used in order to calculate the delay and offset between the Master clock and the Slave clock.

The sum of the bidirectional delay between the Master clock and the Slave clock is calculated on the Slave clock side that has all the required timestamps [13]. This delay represents Round-trip time, which is the time of how long it takes for a network request to go "from a starting point to a destination and again to a starting point" [14]. By using the equation 1.2, the exclusive delay which is caused by the transmission link, not the time offset, can be determined. In this case, it is assumed that delay is symmetric, in other words, that time t between sending of *t1* and marking of *t2* equals to time t between sending of *t3* and marking of *t4*.

$$Bidirectional\ delay = (t4 - t1) - (t3 - t2) \tag{1.2}$$

The unidirectional delay on the link between the Master and Slave clock devices can be described the same way as the previous formula [13], however, the calculated delay is divided by two, considering that it is now only a single direction delay, which does not take into account the time taken for a reply to come back. The formula can be seen in 1.3.

$$Unidirectional\ delay = \frac{Bidirectional\ delay}{2} \tag{1.3}$$

Unidirectional delay could be expressed using other formulas as well [9]. By using the formula 1.4, the same results would be obtainable as with the equation above. Therefore it is completely arbitrary which implementation for delay calculation would be used. Some materials may specify either the equation defined above or the one below, however, they can be used interchangeably.

$$Unidirectional\ delay = \frac{(t2 - t1) + (t4 - t3)}{2} = \frac{(t2 - t3) + (t4 - t1)}{2} \tag{1.4}$$

The offset could be described as a time difference between the Master and the Slave clock, considering that all times are measured at the same instant [9]. By knowing the specific offset value, the Slave clock can determine how much it is supposed to correct the time to match the Master clock. If times *t1* and *t2* were taken as example times and delay would be added, the resulting formula would be as seen in 1.5.

$$Offset = t2 - t1 - Unidirectional\ delay \tag{1.5}$$

Based on the determined offset value, the internal clock of a computer is readjusted so that it matches with the Master clock. Readjusting of the internal clock may depend on the specific PTP software of hardware implementation, and how it

manages its internal time. If the offset equals to zero it means that the Master and Slave clocks are perfectly synchronized.

It should also be noted that if the Transparent clock is placed in the network path between the Master and the Slave clock, the Correction field mechanism can be applied. This mechanism works by subtracting the arrival time on the Transparent clock *t2* from the departure time *t1*, and adding the result value to the Correction field value in the *Sync* packet itself. The Correction field value is afterwards subtracted from the final time at the Slave clock side. This way the delay can be determined with greater precision considering that Transparent clocks provide delay information themselves. This method is however more relevant in the P2P delay mode, which is described in the following sub-chapter.

1.2.2.3 Time synchronization in Peer-to-peer delay mode

The synchronization with Peer-to-Peer or Peer delay mode works in many ways same as with E2E mode with the main difference being the fact that delay is calculated between each adjacent device in the network, considering that there are only devices in the network which are PTP-enabled, which means that they act as a Transparent clock, and they all use the same delay mechanism. In Peer delay mode, the delay calculation is done with a set of specialized messages, which are unrelated to the Synchronization process itself as it was in the previous case, but are used solely for the calculation of delay between two adjacent nodes. Peer delay determination can be initialized by either a Master or a Slave clock, thus in this case nodes shall be denoted simply as Node 1 and Node 2 [13].

The process starts with Node 1 sending the *PDelay_Req* message to Node 2 and marking the time of sending as *t1*. The arrival time of the *PDelay_Req* message to Node 2 is marked down as *t2*, and this time is subsequently added to the message data. The Node 2 afterwards responds with *PDelay_Resp* message marks down the time of sending *t3* and includes it to the message if the one-step mode is used. Node 1 upon arrival of *PDelay_Resp* message, marks down the time *t4*. In the case that a two-step mode is used, one more additional message is sent from Node2 called *PDelay_Resp_Follow_Up*, which contains the previously created time *t3*.

The result from this operation is that Node 1 has all the relevant timestamps and is able to calculate unidirectional and bidirectional delay the same way as it was done in E2E mode [13]. Considering that delay is calculated between each node on the network, it is much easier to determine which nodes are behind the asymmetry of the delay and which nodes have the biggest delay between them. Also since the peer delay messages are sent only between the neighbouring nodes, the Master clock

devices are not unnecessarily overwhelmed [15].

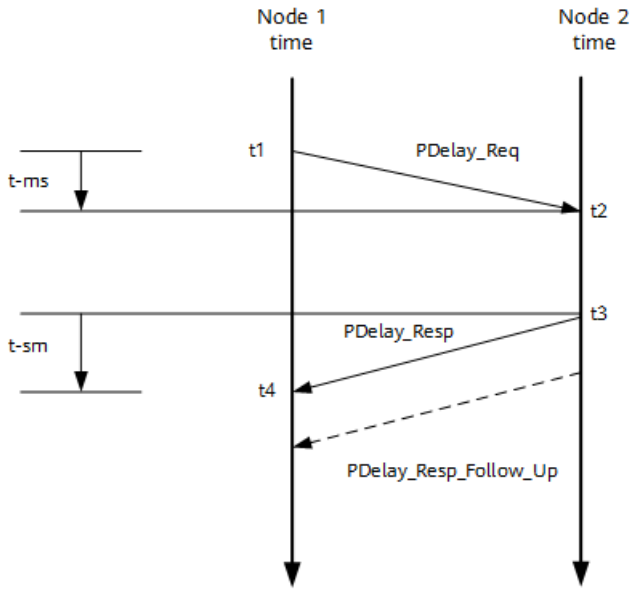


Fig. 1.3: PTP synchronization process in P2P[13] (t is same as t)

As was said before, the above-described process is unrelated to the clock synchronization itself and serves solely for the purpose of delay calculation between the P2P Transparent clocks. Synchronization still needs to be done the same way as was described in E2E mode, with the exception that only *Sync* and *Follow_up* messages are required to be recognized by the network and *Delay_Req* and *Delay_Resp* are not needed [4]. This means that once both Node 1 and Node 2 exchange the set of messages as seen in figure 1.3, the Master clock sends the *Sync* message in order to Synchronize with the Slave clock. Note that "t-ms" as seen in the figure means the time from Master to Slave and "t-sm" means the time from Slave to Master. It should be also noted that Delay estimation can be done both for downlink and uplink communication [13].

When the *Sync* message passes through the node, which is a Transparent clock, the calculated delay is added to the value in the Correction field, in addition to the difference between the departure time and arrival time to the Transparent clock [4]. The following process is done for every Transparent clock device, considering that with each passing, the uplink node becomes Node 1 and downlink node becomes Node 2 and so on. Once the *Sync* message arrives at the Slave clock, the Correction field is subtracted from the arrival time *t2*, as well as *t1*, which is in this case only value needed for offset determination, considering that the Correction field mechanism is in the P2P delay mode guaranteed to be precise.

1.2.3 Network properties

In the previous sections, data communication between nodes was described simply as messages, for the sake of simplicity. These messages, however, have certain properties and hold a certain form. The messages can be transmitted using different layers of the network, as described by IEEE 1588, which allows different PTP encapsulation modes. In the following section, the different methods of PTP message encapsulation as well as their structure shall be explained.

1.2.3.1 Encapsulation properties

PTP messages need to be encapsulated into another protocol, which could serve as a transport protocol for the PTP information within the network. The common means of transport for PTP messages is a Transport Layer protocol called *User Datagram Protocol* (UDP). However with the newer implementation of the PTP, as described in the IEEE 1588v2, the PTP messages can be encapsulated within the Data Link Layer using Ethernet protocol.

The main difference in the above-specified modes is the performance and convenience. Considering that the Ethernet encapsulation mode is closer to the Physical Layer than UDP encapsulation mode, it can be expected to have better performance, considering the lower overhead of the messages.

1.2.3.2 UDP/IP encapsulation

The encapsulation of the PTP messages into UDP, is a method to be used when a PTP packet is to be transversed inside or between IP-based networks [16]. The UDP packet containing the PTP message can be encapsulated in IPv4 or IPv6, allowing PTP to be used outside of Local Area Network on devices that do not necessarily support over-the-ethernet transport for PTP. The port numbers that are used for UDP/PTP messages are **319** for Event messages and **320** for General messages sent either through Unicast or Multicast [4]. Figure 1.4, shows the structure of the data encapsulation within the IP packet, which remains the same for IP versions 4 and 6.

By default, PTP messages shall use the following addresses for multicast communication. In the IPv4 network for Peer delay messages, the network address **224.0.0.107** is used, and for all the other messages **224.0.1.129** address is used [9]. In the IPv6 network Peer delay messages, the network address **FF0X:0:0:0:0:0:181** is used, and for all the other messages **FF02:0:0:0:0:0:0:6B** address is used [9].

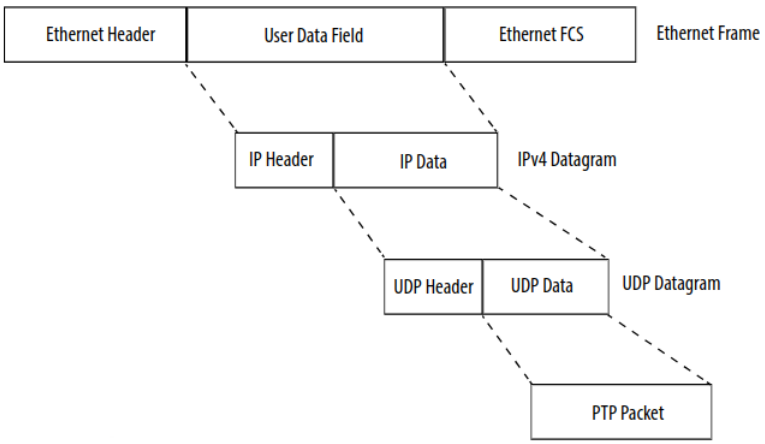


Fig. 1.4: PTP encapsulation in UDP/IP [16]

1.2.3.3 Ethernet encapsulation

Encapsulation of PTP data in an Ethernet header is the preferred mode of encapsulation since, apart from the potential performance benefit already mentioned, the deployment is significantly easier compared to PTP over UDP/IP [17]. The usage of the Ethernet as a transport protocol for PTP allows only communication within one Local Area Network, where all nodes in the PTP network function as some type of clock, described in the chapter 1.2.1.

The Ethernet frame, as defined in IEEE 802.3, is used for all Medium access control communications. The header consists of several parts, each with a specific size, see figure. The most relevant ones in regards to PTP packet transportation are however: **EtherType**, **destination MAC**, **source MAC**, and **payload**.

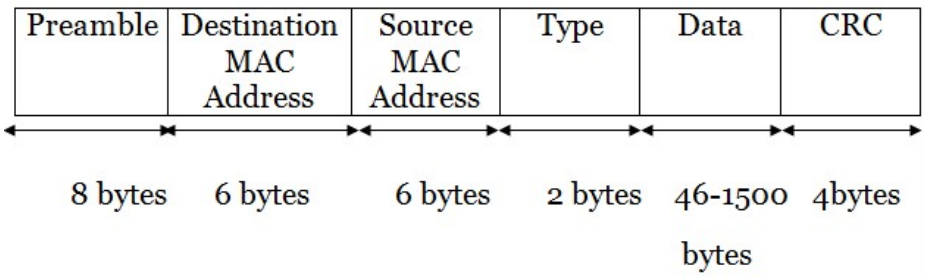


Fig. 1.5: Ethernet header [18]

The EtherType is a specific Ethernet header field that specifies the type of pro-

tocol that is encapsulated in the Ethernet frame payload. In this field, which is two bytes long, are usually specified most common network protocols, such as IPv4 or IPv6. However, in the following case type is defined as **0x88F7**, which stands for PTP over IEEE 802.3 [4].

The MAC, which is used in the following two fields in the Ethernet header, is a specific number identifier used for Data Link Layer communication within the Local Area Network [19]. The MAC address itself is a 48 or 64-bit hexadecimal number in the given form **XX-XX-XX-XX-XX-XX**, where the first 3 bytes represent Organizational Unique Identifier approved by IEEE, and the rest is given by the device manufacturers [19]. The manufacturer also assigns MAC addresses to *Network interface controller* (NIC), of the device. These addresses are considered permanent for each NIC, and each device in the network should have completely unique MAC address.

Moving back to the Ethernet header structure, the destination MAC address field in the Ethernet frame determines the destination device for the given Ethernet frame. In relation to that the source MAC address defines the source device of the given Ethernet frame. As the frame travels through the local network, the destination of the device always remains the same, if it is considered that as a destination only multicast PTP addresses are used. If that frame would transverse through the router in Local Area Network, upon leaving the router, the source MAC address would become that of a router.

For communication over Ethernet, PTP uses two specific multicast addresses, similarly as in the case of PTP communication over UDP/IP. These addresses are understood by all PTP-enabled devices in the network, and each determines the different behavior of the devices interacting with the PTP packet. The first multicast address **01-1B-19-00-00-00**, is a "standard Ethernet MAC address that is expected to be flooded by all types of Ethernet bridges and switches and also by a large number of base station vendors" [17]. This first 3-byte part of this address represents OUI given by IEEE, and the rest of the address is derived from the "pool of multicast addresses within that given space" [4, p. 387]. Frames with this destination address can be forwarded even by devices that do not necessarily support PTP. Therefore, this multicast address can be used for all PTP delay-related messages, except Peer delay messages. The second multicast MAC address **01-80-C2-00-00-0E**, is derived from the pool of multicast addresses given by the IEEE 802.1Q standard. Frames with this destination address are required to not be forwarded to other ports of interfaces, as this address is meant only to be used in the case of Peer delay mode-related messages [17] [4].

The final important field in the Ethernet header is the payload or data part. This field can be up to 1500 bytes in size, and it contains the PTP message. The structure of the PTP message itself will be described in the following section.

1.2.3.4 PTP header structure

All messages related to PTP have a common header structure [4]. This means that all messages have the same fields within the header, unrelated to what type of message is being sent. It also remains the same whether the PTP message is encapsulated into UDP or Ethernet. Apart from the header, there is also a message body and suffix, of which more shall be described later.

As can be seen in the figure 1.6 taken from the IEEE 1588 standard document, there is a number of header fields, where each of them occupies a fixed number of bytes/octets. By summing all the sizes of the fields, the total size of the header itself is 34 bytes. The values in the Octets column describe the total size of the field or fields in bytes/octets located on the same row as the value. The Offset in the header is a value that is used to indicate the index of the field in the header, based on the assigned size in bytes. The Bits in the header determine the specific position of the field within the byte that is in size less than one octet. This means that *messageType* starts on the first bit with index zero and *majorSdoId* starts on the fifth bit with index four.

Bits								**Octets**	**Offset**
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**		
majorSdoId				messageType				1	0
minorVersionPTP				versionPTP				1	1
messageLength								2	2
domainNumber								1	4
minorSdoId								1	5
flagField								2	6
correctionField								8	8
messageTypeSpecific								4	16
sourcePortIdentity								10	20
sequenceId								2	30
controlField								1	32
logMessageInterval								1	33

Fig. 1.6: PTP header structure [4]

The first field in the PTP header, *majorSdoID* together with *minorSdoId* consists of the *sdoId*, which provides isolation of PTP Instances between different Qualified

standards development organizations [4]. The *messageType* field describes which specific type of message is contained within the header. As was previously mentioned several times throughout the thesis, there are several types of messages, and each fulfills its own function in different PTP processes. The numbering is in hexadecimal format. Taking into account the number of messages, including the reserved spaces, the first hexadecimal number is 0 and the last is F. The messages are further divided into the Event class and the General, which are determined by the most significant bit [9] of the message.

The fields *versionPTP* and *minorVersionPTP* represent which version of the PTP protocol is used by devices on the network, so compatibility can be ensured. The *messageLength* represents the number of bytes that are used to represent the message, which includes the header, body, and suffix [21]. The *domainNumber*, is used to identify which PTP domain the PTP message belongs to. This means that PTP messages shall only be exchanged among clocks that belong to the same domain as specified in the message. The *flagField*, represents various conditions or settings, which differ based on the message type. The specific function of the flag is determined based on which octet and which bit in the given octet is set, as well as which type of PTP message is used. The *correctionField*, contains the correction value represented in the nanosecond scale. This value is determined by the time the packet spends in the Transparent clock and is used for the delay calculation [21]. In case the value of the correction field value is too large to be represented, it contains ones in all bits except the most significant bit. The *messageTypeSpecific* field is solely for the internal usage of the PTP protocol within the components outside of the main PTP synchronization process [4].

The *sourcePortIdentity* field is used to identify the source PTP port that sent the message. The *sequenceId* field contains the sequence number for individual types of PTP messages [21]. The *sequenceId* value in each consecutive PTP message of the same type is always increased by one. This means, that each PTP port maintains *sequenceId* pool for each type of PTP message in use [9]. There are however exceptions for selected types of messages, for which the pool is not maintained. As described in the IEEE 1588, these messages are Pdelay_Resp, Follow_Up, Delay_Resp, Pdelay_Resp_Follow_Up and PTP management messages. The *controlField* is related to the hardware compatibility of the PTP devices. However, in the newer implementations, this field is to be considered obsolete. The last field in the PTP header is *logMessageInterval*. This value represents the second logarithm of the interval for sending of PTP message. This value may however vary based on which type of message is used.

Apart from the header, the PTP packet consists of the PTP payload, which shall be described in the following subchapter, and the suffix. The suffix in the PTP packet consists of *Type Length Values* (TLV). These can be described as "management messages, which are used to configure the protocol settings" [22]. There is a possibility to use TLV as an authentication method, in order to increase the security of the PTP implementation. However, in case the addition of TLV, would cause frame size to exceed its maximum limit the TLV would not be added to the PTP packet [9].

1.2.3.5 PTP message structure

The PTP message itself comes after the header and before the suffix. The message contains the main part of the desired PTP functionality. To start from the simplest PTP messages first, Sync and Follow_Up messages shall be described. The Sync message apart from the header contains *originTimestamp* field. The *originTimestamp* is a ten-byte field that contains the precise time of the creation of the packet in nanoseconds. Similarly, the Follow_Up message contains a ten-byte *preciseOriginTimestmap* field, which is used in the case that the *originTimestamp* value in the Sync message is not included.

The Delay_Req message consists of a ten-byte *originTimestamp* field and a ten-byte reserved field. The *originTimestamp* has the same meaning as in the Sync message. The reserved field is there for a reason so that the total length of the PTP message is the same as that of Delay_Resp message [4]. The Delay_Resp message consists of a ten-byte *receiveTimestamp* field, which contains the time at which was the Delay_Req message received on the slave device, and a ten-byte *requestingPortIdentity* field, which contains the port identity of the PTP port which sent the Delay_Req message. The Pdelay_Req has the same structure as the Delay_Resp message. The Pdelay_Resp contains the ten-byte *requestReceiptTimestamp* field which contains the time at which was the Delay_Req message received by the device which sends the Pdelay_Resp message. The Pdelay_Resp_Follow_Up message consists of the ten-byte *responseOriginTimestamp* and the ten-byte *requestingPortIdentity* fields. The responseOriginTimestamp field contains the timestamp of when the PDelay_Resp message was sent, in case the requestReceiptTimestamp value is not included in PDelay_Resp. The requestingPortIdentity field has the same meaning as in the previous messages. The context and functionalities of the above-mentioned messages are closely described in the chapter 1.2.2.2 and the subchapters after that related to the synchronization process.

The Announce message, which is related to BMCA, consists of considerably more

fields than the previous messages. The ten-byte *originTimestamp* field is either value zero or an estimate of no less precise than one second of when the Announce message was transmitted [4]. The two-byte *currentUtcOffset* field specifies the offset of the clock, sending the Announce message, from the UTC [9]. The one-byte fields *grandmasterPriority1* and *grandmasterPriority2* correspond to values *priority1* and *priority2*, which are explained in the table 1.1. The four-byte *grandmasterClockQuality* and eight-byte *grandmasterIdentity* fields correspond to *clock class* and *clock identity* respectively, which are explained in table 1.1. The two-byte *stepsRemoved* field, contains the number of how many PTP Communication Paths were passed between the Grandmaster clock and the slave device, with the initial value being zero [9]. The final one-byte *timeSource* field describes the time source that is used by the Grandmaster clock.

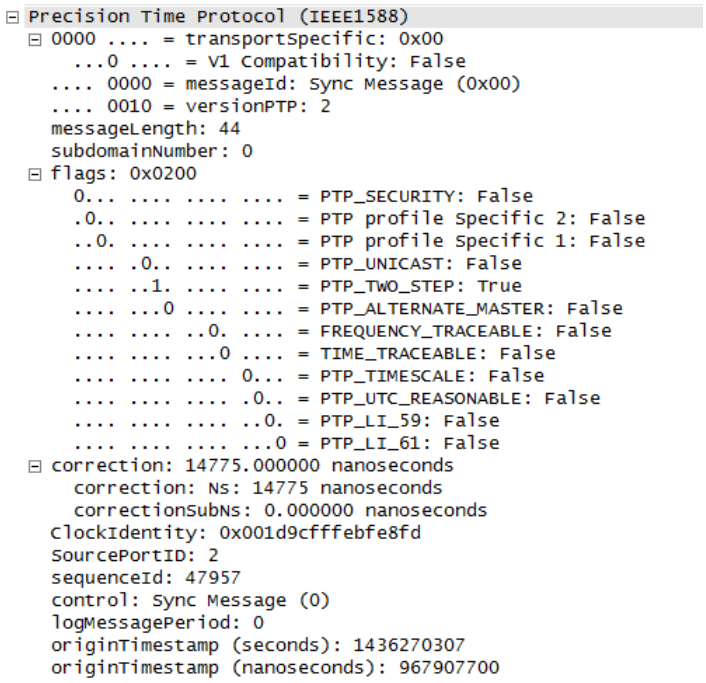


Fig. 1.7: PTP Sync packet example [20]

1.3 Use case scenarios and applications

As was mentioned before, PTP finds its use case in many fields and applications that rely on precise time synchronization. How exactly are the timestamps delivered in PTP messages used may differ significantly from industry to industry. In the manufacturing industry, PTP may be used in many different tasks, such as log-keeping

or machine coordination and synchronization. When PTP is used for surveillance purposes there will be a need to precisely synchronize the audio and video streams. However, for the sake of simplicity, this subchapter shall focus mainly on depicting the topology, which would be used in real-case scenarios.

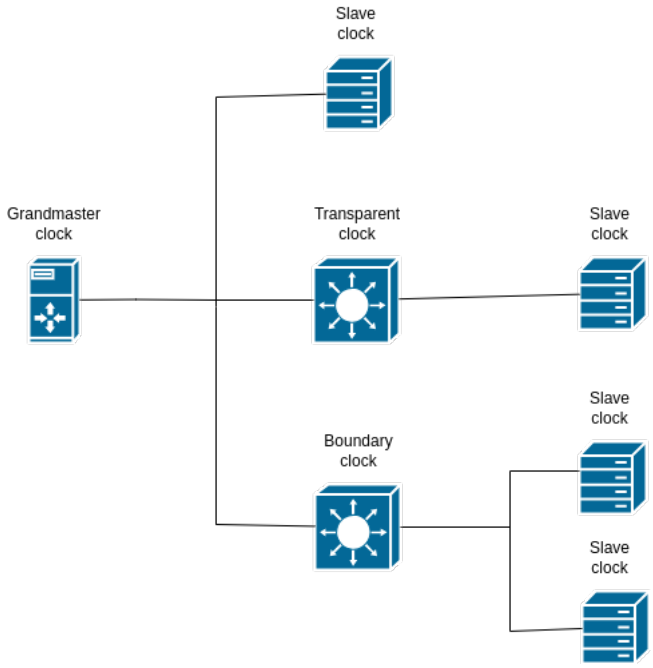


Fig. 1.8: Simple topology with all clock types

Looking at the very basic scenario in figure 1.8, it can be seen that all clock types were used in this case. Let us consider that the Grandmaster clock sends a Sync message to all the PTP devices via multicast, assuming that end-to-end synchronization mode is to be used. Looking at the figure from top to bottom, once the Slave clock, receives the Sync message the standard synchronization procedure follows. The first Slave clock is connected directly to the Grandmaster clock, from the PTP point of view, therefore no extra factors in the path need to be considered. In the case of the second Slave clock, the connection to the Grandmaster goes through the Transparent clock. The Transparent clock will simply replicate the Sync message to its outgoing ports and add the value of time the packet spent on the Transparent clock device to the Sync message correction field, which is to be used later in delay calculation. More information can be found in the section 1.2.2.3, which is concerned with this topic.

The last two Slave clocks are connected to the Boundary clock instead of the Grandmaster clock itself. The Boundary clock in this case acts as a Master clock to all the Slave clocks which are connected to it. This way it is ensured that the

Grandmaster clock is not overburdened and the network is more scalable. The Boundary clock however acts as a Slave device to the Grandmaster clock and its time is also being synchronized. In contrast to the Transparent clock, the Boundary clock can provide a use case in scenarios when different VLANs are needed to be used on different ports [23]. Using too many boundary clocks may however lead to inaccuracies as the precision of the Boundary clock always depends on the Master clock which provides synchronization [23]

Now, let us consider a little bit more complex scenario as seen in figure 1.9, which involves a greater number of devices. In this case, the Peer delay synchronization mode is used. Seeing that there is more than one Grandmaster clock available, the BMCA will need to be used in order to determine the most suitable Master clock. In the current scenario, there are only ten Slave clock devices, however with a higher number of devices the network could become easily overburdened if each device were to run its own instance of BMCA. The Boundary clock will act as a Slave device, outside of the local network, and negotiate the most suitable Grandmaster clock, using BMCA as described in the sub-chapter 1.2.2.1. The second Grandmaster clock can be however used as a backup, in case the primary time source fails. This means that the Boundary clock is being synchronized to one of the Grandmaster clocks, while it acts as the Master clock inside the local network. The BMCA can be therefore disabled on the end-node Slave clocks.

Considering that the delay mode is Peer-to-Peer, the delay calculation begins between the Boundary clock and each of the Transparent clocks. Also, the delay is calculated between the Transparent clock and each Slave clock, using the Peer delay mechanism calculation. When synchronization itself is to be done, the Sync message is sent through the Transparent clocks. Note, that there may be used more Transparent clocks than is shown in the topology, with significantly more Slave clocks. When the Sync message passes through the Transparent clocks the calculated delay is added to the Sync message correction field, together with the passage time. Once the Sync message arrives to each Slave clock the time can be corrected based on the final offset. More information on Peer-to-Peer mode can be found in sub-chapter 1.2.2.3.

The above-described topology can be considered to be a standard scenario in many industrial settings. Of course, there might be even more synchronization layers, using Boundary clocks, than in the current scenario, however, the idea remains the same. The Boundary clock can be also connected directly to Slave clocks for better management, however with the increasing number of Boundary clocks, time management and error-safety might prove to be more difficult. It would be therefore

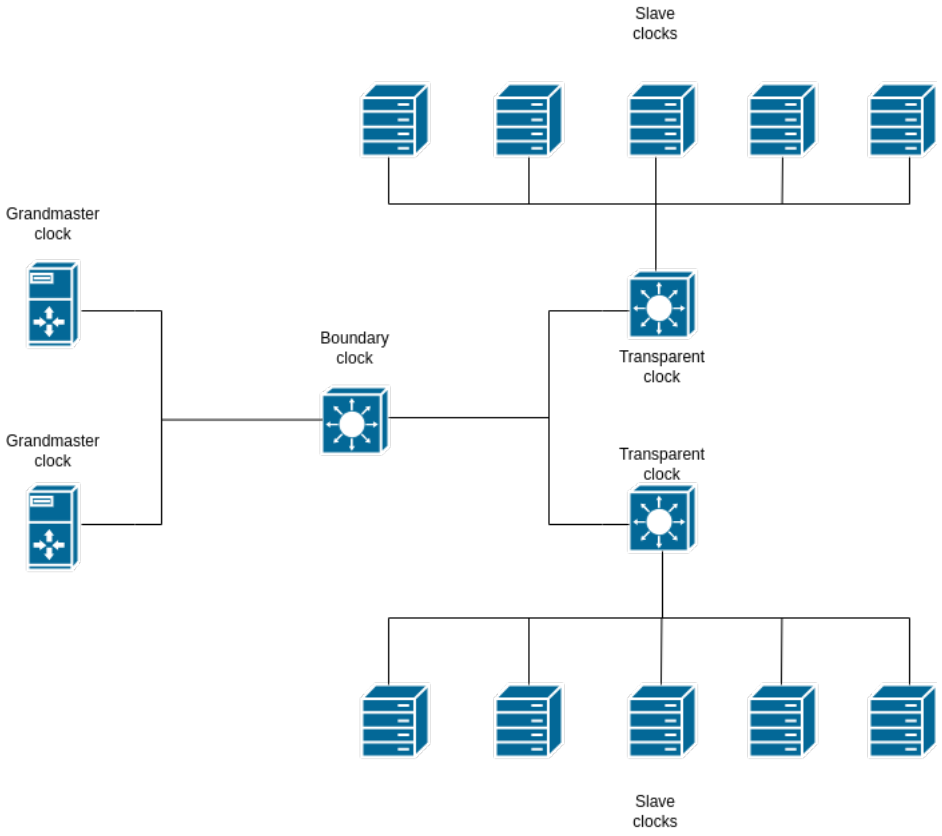


Fig. 1.9: Topology with multiple layers of synchronization

recommended to use Transparent clocks if better accuracy is needed while preventing unnecessary layering with the Boundary clocks. The Boundary clock is better to be used in order to lighten the load from the Grandmaster clock or the Master clock in the given network. If for example there were too many devices connected to the Boundary clock in the topology seen in figure 1.9, it would be optimal to connect another Boundary clock which would act as a Master in its own domain, and would lessen the burden on the previous Boundary clock. Another issue, that needs to be considered is of course the cost and performance of the devices. This would however depend on the specific industry and the budget of the implementation.

2 Security protocol selection

Network security is a vast topic, however, this chapter shall be concerned mainly with the encryption of specific communication channels. Considering the scenario where the physical network is shared with other subjects or is public, it is very much needed that secure communication channels are established in order to convey PTP messages and other related traffic. Generally speaking, the traffic, which needs to be encrypted is encapsulated in a network security protocol, which takes care of the secure transfer of communication. Considering that PTP itself can be encapsulated under different layers of a network, a different security protocol needs to be considered for each use case. Each protocol will encrypt only the data that are located on the same layer of a network and the layers above them. The performance of the applications may also differ based on which protocol is used. This chapter shall concern itself with the description of security protocols and other security-related aspects.

2.1 Security protocols for PTP over UDP

When the PTP is encapsulated inside the UDP transport protocol, the security protocol which can be used to encrypt the communication, needs to operate either on the Transport Layer or the Network Layer. Even though these protocols can provide sufficient protection against external attacks, their biggest limitation is the potential effect that they have on the PTP's efficiency and accuracy. Due to the fact that the encryption itself happens on the Network Layer, the intermediate devices are not able to access the contents of the PTP message. This essentially disables the functionality of the Transparent clocks, which aren't able to interact with the PTP messages, unless they are able to decrypt the packet payload. For this reason, only the synchronization with E2E delay mode can be used, unless the Transparent clock device is appropriately configured as well. [24].

Speaking about the implementation, one of the simplest ways to encrypt the general traffic between two points in the network is by creating an encrypted network tunnel for communication. All the devices which connect to *Virtual private network* (VPN), are perceived as if they were inside the local private network. In this subchapter, we shall look at the different security protocols, which are suitable to be used when the PTP communicates over UDP.

2.1.1 Wireguard overview

There is a significant number of VPN protocols and services that are able to provide specific needed functionality, however, for the sake of the thesis the Wireguard has been chosen as one of them, due to its simplicity, reliability, and popularity. The Wireguard is an open-source VPN protocol, created as a replacement for OpenVPN and IPsec, which works by encapsulating the traffic in an additional encrypted UDP header under the IP header [25], thus making the whole communication connectionless. This means that there is no need for communication to go through the connection establishment process before it starts. This attribute can be considered to be a very useful feature, especially in cases where the network connection is not stable. However, in scenarios where the network is stable, this does not play any significant role. The encryption options with Wireguard are limited compared to other services, considering that Wireguard enforces only the default encryption protocol.

2.1.1.1 Wireguard underlying operations and properties

For Wireguard in order to communicate with other devices using VPN a separate virtual interface needs to be created. This interface acts as a standard network interface and can be managed that way. Note that for any PTP-related communication, the virtual interface would have to be used in order for traffic to be encrypted.

The user identification is done strictly using the 32-byte public key, given by the elliptic curve Curve25519 [26]. The virtual interface has an assigned IP address belonging to the VPN network address range and an open UPD port. It also has assigned private and public key pairs, where the private key is used for decryption and the public key for identification. The algorithm which encrypts the payload is called ChaCha20 [26]. The Wireguard interface must also know the public key of the host with which the interface communicates. Because accepted IP addresses are linked to known public keys, the Wireguard interface will not accept any communication coming from devices that do not have their IP address linked to any known public key [26]. The virtual interface will also use the known public keys to encrypt the outgoing communication, based on the destination device. The public keys of known devices can be added manually to the virtual interface or they can be securely exchanged through the network. In case that the destination IP address, of a packet that would be sent through the virtual interface, does not match any known public key, the packet is dropped [26].

However, before any encrypted communication starts the initialization must be performed. The starting handshake is done in a single RTT using NoiseIK handshake

based on the Noise protocol, which is based on the Diffie-Hellman key exchange algorithm [27]. These are the steps that must be secured before an actual PTP-related communication happens. All the packets which are to be sent, before the encrypted communication channel is established, are queued. The process of establishing an encrypted channel through the virtual network interface is done when initiating and receiving nodes exchange a set of messages.

2.1.2 IPsec overview

In contrast to Wireguad, IPsec is not a single protocol or a VPN service. It can be rather described as a set of open-source protocols used for establishing encrypted secure connections between connected devices. The IPsec can be either VPN-based on policy-based, depending on the specific implementation. It is also generally more complex to manage and implement, considering the much bigger codebase and much broader encryption options, which requires the user to have a knowledge of cryptography standards in order to make sure that the chosen settings do not compromise the security [25]. The IPsec, unlike Wireguard, is connection-based, meaning that for devices in order to communicate securely using IPsec encryption, a connection has to be established using connection-establishing mechanisms, or if the connection is interrupted, it has to be re-established. This however poses no problem if the network connection is stable. The IPsec encapsulation of data depends on which IPsec mode is to be chosen.

2.1.2.1 IPsec underlying protocols

There are several underlying protocols that create the functional IPsec [29]. The first protocol is *Internet Key Exchange* (IKE). This protocol is used to exchange the required security parameters for secure communication to be established [29]. To be more specific the IKE, does the following tasks: negotiates IPsec configuration parameters, authenticates secure key exchange, and provides mutual peer authentication and identity protection [30]. IKE can use the Diffie-Hellman algorithm in order to exchange the cryptographic keys between the devices or alternatively the keys can be entered manually [30]. There are two versions of IKE; IKE-1 and IKE-2, of which the IKE-2 is the preferred one, due to improvements compared to the previous version.

Another protocol used by the IPsec is *Authentication Header* (AH). The AH protocol is used in order to authenticate the data source and integrity of IP packets. The AH is however not designed to provide the encryption of the packet payload.

It works as a header that is added to the existing IP header and is used to check the integrity of the whole IP packet [29]. In standard Transport mode, the AH adds only its header under the IP header and authenticates the whole packet which is done by verifying the value of the calculated hash ICV on the receiver side [28]. In Tunnel mode, the AH header is added on top of the existing IP header, and on top of that the additional IP header is added. This allows packets to be routed as needed, however without any encryption [28]. In case the AH protocol would be used together with ESP, the AH would be the one to encapsulate first. Note that usage of AH is optional, and is not required for correct functioning of IPsec.

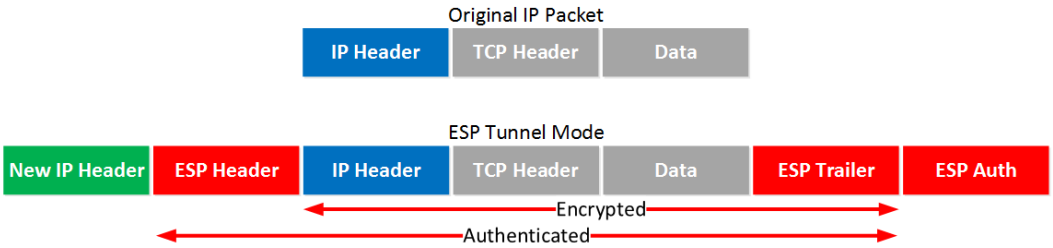


Fig. 2.1: ESP encapsulation in tunnel mode [28]

The last of the protocols used in the IPsec is the *Encapsulating Security Payload* (ESP), which is the most extensive one. The ESP protocol serves for encryption of data while at the same time, it can provide IP header authentication depending on which mode is used [29]. The ESP can be used together with AH, where the AH would be used as an additional layer of authentication or alternatively, if header authentication is disabled in ESP. The ESP can operate, similarly as AH, in two separate modes, the Transport mode and the Tunnel mode. The Transport mode does not authenticate or encrypt the IP header, which makes them vulnerable to any potential attacks [31]. Only the payload of the IP packet would be encrypted and authenticated. However, the benefit of using the ESP in Transport mode is potentially reduced overhead, compared to the Tunnel mode. The Tunnel mode in ESP, both encrypts and authenticates the original IP header. As can be seen in figure 2.1, the ESP adds its own header on top of the existing IP header, which is then encrypted together with the payload. Also, the ESP packet trailer is appended to the encrypted payload. The ESP then adds an additional IP header on top of the ESP header and an optional authentication data field to the end of the packet, which contains ICV [31].

In case AH and ESP were to be used together, the situation would become slightly more complex, depending on which modes are used in both of these pro-

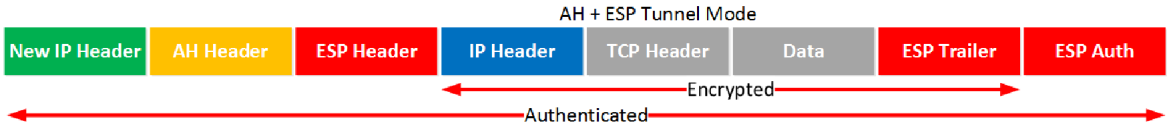


Fig. 2.2: ESP and AH encapsulation [28]

tocols. Combined encapsulation can be seen in figure 2.2. This way the complete authentication of all data and headers can be achieved. However, considering all the added data fields, the total added packet overhead is quite significant.

2.2 Security protocols for PTP over Ethernet

When the PTP uses Ethernet encapsulation as a communication medium, the standard security protocols that would be considered in other cases cannot be used. Instead, only the protocols that can ensure secure communication for Ethernet traffic can be applied, or alternatively, if the devices do not support secure Ethernet implementation, the PTP over Ethernet as a whole would be encapsulated in a different protocol. Encryption of PTP over Ethernet may be more difficult to implement, compared to previous PTP over UDP, considering that hardware support may be needed and the security protocols working on the Data Link Layer may not be as common as protocols that work on Network or Transport Layer. Nevertheless, the Data Link Layer encryption can be the best one in terms of efficiency, considering that the encryption is done closer to the physical level.

2.2.1 MACsec overview

The *Medium access control security* (MACsec) or IEEE 802.1AE is a network security protocol that works by encrypting the payload which would be in a regular Ethernet header or the whole Ethernet frame. The encryption algorithm used by Macsec is AES-GCM, with a 128-bit key length. Unlike with VPN, the encrypted communication can happen only between two devices or within the local area network, where the Ethernet is used as a Data Link Layer protocol. The MACsec mostly contains the same fields as an Ethernet header, which can be seen in the figure 1.5, with addition to the MACsec Security Tag field and the *Integrity Check Value* (ICV) [32] as can be seen on the figure 2.3. MACsec can either function on a hop-by-hop decryption and encryption basis [24] or alternatively, it can be done solely on the end-devices.

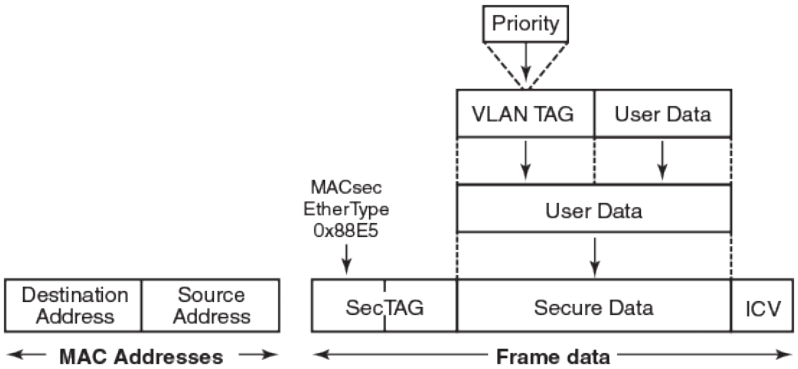


Fig. 2.3: MACsec frame with encrypted VLAN frame [33]

In the scenario that none of the switches in the network topology support MACsec, the encrypted frames wouldn't be able to be encrypted or decrypted. The switches, however, would be able to forward the frames to the hosts based on the source and destination MAC address [34]. This would, however, have the same effect on the Transparent clocks as was already mentioned in the section 2.2, that being the fact that the Transparent clocks would not be able to interact with the PTP message, thus potentially lessening the precision and accuracy. It also needs to be considered that some switches may not behave as expected if they do not recognize the MACsec Security Tag field.

Looking at the different scenarios, where the switches in the network do support MACsec, the hop-by-hop mechanism would be used. This means that all the frames upon arrival at the switch port would be decrypted. In case there is a running Transparent clock instance on the switch, it would be able to interact with the PTP message ordinarily. Therefore, after replicating the message and adding the value to the Correction field in the PTP message, the switch would encrypt it and send it further down the network based on its destination MAC address. It should also be noted that MACsec can be optionally enabled only on some switch ports. Meaning that, if the MACsec is not enabled on the specific port, that port will process the frames as a regular switch [34].

2.2.2 Ethernet over IP

This can be considered rather an unconventional method, considering that communication is done by conveying the unencrypted Ethernet frame, which would normally be communicating on the Data Link Layer, using *Ethernet over IP* (EoIP). To provide the needed level of security, EoIP would need to be used together with

some security protocol described in section 2.1, which would additionally encapsulate the Ethernet frame [35]. This solution would not be optimal for standard use case scenarios, considering the massive overhead, which would be part of the whole encapsulation and decapsulation process. However, if part of the network is being synchronized using the PTP over Ethernet, this method could be used to communicate the Ethernet frame with the PTP packet to different parts of a network, while keeping the necessary security requirements.

3 Implementation

This chapter shall concern itself with the description of the technical implementation of the concepts described in the previous chapters. Note that most of the configuration will be done using Linux *Command-line interface* (CLI), which is often referred to as a terminal. Many commands are executed with superuser privileges which is denoted as *root*, and the device name is after @ sign.

3.1 Embedded Linux for ARM

The devices used for the implementation of PTP in this thesis, run on ARM Cortex processors, produced by NXP semiconductors, designed to be used in industrial settings. Due to the proprietary character of the real-world application, which is the Powerdynax company in charge of, further specifications of the used devices cannot be revealed. The operating system used on these devices is custom Linux for embedded systems. This means that the Linux instance is made for the specific ARM device with a specific use case. This is done with the help of the Yocto project which offers a set of various tools that are used to create customized images. All needed software packages and binaries are built together with the Linux itself. The building or "baking" process is done using BitBake, which consists of layers and allows for the specification of exact tasks to be done. The manufacturers of the specific boards often provide *Board Support Package* (BSP) that can be applied when creating a Linux image for recommended device [36]. Additionally to that different Yocto layers may be provided by various developers, that describe additional hardware-specific packages [37]. The layers can also be added by the user, that specify packages that have not been provided by the manufacturers of the board. The figure 3.1 shows the configuration and build process of the Poky, which serves as a reference system build, using Yocto tools [39].

The build directory is first created by executing a command as seen in 3.1 in a Linux terminal.

Listing 3.1: Create build directory

```
me@my_pc:~/my_project$ source_dir oe-init-build-env build_dir
```

The BitBake executed from the created build directory looks into the existing poky layers and tries to create a Linux image based on them. The user may specifically define, which layers are to be included in the build process and which are to be excluded, in layer configuration files. Again, due to the proprietary character of

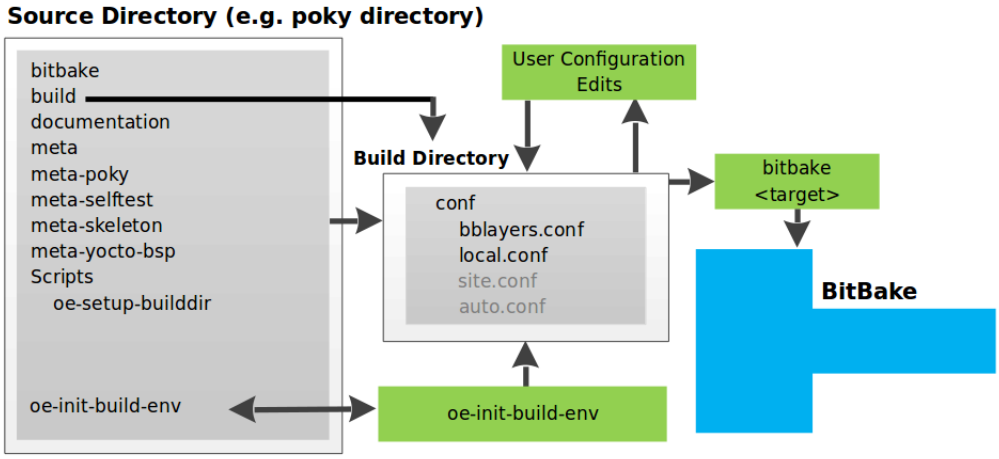


Fig. 3.1: Poky Yocto layer [38]

the used devices, more information in regard to specific Yocto layers and configurations, used for this device, cannot be provided. However, for the purpose of the thesis, only the description of the installation of the required tools is needed. For this the *IMAGE_INSTALL_append* variable in a *local.conf* file would be used. The *local.conf* file specifies packages that are to be installed with Linux image, together with the additional behavior of the operating system [38]. As can seen in the 3.2, some of the packages needed for the purpose of the thesis were specified to be built.

Listing 3.2: Install packages

```
IMAGE_INSTALL_append = " \
...
  ethtool \
  tshark \
  linuxptp \
  wireguard-tools \
  iproute2 \
  strongswan \
... "
```

The *bblayers.conf* specifies which layers, that include the packages specified in the *layers.conf*, should be read by BitBake. Once the configuration is done, the Linux image can be created using *bitbake* utility. *Bitbake* allows for the specification of different types of images, which may have different properties, however for the current use-case, the *core-image-full-cmdline* suffices [40]. The building process is started by executing 3.3. Note that the process of Linux image building may

take several hours, depending on the available resources of the device that runs the compilation.

Listing 3.3: Bitbake image

```
me@my_pc:~/my_project/build_dir$ bitbake core-image-full-
   cmdline
```

Once the customized Linux image is created, it can be found in *tmp/deploy/images* directory, located in the previously created *build_dir* [41]. The Linux image must be then loaded to ARM devices in order to be used. This can be done using a USB flash disk, or memory card, to which the image can be loaded, and from which the ARM device can also boot the operating system. Loading of the image into the removable device is done by *dd* Linux utility as seen 3.4, where the *if* specifies the source, and *of* specifies the destination, which is the removable disk device [42]. Note that the resulting created image may be compressed by BitBake, it will therefore need to be extracted based on the used compression format. Image can also be flashed from the removable disk device directly into the internal memory of the ARM device using the same principle as seen in the 3.4, where the destination would be the internal disk, which overwrites all existing data. Note that embedded devices often use their own preinstalled bootloader. If the new system was to be run from the memory card, it must be specified in the bootloader environment, what partition it should boot into. The bootloader used in the current instance is U-boot, which can be accessed directly at the beginning of the boot process. Once the U-boot environment is accessed, the variable which specifies the boot partition must be changed so that the destination partition is that of a removable device. The U-boot environment is then saved to a separate memory partition, located apart from that of a regular filesystem memory.

Listing 3.4: Flashing image

```
me@my_pc:~/my/path$ dd if=core-image-full-cmdline.my_format
   of=/dev/sda
```

3.1.1 Additional configuration

It must be noted that in the current case, some tools cannot be compiled using Yocto, due to kernel compatibility issues. There is therefore a need to cross-compile these tools for ARM architecture on a different computer and then move them to the destination ARM device. This is the case with Wireguard, where the latest implementation does not support the kernel version used on the current devices. For this

reason, the alternative Wireguard implementation written in the Go programming language must be used. The compilation of the Wireguard-go will be done as seen in 3.5, and the resulting binary file will be moved to the destination device. Note that the *wireguard-tools* package specified in the Yocto *local.conf* file contain only the tools used for the configuration of Wireguard, not its implementation.

Listing 3.5: Cross-compile wireguard-go

```
me@my_pc:~/wireguard-go$ GOOS=linux GOARCH=arm CGO_ENABLED=0
   go build -v -o wireguard-go
```

In the case of Macsec, its compilation must be enabled directly in the kernel configuration file, which is usually specified by having *defconfig* in its name. If done through Yocto there needs to be created a patch, which adds the CONFIG_MACSEC=y item to the specified configuration file to build a Macsec as a static driver. Alternatively, the Macsec can be built as a module, by specifying the "m" option. The building process continues as specified before, with the difference that the whole kernel may be rebuilt again.

The Strongswan, which is to be used as an implementation of IPsec, for its proper function requires additional configuration of the kernel configuration file *defconfig* as well. In regular Linux distributions the selected kernel configuration options may be enabled by default, however considering the character of the ARM device, in this case, it has to be done manually. All requirements can be seen in the electronic appendix.

What should be also noted is the fact that the versions provided by the existing layers may not have all the features needed for the correct configuration of the system. In this case, it is the *ptplinux*, *strongswan* and *iproute2* for which additional *.bb* recipes must be made in the custom layer, which specifies the newer version that is provided by the existing layers. The relevant part of Yocto layer structure used in this build can be seen in Appendix listings, where layers that describe *iproute2*, *linuxptp* and *strongswan*, were taken from the latest existing OpenEmbeded framework and modified so that they would be compatible with current Yocto version [46],[47],[48].

3.1.2 Networking

Once all to-be-used ARM devices are up and running, the networking needs to be set up. This can be done in a standard way by using *ip address add* command. For testing purposes, the IPv4 network range to be used in the current implementation on physical interfaces will be *TEST-NET-1*: *192.0.2.0/24*, as recommended by RFC

5737 [43]. Therefore, setting an address on the Grandmaster clock device network interface *eth1* is done by executing 3.6. The IPv4 address on all other network interfaces used in the local network is done in the same manner, just with different addresses. If networking is not set, the device would have to be accessed by means of a serial interface or multimedia interface.

Listing 3.6: Set IPv4 address

```
root@ptp_deviceA:/# ip a a 192.0.2.1/24 dev eth1
```

Another aspect that needs to be set for the Grandmaster clock device is synchronization with the external time source. Normally the PTP Grandmaster would be synchronized with GPS or directly with the atomic clock. Considering the unavailability of these means, the NTP protocol will be used instead. In order to synchronize the Grandmaster with the external network source using NTP, the *chrony* and *chronyc* packages are required, which can be installed the same way as seen in the 3.2. The external NTP server, which must provide time for the Grandmaster clock will be specified by adding specific information to the *chrony.conf* file located in *etc* directory. In the added line the *pool* specifies that a pool of servers can be used. The *ntp.nic.cz* specifies a public NTP server used in the region [45], and the *iburst* specifies the initial burst of requests to speed up NTP synchronization [44].

Upon modifying the *chroy.conf* file, *chronyd* service must be restarted in order for specified settings to take effect. This can be done by *systemctl restart* followed by the name of the *systemd* service. To verify that the device is being synchronized with the external NTP server the *chronyc* command can be issued as seen in 3.7. The meaning of the NTP parameters shall not be further explained, as this is not the primary focus of the thesis, however by seeing the specified *ntp.nic.cz* in the list of active servers, it can be confirmed that the NTP connection is active.

Listing 3.7: NTP connection

```
root@ptp_deviceA:/# systemctl restart chronyd
root@ptp_deviceA:/# chronyc sources
MS Name/IP address         Stratum Poll Reach LastRx Last sample
===============================================================
^* ntp.nic.cz                1   6   377    50   -128us[ -142us]
   +/- 2688us
```

It should also be noted that the used devices have two network interfaces, of which the *eth0* on the Grandmaster clock device is used for communication with the NTP server and SSH connections, and *eth1* which is to be used for the purposes of PTP and encryption. The user should also make sure that there are no layers

that would create unneeded *systemd* services as they would add additional load to the system unnecessarily.

3.2 PTP for Linux

The PTP support for the Linux kernel was implemented by Richard Cochran in 2010 [49]. The PTP infrastructure for Linux works together with SO_TIMESTAMPING Linux socket option which allows for hardware timestamping of the arrived packets [50]. The PTP for Linux works on the basis of a clock driver, which is kernel-based, and a class driver which is userspace-based [49].

3.2.1 System properties

The previously mentioned SO_TIMESTAMPING is a Linux *socket* interface option based in the Linux user space. The *socket* is used as a C programming language function based on its use case. The *socket* function may provide many functionalities based on which option is set by *setsockopt*, which is another function [51]. The SO_TIMESTAMING socket option provides support for timestamp generation from multiple sources [50]. It should not be confused with the SO_TIMESTAMP, which generates timestamps only for incoming messages. The SO_TIMESTAMING generates timestamps for both the reception and transmission of messages over a network. The source of timestamps provided by SO_TIMESTAMPING is determined by what bit is set in *setsockopt* function [50]. In the case of PTP, it would be optimal that timestamps are generated directly by the network adapter, considering that it is supported by the device. This way the timestamps, in previous chapters described as *t1*, *t2*, *t3*, *t4* can be created.

The PTP infrastructure for Linux defines basic clock operations as Setting the time, Getting the time, Shifting the clock by a given offset, and Adjustment of clock frequency [49]. In the white paper provided by Renesas semiconductor manufacturer, the PTP implementation of the Slave clock consists of four main components [52] which can be seen in the figure 3.2, and are described as follows:

1. The user space drivers, which implement the protocol according to IEEE 1588, and the clock servo algorithm used for readjusting the internal clock [52].
2. Linux kernel and its drivers, interacting directly with the hardware.
3. The unit which is capable of providing hardware timestamps, integrated with the network interface. In figure 3.2 denoted as TSU.
4. The hardware clock that generates the time and provides it to the unit, which is controlled by the servo algorithm. In figure 3.2 denoted as PHC.

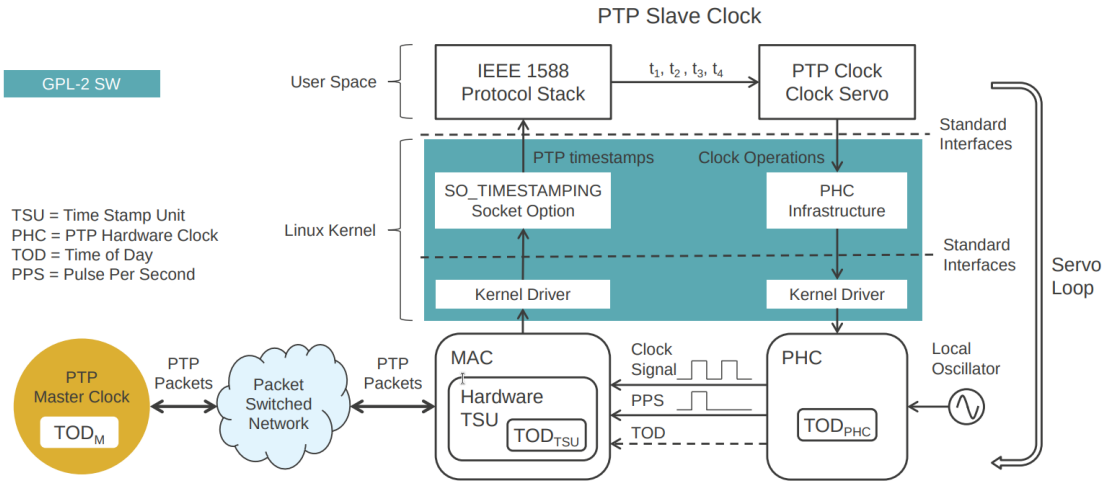


Fig. 3.2: Linux PTP scheme with GPL-2 Linux kernel license [52]

The clock servo is an algorithm that allows for the clock to be readjusted based on the calculated time offset while avoiding the re-setting of an existing value of the clock [53]. The example reason why the correct time value cannot be simply set as a replacement for the current time is that if the Slave clock time was too much ahead compared to the Master clock, the time-back could happen [53]. A similar problem would occur if the Slave clock was too behind compared to the Master clock. Inconsistencies and disturbances among other system processes would very likely start occurring. This is why the servo algorithm was introduced into PTP to gradually correct the timing error over a certain period of time.

It should also be noted that the default timescale used by the PTP hardware clock is *International Atomic Time* (TAI), which is different from the typical system timescale which is in UTC [54]. The TAI at the time of writing is currently diverged by 37 seconds from the UTC, meaning that UTC is 37 seconds behind the TAI [55]. Due to the fact that the Grandmaster clock may not have the means to obtain precise TAI, the hardware clock may allow the usage of an arbitrary timescale. This behavior can be however configured directly from the user space driver.

As has been mentioned before, the official Linux specification defines clock drivers, which are kernel-based, and those that are userspace-based. Note that user space in Linux is an environment for the execution of application software, while the kernel deals with system calls and is able to interact directly with the hardware. The clock kernel driver is registered to a class driver, meaning that all the hardware clocks can be managed from the user space. Implementation of the clock drivers may differ, considering the properties of the hardware.

The user space implementation of PTP on Linux is done by The Linux PTP Project software. This project provides a set of tools and daemons that manage specific clock operations [54]. Note that a daemon is software designed to run as a background process. The *ptp4l* is a daemon that synchronizes the PTP hardware clock from the network interface [54]. The *phc2sys* is a daemon that synchronizes the PTP hardware clock and the system clock [54]. The *pmc* is the tool used to configure *ptp4l* in run-time.

3.2.2 PTP driver tools

As mentioned previously, the PTP timestamps its messages directly at a network interface. It is therefore important to determine which timestamping options are supported by the network interface in question. This can be determined by using the *ethtool*, which is a Linux utility used to "query or control network driver and hardware settings" [56]. Using the *ethtool* command in the Linux terminal, the required properties of the network interface in use can be determined. Upon inspection of all available options with command *ethtool –help*, it can be determined that the correct option for showing the timestamping possibilities of a given network interface is *–show-time-stamping*, which can be alternatively used as a *-T* [56]. Therefore after issuing the command *ethtool -T* into the Linux terminal, with a specific network interface as an argument, the output as seen in 3.8 can be observed.

Listing 3.8: Network interface timestamping information

```
root@ptp_deviceA:/# ethtool -T eth1
Time stamping parameters for eth1:
Capabilities:
        hardware-transmit     (SOF_TIMESTAMPING_TX_HARDWARE)
        software-transmit     (SOF_TIMESTAMPING_TX_SOFTWARE)
        hardware-receive      (SOF_TIMESTAMPING_RX_HARDWARE)
        software-receive      (SOF_TIMESTAMPING_RX_SOFTWARE)
        software-system-clock (SOF_TIMESTAMPING_SOFTWARE)
        hardware-raw-clock    (SOF_TIMESTAMPING_RAW_HARDWARE)
PTP Hardware Clock: 0
Hardware Transmit Timestamp Modes:
        off                   (HWTSTAMP_TX_OFF)
        on                    (HWTSTAMP_TX_ON)
Hardware Receive Filter Modes:
        none                  (HWTSTAMP_FILTER_NONE)
        all                   (HWTSTAMP_FILTER_ALL)
```

Upon the inspection of available capabilities, it can be determined which timestamping sources are available for the network interface *eth1*. For the hardware timestamping the following flags are the most important, for which the definitions are taken directly from the Linux kernel documentation [50].

- SOF_TIMESTAMPING_TX_HARDWARE - request transmission timestamp created by the network interface
- SOF_TIMESTAMPING_RX_HARDWARE - request reception timestamp created by the network interface
- SOF_TIMESTAMPING_RAW_HARDWARE - report hardware timestamps when available

For the software timestamping the following flags are important [50].

- SOF_TIMESTAMPING_TX_SOFTWARE - request transmission timestamp created by the driver; timestamp is created when data leaves the kernel
- SOF_TIMESTAMPING_TX_SOFTWARE - request reception timestamp created by the driver; timestamp is created when data enters the kernel
- SOF_TIMESTAMPING_SOFTWARE - report any software timestamps when available

The PTP Hardware Clock parameter describes which hardware clock is currently in use. As seen in 3.8, the selected clock is the one denominated with index 0. The HWTSTAMP_TX_OFF option disables the hardware timestamping for to-be-transmitted packets [50]. The HWTSTAMP_TX_ON enables the hardware timestamping for to-be-transmitted packets [50]. The Filter modes would be used to indicate which incoming messages are to be timestamped. However in this case there are only HWTSTAMP_FILTER_NONE and HWTSTAMP_FILTER_ALL options that determine if either no timestamp for any received packet should be provided or all receiving packets should be timestamped.

3.2.2.1 ptp4l

After it has been established that PTP timestamping is supported on a network interface, it can be proceeded further with the usage of PTP tools. The *ptp4l* utility can be either used explicitly from the terminal, or it can be configured as a *systemd* service, running in the background with the start of the system. In order to determine which configuration settings are to be used, the *ptp4l -h* command can be issued in the system terminal or the documentation [57] can be consulted. By issuing the command *ptp4l -m -i eth1* in the Linux terminal, the *ptp4l* daemon is started, where *-m* specifies that all messages should be written into *standard output* (stdout) and *-i* is used to specify the network interface [57]. Note that no other attributes

were specified, therefore other options shall be interpreted as defaults, meaning that PTP messages are to be encapsulated inside UDP and IPv4 protocols, the delay mode is E2E and the timestamping is done by hardware if available.

Listing 3.9: Ptp4l initializatoin

```
root@ptp_deviceA:/# ptp4l -m -i eth1
ptp4l[36.136]: selected /dev/ptp1 as PTP clock
ptp4l[36.141]: port 1: INITIALIZING to LISTENING on
   INIT_COMPLETE
ptp4l[36.142]: port 0: INITIALIZING to LISTENING on
   INIT_COMPLETE
```

As observed in 3.9, the *ptp4l* automatically selected */dev/ptp1* clock for hardware timestamping, which belongs to selected *eth1* interface. The following messages indicate that the port had been put into a listening state, meaning that the port is waiting for the Master clock to send an Announce message. If an Announce message was to be received, the BMCA would be performed.

Listing 3.10: Ptp4l listening

```
ptp4l[43.682]: port 1: LISTENING to MASTER on
   ANNOUNCE_RECEIPT_TIMEOUT_EXPIRES
ptp4l[43.682]: selected local clock 91b711.fffe.ece40d as
   best master
ptp4l[43.682]: assuming the grand master role
```

As can be seen in 3.10, in this case, the device did not receive any Announce message after a period of time, meaning that standard BMCA is not performed. Instead, *ptp4l* selects the local clock as the most suitable provider of time and assumes the role of the Grandmaster clock on a selected network interface. All the devices that are in the same network will receive messages, which are sent from the address of the configured device to the multicast group, which are all interfaces with initialized *ptp4l* part of.

The brief output from packet-capture utility *tshark*, which can be seen in 3.11, shows all the packets captured on the network interface *eth1*, which is again specified by *-i* parameter. By using option *-Y ptp*, only the packets related to PTP are displayed. The source IPv4 address *192.0.2.1* is that of a Master clock device and the destination is a multicast address *224.0.1.129* used by PTP devices. The number after the protocol name describes the size of the message in bytes. The *ptp4l* in multicast mode sets the behavior of the Master clock so that it does not wait for

the Slave clocks to request advertisement, but simply periodically sends Announce messages together with Sync and Follow_Up that are needed for synchronization.

Listing 3.11: Tshark capture PTPv2 Announce

```
root@ptp_deviceA:/# tshark -i eth1 -Y ptp
Capturing on 'eth1'
1 0.000000000     192.0.2.1 ? 224.0.1.129 PTPv2 106 Announce
   Message
2 0.999038000     192.0.2.1 ? 224.0.1.129 PTPv2 86 Sync
   Message
3 1.000302500     192.0.2.1 ? 224.0.1.129 PTPv2 86 Follow_Up
   Message
```

The initial setup of PTP for a specific network interface using *ptp4l* is rather simple assuming that all predispositions are fulfilled. Let us consider the scenario where to the existing device called *ptp_deviceA*, another device called *ptp_deviceB* is connected directly via the Ethernet cable. The IP address of the *ptp_deviceB* is set so that it is in the same subnetwork as *ptp_deviceA*. Therefore the address of *ptp_deviceB* would be *192.0.2.2/24*.

Once the configuration is done the *ptp4l* command is executed the same way as in 3.10, with the addition of *-s* flag, which indicates that the device will only act as a Slave clock. Note that with this flag added, the device may still indicate that the local clock was selected as a most suitable provider, however, it may not act as a Master clock on any network interface, instead, it will only listen for Announce message on a specified network interface.

As can be seen in 3.12, after *ptp4l* has been executed, the initialization is the same as in 3.9. What can be noticed at first is that the *ptp_deviceB* does not select its own clock as the best Master clock but instead determines that the most suitable Master clock is *91b711.fffe.ece40d*, which belongs to *ptp_deviceA*. Upon determining the Master clock the *ptp_deviceB* changes the port state from LISTENING to UNCALIBRATED, meaning that the Slave clock is already communicating with the Master clock device, however, it is still not synchronizing. The PTP port in this state is initializing the servo algorithm and preparing to start creating clock adjustments [4]. Once the state changes from UNCALIBRATED to SLAVE the clock adjustments are started to be performed based on the PTP port from which it is being synchronized [4]. The state of the servo itself is indicated by *s0*, *s1*, and *s2* flags [58]. The *s0* indicates an unlocked state, meaning that changes to be done to the clock are still being calculated, but not applied. The *s1* indicates the clock step state, meaning that the clock can be changed without limitation. The final state

s2 indicates the locked state, meaning that the clock cannot be changed drastically but is slowly adjusted to match the Master clock. The process of synchronization and change of states may take significantly longer with the software timestamping, considering that system resources are also used by other applications, which require their own CPU time.

Listing 3.12: Synchronization between two nodes

```
root@ptp_deviceB:/# ptp4l -m -s -i eth1
...
ptp4l[154.812]: port 1: new foreign master 91b711.fffe.ece40d
   -1
ptp4l[158.813]: selected best master clock 91b711.fffe.ece40d
ptp4l[158.813]: port 1: LISTENING to UNCALIBRATED on RS_SLAVE
ptp4l[161.814]: master offset -209846025 s0 freq      +0 path
    delay      -155
ptp4l[163.815]: port 1: UNCALIBRATED to SLAVE on
   MASTER_CLOCK_SELECTED
ptp4l[164.815]: master offset      -1076 s2 freq  +12550 path
    delay      -155
```

The *master offset* value indicates the offset of the Slave clock from the Master clock, as was described in the theoretical chapter 1.2.2.2. The offset values shown in the output are in nanoseconds. As can be noticed in the provided output, the initial offset between the Slave and Master is enormous. This is due to the fact that the device *ptp_deviceB* was not, prior to establishing a PTP connection, connected to any external time source, thus starting at zero, from the UNIX time epoch perspective. Once the servo algorithm in state *s1* jumps the clock to a time closer to the Master clock, the following offset is much smaller than the initial one and the clock is being slowly readjusted. The *freq* value describes the servo frequency adjustment of the clock in *Parts Per Billion* (ppb) unit [58]. The *path delay* represents the calculated delay between Master and Slave as described in the theoretical chapter 1.2.2.2.

If *ptp4l* instance on both *ptp_deviceA* and *ptp_deviceB* would be used with *-S* option, the results would likely be significantly worse, due to the overhead caused by software timestamping.

The configuration of *ptp4l* can be done through a file, using *-f* parameter, or *pmc*, which provide much more in-depth options than command line flags. The common practice is to set *ptp4l* as a *systemd service*, which operates *ptp4l* in the background with all the options specified in *.conf* file.

3.2.2.2 pmc

With the *pmc* management client, more detailed information can be provided regarding the PTP synchronization. It can also be used to set additional options for the existing PTP instance. Therefore, in order to use *pmc*, the *ptp4l* must be active. The *pmc* is executed the same way as the *ptp4l*. The *pmc* reads *standard input* (stdin), meaning that specific actions need to be queried from *pmc* in order to obtain needed data [59]. The action used to obtain data is GET and the action used to set data is SET. These actions are followed by specific MANAGEMENT_IDs that describe the information that is to be queried [59]. In order to communicate with the existing *ptp4l* process, the *-u* flag needs to be specified, which specifies Unix Domain Socket as a communication method [59]. Unix Domain Socket is used for communication between processes running on the same system, and each process has assigned its own endpoint where as the default is specified that of a *ptp4l* process [59]. Another option used with *pmc* is *-b*, which specifies the number of boundary hops [59]. Therefore in order to display information about the local clock, which acts as a Slave clock, and the adjacent clock, with which it communicates, the *-b 1* would be used. In order to display only the local clock, the *-b 0* option is used. The output when using *pmc* with CURRENT_DATA_SET id can be observed in 3.13.

Listing 3.13: Pmc with 'CURRENT_DATA_SET'

```
root@ptp_deviceB:/# pmc -u -b 1 'GET CURRENT_DATA_SET'
sending: GET CURRENT_DATA_SET
        57c039.fffe.f97b52-0 seq 0 RESPONSE MANAGEMENT
           CURRENT_DATA_SET
                 stepsRemoved     1
                 offsetFromMaster 3.0
                 meanPathDelay    453.0
```

The *stepsRemoved* variable describes the number of nodes passed in the communication path [58]. The *offsetFromMaster* describes the last measured offset from the Master clock and the *meanPathDelay* describes the mean value of path delay between the Slave and Master clock [58].

To get more precise information about the clock itself, the TIME_STATUS_NP would be used, of which the parameters are described as follows. The *master_offset* variable has the same meaning as above described *offsetFromMaster*. The *ingress_time* is a UNIX timestamp in nanoseconds of when was the last PTP message received [61]. The other important variable is *lastGmPhaseChange* which describes when the phase of the Grandmaster clock changed. The *gmPresent* variable is *true* if the clock

is synchronized to the Grandmaster clock. If the clock is Grandmaster itself then the value is *false* [58]. The *gmIdentity* specifies the identity of the Grandmaster clock. In order to change specific variables using *pmc*, action would be performed as seen in 3.14. The BMCA-related *priority1* value was changed from default 128 to 1 on the Master clock device, meaning that all the devices running BMCA will select this clock as the first option. All the other parameters, described in 1.1 can be configured the same way.

Listing 3.14: Pmc with 'CURRENT_DATA_SET'

```
root@ptp_deviceA:/# pmc -u -b 0 'SET PRIORITY1 1'
sending: SET PRIORITY1
        91b711.fffe.ece40d-0 seq 0 RESPONSE MANAGEMENT
           PRIORITY1
                priority1 1
```

As was previously mentioned the PTP hardware clock by default uses TAI timescale which is diverged by 37 seconds from the UTC timescale used by the system clock [55]. To obtain the correct system time on Slave clock devices, this offset must be accounted for, when using *phc2sys* the timescale can be set to arbitrary by setting *ptpTimescale* and *currentUtcOffset* to zero using *pmc* with GRANDMASTER_SETTINGS_NP MANAGEMENT_ID. Note that when the intention is to change even a single variable, the *pmc* will require that all other variables are also specified in the same SET action.

3.2.2.3 phc2sys

The *phc2sys* is used to synchronize two different clocks on the same device, usually the system clock to the PTP hardware clock used on the network interface. The *phc2sys* is required because *ptp4l* does not automatically synchronize the PTP hardware clock to the system clock that is used by the system. Note that if *ptp4l* uses software timestamping, the *phc2sys* does not need to be in place considering that the PTP software timestamps are already provided by the system clock. When synchronizing hardware and system clock, from the time synchronization perspective, the system clock becomes the Slave clock and the PTP hardware clock becomes the Master clock, where all the communication happens only within the device system. Assuming that the *ptp4l* is already running, the *phc2sys* can be executed as seen in 3.15. The *-s* flag specifies the clock that will take the role of the Master clock in the operating system, which is in this case the PTP hardware clock on the specified network interface [62]. The *-c* flags specify the clock to act as a Slave clock

within the system [62]. In this case, this would be CLOCK_REALTIME, which is the system clock. Note that the roles of *eth1* and CLOCK_REALTIME could be arbitrarily exchanged as *phc2sys* does not limit which device clocks synchronize with each other. The *-S 1* parameter specifies that servo can be returned to *s1* mode if needed, considering that if some sudden time changes were to occur, due to misconfiguration for example, it would take too long for servo to readjust the clock while in *s2* mode. The *-w* specifies that *phc2sys* should wait until *ptp4l* is in a synchronized state, and then synchronizes the system clock according to *currentUtcOffset* which is specified in the Grandmaster clock settings, which can be displayed by *pmc*. Alternatively, the UTC offset can be provided manually with *-O* option and a specific number to adjust for the divergence. In the case of TAI the value -37 would be used.

Listing 3.15: Synchronization of system clock with hardware clock using phc2sys

```
root@ptp_deviceB:/# phc2sys -m -s eth1 -c CLOCK_REALTIME -S 1
    -O -37
phc2sys[4118.673]: CLOCK_REALTIME phc offset 37007191239 s0
   freq      +0 delay   3875
```

The Grandmaster clock with hardware timestamping must also have its time source. Considering that different service already provides timing to *ptp_devicaA*, the system time can be used as a time source for the PTP hardware clock on the Master. This can be either done automatically, by specifying *-a* autoconfiguraiton command with *-r* option used twice to specify system clock as a time source [62], or manually by simply setting the network interface with hardware clock as a Slave and the CLOCK_REALTIME as a Master.

Notice that the delay between two clocks on the same device can be larger than between two directly connected clocks on the network. This is due to the fact that the system clock is software-based, therefore *phc2sys* has to fight for the CPU time with other processes.

3.3 Network security on Linux

There are many different tools that can be used to cryptographically protect the network traffic in the Linux operating system. These tools often differ in the complexity of implementation, as well as the difference in the setup process.

3.3.1 Wireguard-go encryption

The original implementation of the Wireguard was written to be run from the kernel of the operating system. However, as was mentioned in the 3.1.2, the standard Wireguard cannot be compiled using Yocto, due to kernel compatibility issues. Because of this, an alternative written in Go must be used. The most crucial difference compared to the standard Wireguard version is that Wireguard-go runs from the user space. Generally speaking, the speed of processes executed in the kernel or user space is the same. The main culprit of slow-down for the userspace applications comes when the system calls need to be made. For each system call the user space applications need to switch to supervisor mode, which takes additional resources and CPU time [63]. Nevertheless, the configuration process of the Wireguard remains the same. The first step that needs to be done is the creation of a virtual interface, which will be used for routing the encrypted network traffic. This can be done by simply executing the *wireguard-go* binary file, with the specified interface name as an argument.

Once the virtual interface is created an IPv4 address that is to be used as a point for encrypted communication, must be assigned. The IPv4 address will be assigned the same way as described in the 3.1.2, with the selected IPv4 range being *TEST-NET-2*: *198.51.100.0/24* as recommended by RFC 5737 [43]. Note that these addresses will serve only for the encrypted communication between the virtual network interfaces. However once traffic comes out of the selected physical interface, the source IP address becomes that of the physical network interface. In order to communicate with other peers, every virtual interface needs to have assigned a public and private key. These can be generated using *wg* tool, which was compiled in the previously mentioned *wireguard-tools* package. Using *wg genkey* command allows for the generation of a private key and a public key, which is derived from the private key. Once these keys are created they can be assigned to the virtual interface by using *wg set* command [26] as seen in the 3.16. When the keys are assigned to the interface, it can be rendered active using *ip link* command.

Listing 3.16: Key creation

```
root@ptp_deviceA:~# wg genkey > private_keyA
root@ptp_deviceA:~# wg pubkey < private_keyA > public_keyA
root@ptp_deviceA:~# wg set wg0 private-key ./private_keyA
```

After each device is configured, the peers with which encrypted communication is to be done must be explicitly specified for the virtual interface. This can be done again using *wg* tool. Every Wireguard interface is identified by its public key,

therefore this public key must be also known by the opposite interfaces which are to communicate with each other. Additionally, both IP addresses of the virtual and physical interfaces must be specified, including the network port number of the peer Wireguard instance. This is done by executing *wg set wg0* with the addition of *peer* keyword followed by the public key of a peer device. In addition to that the IP address of a virtual interface of a peer device is set by *allowed-ips* keyword, and an address of an endpoint, which is in this case address and port number of a physical interface of a peer device, is specified by *endpoint* keyword. Once everything is set, the configuration of all virtual interfaces can be verified by using the *wg show* command. Note that each virtual interface on the same device can have different configuration settings, meaning that each virtual interface can be set up so that it communicates with different peers.

Additionally, the *tshark* can be used to verify how the traffic is being handled. When issuing *ping* command to the IP address of a virtual interface of a peer, and then observing the traffic coming out of the current device virtual interface, it will be seen that it is unencrypted. However, when looking at the physical interface of the same device, the traffic is already encapsulated and encrypted.

3.3.1.1 Configuration for PTP

For PTP implementation this means that the network interface that is to be specified when using *ptp4l* utility must be the one that is created by *wireguard-go*, if the user wants the PTP traffic to be encrypted. There are however some additional steps that must be performed in order for PTP to function over Wireguard. Multicast communication is not natively supported by Wireguard, for this reason, the configurations to the virtual interface must be made. That is done by setting *multicast* with *ip link set* command on a specified virtual interface as in 3.17. Additionally, the multicast addresses that the Grandmaster clock uses for communication have to be added to *allowed-ips* to the specified peer, on all devices.

Listing 3.17: Multicast configuration

```
root@ptp_deviceA:/# ip link set dev wg0 multicast on
root@ptp_deviceA:/# wg set wg0 peer ‘cat public_keyB‘ allowed
   -ips 198.51.100.1/32,224.0.1.129/32,224.0.0.107/32
```

The crucial issue when configuring a VPN tunnel is that there is no MAC address assigned to the Wireguard virtual interface. That means that PTP cannot generate the clock identification value as it would with the regular network interface and instead it uses a default value *000000.fffe.000000*. This means that the Slave clock

cannot be synchronized to the Master, due to the fact that both have assigned the same default clock identification value. The *ptp4l* is not able to distinguish between different clocks if they do not have a unique identity. That can be fixed by reassigning the *clockIdentity* value for the given PTP instance through the configuration file, for each device in use, where the new values can be completely arbitrary. Once it is ensured that each clock in the encrypted network has its own *clockIdentity* the synchronization can proceed the standard way. The output 3.18 shows both the contents of the configuration file for device *ptp_deviceB* and the start of synchronization on the virtual interface. Note that some of the earlier versions of PTP for Linux do not support the changing of clock identity. Therefore the custom layer needs to be made for the Yocto project, in order to specify the suitable version of *ptp4l* to be built into the Linux image.

Listing 3.18: PTP configuration file and synchronization throught the virutal interface

```
root@ptp_deviceB:~# cat settings.cfg
[global]
# PTP clock identity for ptp_deviceB
clockIdentity            000000.0000.000002
root@ptp_deviceB:~# ptp4l -s -m -i wg0 -S -f settings.cfg
```

Another fact that needs to be considered is that due to the Wireguard network interface being virtual, the hardware timestamping is not supported, which can again be verified by *ethtool -T*, with the selected virtual interface as an argument. As was mentioned before, software timestamping brings a significant increase in path delay, compared to hardware timestamping, which is in this case increased even more due to the encryption. It should be noted that the total increase in the path delay is rather significant. The exact values shall be shown and analyzed in the part of the thesis that concerns itself with the practical measurement.

The creation of the Wireguard virtual interface is done by creating a TUN kernel virtual network device used for reception and transmission of packets from the user space [64]. It creates a virtual network within the device itself where the packets are created in the user space and afterwards routed to the physical interface. TUN operates on the Network Layer, which is also related to a TAP device, which operates on the Data Link Layer.

Nevertheless, upon inspecting the kernel driver for TUN, it can be determined that there is indeed no pointer that would indicate a link to the physical underlying device that would provide hardware timestamps. Due to this fact, there is no feasible

solution for the implementation of hardware timestamping on a TUN type of network interface.

3.3.2 StrongSwan encryption

There are many different implementations of the IPsec protocols, and each has its own specifications. The StrongSwan has been chosen as it has verified support for the ARM devices which are used in this thesis. The StrongSwan project is a descendant of the previous FreeS/WAN project and is maintained by the part of developers who contributed to the original project. The configuration of the StrongSwan is a little bit more extensive than other protocols described in this thesis as it requires keeping track of configuration files which are used on each peer device. The newest version of StrongSwan makes use of *swanctl* utility, which can be used however only when the corresponding systemd service is active. To verify if the *strongswan* service is active, the *systemctl status* command can be used with the name of the service as an argument.

In order to initiate the encrypted connection between two hosts, the previously mentioned configuration file needs to be created in the path used by *swanctl* utility, that is */etc/swanctl/conf.d*. The configuration file uses a JSON-like format, where multiple different connections can be described and each section describes different properties of the described connection. The first section of the configuration file concerns itself with the specification of the IKE protocol parameters, that is *local_addrs* and *remote_addrs*, which is to be used for IKE operations. These addresses are of course those of *ptp_deviceA* and *ptp_deviceB*. The authentication basis is usually performed with the use of certification authorities and public keys, however for the sake of simplicity only PSK shall be used, which is specified by adding *auth* field as *psk* for both *local* and *remote* subsections in the file. The public and private keys could be used as well if needed certificate structure was to be provided, including the certification authority.

The most important part of the IPsec configuration is the subsection *children*, which defines Child Security Associations. The CHILD_SA is any Security Association, which was negotiated by IKE protocol and defines how is input and output traffic treated based on the defined selectors [65]. By default, all traffic is treated dynamically meaning that selectors are determined based on the existing traffic [66]. However, to ensure the most predictable behavior, it is best to define selectors for outgoing and incoming traffic manually. The traffic matching is however supported only for the Network Layer, meaning that only packets with IP headers are matched. The implication for PTP is that due to the fact that IPsec is not able to

match Data Link Layer packets by policy, all PTP messages sent in *L2* mode would be unencrypted.

The CHILD_SA also allows specification of the cryptographic algorithm which is to be used. The algorithm can be defined for ESP and AH protocols. Unfortunately, StrongSwan does not currently support simultaneous usage of ESP for encryption and AH for authentication. All the proposed cryptographic algorithms must be supported by the Linux kernel and the StrongSwan must be built with adequate plugins. The encapsulation mode used by default is *tunnel*, however, it can be changed to *transport* with *mode* parameter. The example child configuration can be seen in the 3.19. Note that all proposed selectors and algorithms must be accepted by all sides before the connection can be initiated. Additionally, the physical network interface may be specified.

Listing 3.19: Swanctl child specification

```
ptp-conn {
...
        ptp-conn-child {
            local_ts=192.0.2.0/24
            remote_ts=192.0.2.0/24
            interface=eth1
            esp_proposals=aes256gcm128
        }
...
```

Another completely separate field called *secrets* has been defined in the same file, which would contain PSK that is to be used for the IKE verification. The key may be generated with the help of */dev/urandom*, which is also mentioned in the Macsec configuration chapter.

Management of the plugins that are to be used with the *swanctl* utility is done in the */etc/strongswan.conf* file, however for the very basic connection no extra plugins other than default ones are needed. Finally, once the configurations are set on all peer devices, *strongswan* service must be restarted with *systemctl restart* command. The Security association is established with *swanctl -i –child* followed by the name of the CHILD_SA, which also establishes the IKE Security association.

3.3.2.1 Configuration for PTP

The attempt at enabling standard PTP for Linux may be more complicated than expected. This is due to the fact that PTP uses multicast for communication, which is

not natively supported by many IPsec implementations, including the StrongSwan. To enable transmission of multicast messages in StrongSwan additional experimental plugin would need to be installed called *forecast* [67], which requires manual additions to the Yocto recipe that configures the StrongSwan build. The *forecast* plugin, then would be specified in the mentioned *strongswan.conf* file with the PTP multicast groups as arguments. The multicast groups then would need to be added to traffic selectors on both devices. Additionally, specifications of netfilter marks would have to be added to the child configuration of the Master clock device [67]. After performing the above-mentioned steps the Master clock device can send the standard multicast messages to all peers. This would, however, work only in tunnel mode as transport mode does not support multicast at all. The major issue however shows when trying to synchronize with the Master clock from the Slave device. The Slave clock devices also communicate with multicast, which unfortunately does not work with StrongSwan as the same multicast configuration on both the Master and Slave clock causes traffic to become unencrypted due to policy failures. The two-way multicast communication is not possible with IPsec. Due to this multicast synchronization cannot be securely encrypted using IPsec, as the stability issues are very much prevalent.

To make the PTP synchronization work over IPsec, the unicast Slave would need to be used. The PTP unicast Slave clock works by specifying the Master clock statically by IP address or MAC address in the configuration file that can be seen in 3.20. The *ptp4l* daemon does not look for multicast *Announce* messages coming from the Master or Grandmaster clock, but instead, it starts to synchronize directly by sending the unicast *Announce* to the Master. This is done for all the Master clocks specified in the configuration file. The issue may be the fact that BMCA cannot be performed with any other potential Master clocks, meaning that synchronization options are limited to what is specified in the configuration. The BMCA is not however a crucial factor needed to perform required measurements. To avoid any misconfiguration, the Master clock is also set to unicast mode when operating over IPsec, with the unicast slave. It should also be noted that the delay, when synchronizing via unicast messages may be in nanoseconds higher, due to additional processing used when determining the packet route when departing.

Listing 3.20: Unicast Slave configuration

```
[unicast_master_table]
table_id                1
UDPv4                   192.0.2.1
[eth1]
unicast_master_table    1
```

What should also be noted is that if the *tshark* were not be run on the physical interface while StrongSwan is active in tunnel mode, one would see that incoming packets are showing twice, as "ESP packets and unencrypted as plaintext packets. However, for outgoing traffic, only ESP packets show up" [69]. As mentioned in the StrongSwan documentation, this is "a peculiarity of the Linux kernel" [69].

3.3.3 Macsec encryption

The Macsec security protocol as Layer 2 security protocol was defined by the IEEE standard, however, Linux kernel driver implementation and support was done by Sabrina Dubroca in 2015. Unlike with the previous instances of the security protocols, the implementation of the Macsec communication channel is rather straightforward. What on the other hand is significantly more difficult is the enabling of PTP timestamping for the Macsec virtual interface, which requires direct patching of the kernel source code, of which more shall be explained in the following subsection. As mentioned, the Macsec security protocol operates through a virtual interface which can be created with the help of a standard collection of utilities called *iproute2*. The creation of the Macsec network interface is rather straightforward. It is done by executing *ip link* as would be done when creating any other virtual network interface, with the addition of macsec-specific options, as seen in the 3.21.

Listing 3.21: Macsec interface creation

```
root@ptp_deviceA:~# ip link add link eth1 macsec0 type macsec
    encrypt on
```

Once the interface is created on both corresponding devices, the secure associations can be specified [70]. By secure associations, in this case, are meant the shared cryptographic properties between peer devices *ptp_deviceA* and *ptp_deviceB*. The system allows the usage of *ip macsec* command, which is used to specify the transmit secure association, only if the Macsec virtual interface exists. The transmit secure association specifies the initial packet number and the private key that is used by the current device. The private key, compatible with *ip* tool is generated with the

help of *urandom* character special device [71]. The key can be generated either directly in the terminal, as seen in the 3.22, or with the help of a different programming language, such as Python. The process of key generation and setting of the transmission security association is also done for the other device *ptp_deviceB*.

Listing 3.22: Macsec transmission security association [72]

```
root@ptp_deviceA:~# dd if=/dev/urandom count=16 bs=1 2> /dev/
   null | hexdump -e '1/2 "%02x"' > private_keyA
root@ptp_deviceA:~# ip macsec add macsec0 tx sa 0 pn 100 on
   key 01 `cat private_keyA`
```

What also needs to be configured is a receive channel and receive security association [70] Here it is required to specify the information of the peer device, that is MAC address and a private key that was generated for the device *ptp_deviceB*. Arguments that are used are very similar to those used to configure the receive security association, with the difference that the MAC address of the security association has to be specified, which is *ptp_deviceB*, as well as the private key of that device. The keys are in the current deployment exchanged with the help of a measurement script, as there is no other protocol employed to perform this task. Once again, the same process has to be done for the corresponding peer device, where the specified information is that of the device *ptp_deviceA*. Finally the virtual *macsec0* interface can be set to *up* state using *ip link* command and the IP addresses, which can be those belonging to *TEST-NET-3* address range [43], can be configured using *ip a* command. The configuration of the properties of the Macsec interface can be verified with the command *ip macsec show*.

The Macsec encapsulates everything, which is on the same layer as the Data Link Layer or above, therefore encrypting PTP messages in any operational mode should not be an issue.

3.3.3.1 Configuration and optimization for PTP

As was mentioned before the Macsec virtual interface does not by default support either software or hardware timestamping. To put matters more in perspective, the *ethtool* is used by *ptp4l* to determine if the given interface supports all the required timestamping modes used for the PTP synchronization. However considering that the Linux kernel implementation of Macsec does not by default provide sufficient pointers to the underlying physical interface to the *ethtool* function, the *ethtool* assumes that the timestamping is not supported by Macsec kernel driver and reverts to default values, as seen in the 3.23. Due to the fact that default values do not

specify any possibility of timestamping for transmission packets, it is not possible for *ptp4l* to function at all, either as a Master or a Slave clock.

Listing 3.23: Macsec before patching

```
Time stamping parameters for macsec0:
Capabilities:
        software-receive      (SOF_TIMESTAMPING_RX_SOFTWARE)
        software-system-clock (SOF_TIMESTAMPING_SOFTWARE)
PTP Hardware Clock: none
Hardware Transmit Timestamp Modes: none
Hardware Receive Filter Modes: none
```

Upon further inspection, it has been determined that this very crucial issue must be fixed by directly patching the source code of the Macsec driver. What needed to be done is an addition of C structures and functions to the Macsec kernel driver itself, which would provide the correct information to the *ethtool* when called on the interface created by *macsec*.

In the first implementation of kernel driver optimization, the custom static C structure *macsec_ethtool_ops*, of type *ethtool_ops* is defined, with four designated initializers to which are assigned functions that would provide all the required information to the *ethtool*, when called within *ptp4l* daemon. The *ethtool_op_get_link*, which is called from *ethtool* kernel drivers, returns a boolean value based on whether the specified interface is up or down. Another *ethtool* kernel driver function *ethtool_op_get_ts_info*, provides adequate information about the timestamping capabilities, independent of the device, which would allow for usage of *ptp4l*, however only with software timestamping options. The following two required functions that need to be defined by the user are *macsec_ethtool_get_link_ksettings*, which returns kernel helper again defined in *ethtool* driver [73], and *macsec_ethtool_get_drvinfo* which returns the version of the driver.

To implement the complete hardware timestamping capabilities specifically for the Macsec kernel driver new complex functions would need to be written from scratch. However, considering that the Macsec kernel driver points to *real_dev* structure, which represents the underlying physical interface, in several other functions, the same structure could also be used in order to obtain the timestamping capabilities of the physical interface, which can also provide hardware timestamps. This however requires replacement of the *ethtool_op_get_ts_info* with a different custom function, which can be called *macsec_get_ts_info* that can be seen in 3.24. This function would return pointers to the members of the *real_dev* structure, from which the timestamping possibilities can be determined. Additionally, a function

macsec_dev_ioctl must be added, which would handle all relevant *ioctl* calls and would set function pointers and call them with passed *ioctl* calls.

A very similar implementation of the above-specified requirements has for the most part already been done in kernel driver for VLAN [74]. This means that parts of relevant C structures and functions can be taken from the *vlan_dev* kernel driver and put into functions of the Macsec kernel driver described above. What needs to be added or changed are the pointers to the Macsec driver-specific structures, which refer to the underlying physical network interface, and Macsec driver-specific datatypes. Note that all above-specified function names starting with *macsec*, were named as such just for the sake of comprehension, however, the function names can be completely arbitrary.

Once the code is modified, the patch file can be created and added to the custom Yocto layer. The Yocto then re-compiles the kernel source code and creates all necessary links. Once the kernel is rebuilt, the standard configuration process for the creation of the *macsec0* interface can be done. The Macsec virtual interface now shall have all the same timestamping capabilities as the underlying physical interface. Upon executing *ethtool -T macsec0*, the output shall be identical to 3.8 and *ptp4l* may be used as with regular physical interface.

Listing 3.24: Macsec kernel driver function created according to VLAN driver [74]

```
static int macsec_get_ts_info(struct net_device *dev,
            struct ethtool_ts_info *info)
{
  const struct macsec_dev *macsec = macsec_priv(dev);
  const struct ethtool_ops *ops = macsec->real_dev->ethtool_ops;
  struct phy_device *phydev = macsec->real_dev->phydev;

  if (phydev && phydev->drv && phydev->drv->ts_info) {
     return phydev->drv->ts_info(phydev, info);
  } else if (ops->get_ts_info) {
    return ops->get_ts_info(macsec->real_dev, info);
  } else {
      info->so_timestamping =
        SOF_TIMESTAMPING_TX_SOFTWARE |
        SOF_TIMESTAMPING_RX_SOFTWARE |
        SOF_TIMESTAMPING_SOFTWARE;
      info->phc_index = -1;
  }

  return 0;
}
```

4 Measurements

4.1 Automatization and measurement tool

For the purpose of the PTP measurements and data visualization, a specific testing tool was written in Python programming language, which sets the correct environment for all possible testing scenarios and performs measurements, based on the values parsed from the *ptp4l* logs. This also includes many necessary configurations which were described in the previous chapters. The measurement script is executed by running *python setup_and_measure*, with additional options that can be obtained by using *-h* argument, as seen in 4.1. Additional parameters, such as IP addresses, directory for data saving, measurement timer, and others must be specified in *confdata.yml* file.

Listing 4.1: Measurment tool options

```
me@MyPc:~$ python setup_and_measure/  -h
usage: analyzer_main [-h] [-a] [-sw] [-hw] [-nenc] [-enc] [-
   stat] [-mes] [-packets] [-ntp] [-pull]

Configuration, Measurement and Analysis tool; Specify
   additional options in confdata.yml

options:
  -h, --help  show this help message and exit
  -a          Enable everything
  -sw         Enable measurement with software timestamping
  -hw         Enable measurement with hardware timestamping
  -nenc       Enable measurement with no encryption
  -enc        Enable measurement with all security protocols
  -stat       Do statistics and plot comparisons (There must
     be existing data in dir)
  -mes        Do measurements on provided Master & Slave
     devices
  -packets    Run packet analysis & plotting
  -ntp        Enable NTP synchronization on Master
  -pull       Pull and cross-compile necessary packages
```

The very core of the measurement script assumes that all Master clock and Slave clock devices are able to communicate via *Secure Shell Protocol* (SSH). The SSH creates a secure connection between the host and the destination device, allowing

the user to remotely log in and operate on the device based on the login privileges. In Python, this can be done with the help of *Paramiko* library that allows full usage of all features provided by Linux *ssh* utility within the Python framework [75]. The Paramiko essentially works by creating a Python object based on the SSH connection parameters, which are: IP address, username and password, and allowing different actions to be performed on the remote device, based on the methods provided by the Paramiko object. The provided method called, *exec_command* allows for execution of Linux commands on the remote machine, where the feedback can be obtained from the standard output [76]. For the sake of simpler configuration and measurement, an additional Python class called *MySSHClient* was written, which inherits from the parent Paramiko class and provides two separate methods for reading long or short standard outputs. This is initialized as an object with the Master clock device and Slave clock device parameters.

For each security protocol, a different separate child class is initialized, which initializes the protocol and verifies the connection between the Master and the Slave device, based on the requirements of the given protocol. This includes key generation and key exchange, setting of the virtual interface and addressing, mode selection, and other configurations described in the 3.3. All initialized classes working with security protocols inherit from the same parent class *SecUtils*, from which many similar methods can be used. When the method, belonging to an object that works with a specific security protocol, *do()* is called, all the configuration, for the Master clock and Slave clock device is performed and is considered ready to be used for PTP communication. Note that this can be done only if all the installations and kernel requirements, described in the 3.1.2, are fulfilled.

The reading of PTP log data is done with the use of Python standard library *re*, concerned with regular expression operations. This allows for the parsing of logs returned by the *ptp4l* daemon on the Slave clock device. The separately initialized class called *PtpReader* provides method *do()*, which executes *ptp4l* commands based on the predefined parameters on both Master and Slave devices and parses numerical data from the returned logs. The methods extracted from *PtpReader* show the use-case of parallel iteration, as can be seen in 4.2. The data are parsed by lines and are yielded to another function which saves data into buffers. The buffers which are in the form of *pandas* library *DataFrame*s, must contain always a fixed number of values in all columns before they are to be further processed. This is done to ease the load on the system and prevent complete loss of data in case of failure in long-duration measurement scenarios. The *PtpReader* initializes another class called *PTPSinglePlotter*, which inherits from the *PlotUtils*, and uses *matplotlib* Python

library for data plotting. Plotting is always performed from values contained in a given buffer, and the mean value is always actively calculated for a given set of data. This allows the user to observe a graphical representation of data in a **real-time**, as well as the mean value for the plotted data. The values from buffers are also at the same time saved into *csv* files, which are to be used by a different function for the creation of statistics and comparison plots. *Tshark* tool is also started on the Slave clock device, with the call of *do()* method and created capture files are afterward sent back to the user system via *scp* for later analysis.

After each measurement, the initialized security protocol is disabled so as not to hinder any further measurement operations. Once all measurements are performed the *do()* method from previously initialized class *StatMakerComparator* is called and further analysis and plotting of collected data is performed. The *StatMakerComparator* contains various methods that allow for statistics table generation, comparison plot generation, packet statistics plot generation, and other operations. In the post-measurement analysis, all the data are obtained from the previously saved *csv* files.

All operational modes for PTP measurements are predefined in a Python dictionary that is imported from *vardata.py* module. The previously mentioned *PTPReader* is initialized with this dictionary and reads the specific *ptp4l* operational modes. The *vardata.py* also imports directly the configuration specified, by the user, such as IP addresses, directories for data saving, and other specifications.

Listing 4.2: Synchronization and parsing methods extracted from PtpReader

```
def __run_sync(self, ptp_cmd_master, ptp_cmd_slave):
    for line_m, line_s in zip(
        self.ssh_master.run_continous(ptp_cmd_master,
                                      self.timer),
        self.ssh_slave.run_continous(ptp_cmd_slave,
                                     self.timer),
    ):
        data = self.__parse_lines(line_s)
        log_time = 0
        if data:
            if log_time != 0:
                assert data["ptp4l_runtime"] == (log_time
                                                 + 1)
            yield data

def __parse_lines(self, line):
    tmp_dict = {}
    nums = []
    matches = re.findall(self.pattern, line)
    if servo_state := list(self.servo_states
                           & set(matches)):
        assert len(servo_state) == 1
        servo_state = servo_state[0]
        matches.remove(servo_state)
        for i in range(len(matches)):
            if matches[i] == "+" or matches[i] == "-":
                matches[i + 1] = (matches[i] +
                                  matches[i + 1])
            else:
                nums.append(float(matches[i]))
        nums.append(int(servo_state[-1]))
        if len(nums) == len(self.labels):
            for i in range(len(self.labels)):
                tmp_dict[self.labels[i]] = nums[i]
            return tmp_dict
```

4.2 Results

Upon performing measurements by using the script described in the previous section a number of results can be obtained for analysis. However, before that is done, it should be made clear which operational modes described in the previous chapters are supported. The complete overview of compatible operational modes can be seen in 4.1.

Encryption mode	Timestamping		Encapsulation		Communication	
-	SW	HW	UDP	L2	Multicast	Unicast
No encryption	✓	✓	✓	✓	✓	✓
Wireguard-go	✓	×	✓	×	✓	✓
IPsec - tunnel	✓	✓	✓	×	×	✓
IPsec - transport	✓	✓	✓	×	×	✓
Macsec	✓	✓	✓	✓	✓	✓

Tab. 4.1: PTP possibilites

Another important fact that must be addressed is that due to the availability of only two PTP-enabled devices for measurement purposes, all measurements will be performed solely in *E2E* mode, as *P2P* mode has no utilization unless there are more than three devices available. The type of synchronization is two-step mode.

The unencrypted state for PTP is considered to be the desired operational state, assuming that PTP messages should not be affected by any external factors. However, in order to optimize the cybersecurity of PTP to the furthest extent, it must be determined which security protocol is most optimal and has the least effect on the synchronization process. Data that are to be used to determine this factor were collected by running the script described in the section 4.1, for a 45-minute time period for each supported mode. The total measurement time was therefore around 17 hours. The specific values that are to be considered in the performance analysis are *master offset*, *servo change frequency*, *path delay*, and the *servo stabilization state*. All these are collected from the Slave clock side when the *PTP* is synchronized in one of the supported modes while operating either unencrypted or under security protocol.

As to the physical connection, the Slave and Master clock devices were connected directly by a standard Cat 5e Ethernet cable of approximately 30 cm in length. It must be noted that due to the fact that all measurements are performed in a point-to-point connection, the showcased behavior may not reflect the behavior in a switched network. Note that the process of synchronization is associated with a certain level

of randomness, such as packet jitter or delays caused by processing. The factors of randomness can be observed especially in scenarios where significantly small values are being measured, such as nanoseconds or ppb.

4.2.1 Visualization

4.2.1.1 Methods

The scatter plots in this section show the output generated in real-time by the script described in the 4.1, where the plot updates were always reoccurring by buffers of data with the predefined sizes. The mean values were also calculated in real-time, by taking the mean value of the single buffer, adding it to the means of existing same-size buffers, and finally returning the mean of means. The x-axis in the provided plots always represents data taken from a single *ptp4l* log, which corresponds approximately to one second. Other plots were generated also using the same script, but not in real-time. The y-axis describes the values of a specific parameter, with the unit in square brackets. The names of the parameters are connected with the underscore. The offset values are described as *master_offset*, the change of servo frequency is described as *servo_freq*, the delay is described as *path_delay*, and the servo state is described as *state*. All measurements were performed numerous times in order to verify the consistency of the results.

Regarding the plotting of comparisons between different operational modes, all values that precede before the servo state is changed to locked are removed, including an additional twenty values after the locking of the servo. The twenty-sample cut-off value was specified due to the fact that even after servo locking, other parameters may take some time to stabilize. From multiple observations, it could be determined that complete normalization of values happens before the tenth sample. However, for the sake of complete sureness, an additional removal of ten data samples was performed. This is done to eliminate initial outliers in data and provide more relevant data visualization for comparison plots. The different operational modes in comparison plots are visualized in various colors, where each is described in the disclosed legend.

Only some selected results are showcased in this section. The rest of the results can be viewed in the electronic appendix.

4.2.1.2 No encryption

As can be seen in the figure 4.1, the initial sudden change of the Master clock offset corresponds to the change of state of the servo algorithm, which changes its values

to *2*, meaning that it becomes locked. This also correlates to the servo frequency that becomes stabilized after the initial spike and stays stabilized for the duration of the measurement. It should be noted that even though initial spikes are visually observable, they can be considered negligible when taking into account that actual changes in values are rather small.

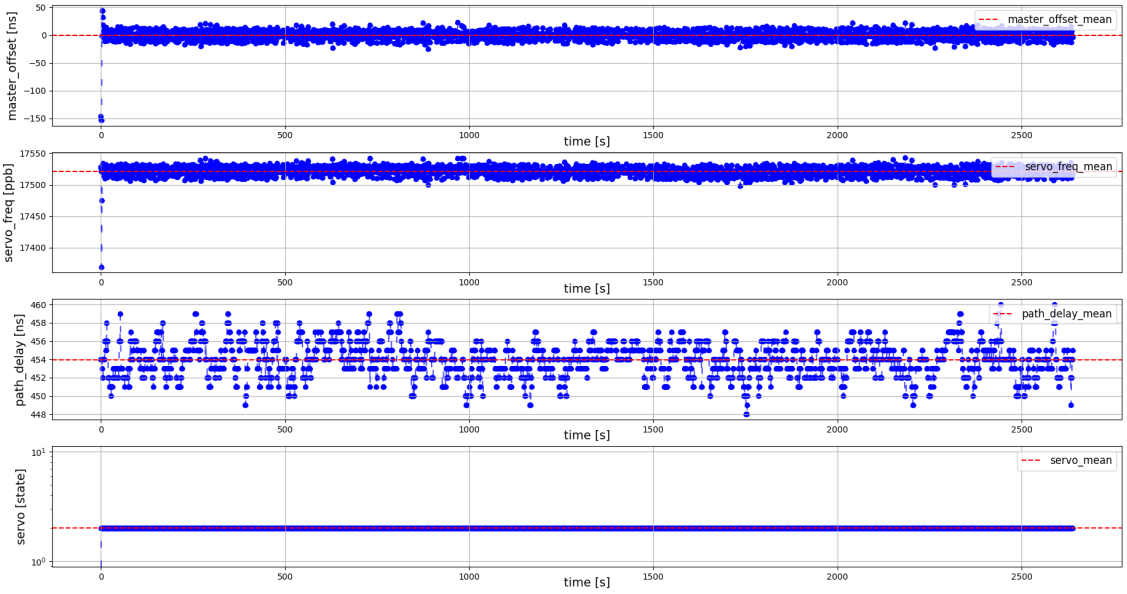


Fig. 4.1: No encryption; Multicast; L2 encapsulation; Hardware timestamping

A very similar behavior, with only marginal differences, can be observed in all other scenarios where PTP communicates with no additional encryption. The servo stabilization also happens very quickly in all cases, as can be seen by observing the change of servo state, from unlocked to locked. Figure 4.2 demonstrates a histogram, where the stability of all parameters, can be observed through the consistency of bin distribution.

This concludes that communication and encapsulation modes of PTP messages, seem to have little impact on the actual clock synchronization parameters in point-to-point communication, where the hardware timestamping is used. This behavior is also expected to be reflected in scenarios where security protocols are applied.

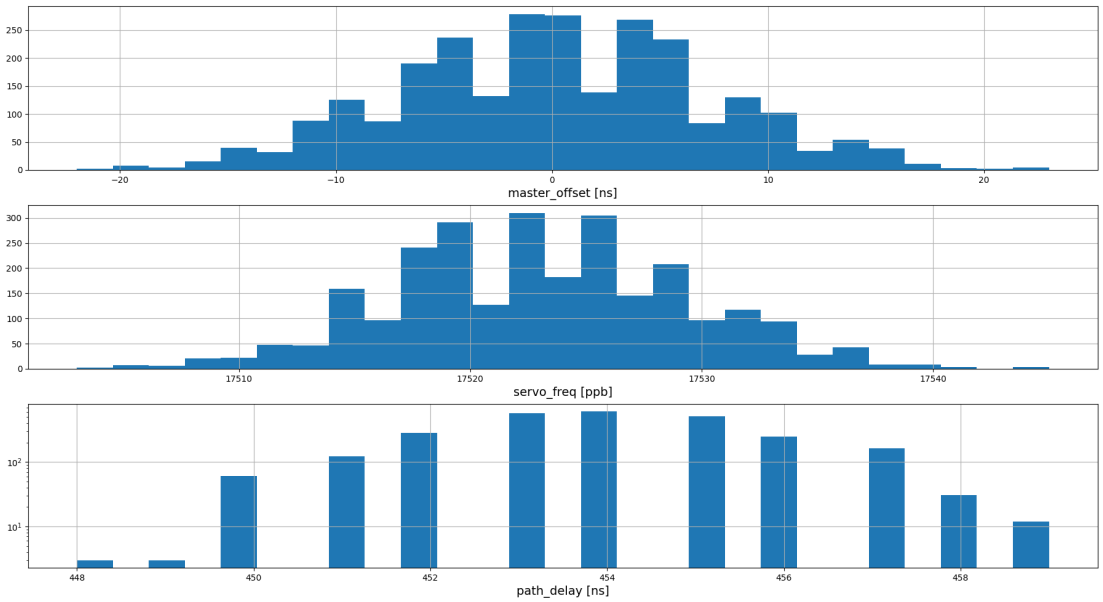


Fig. 4.2: No encryption; Unicast; UDP encapsulation; Hardware timestamping

Very different behavior is however observable in scenarios where the software timestamping is used. As seen in the 4.3, not only does the servo stabilize after a longer time, but also path delay values are significantly higher. As for the master offset and servo frequency values, occasional spikes in data occur even once the servo is stabilized. It can also be noticed that with the instantaneous change of the master offset, an immediate change in the servo frequency is observed as well. The servo algorithm tries to adjust the Slave clock according to the instantaneous offset value in order to mitigate the time difference between the Master and the Slave clock as fast as possible. It can also be noticed that irregularities start to occur in the latter half of the measurement, after the 1500-second mark. In the case where the PTP messages are encapsulated in UDP, very similar results with similar irregularities are observable, however rather sparsely.

This behavior seems to be even further intensified when changing the communication mode to unicast. As observed in the 4.4, the spikes are much more prominent in all stages of measurement. Curiously, the sudden changes in offset and servo frequency do not seem to be reflected in path delay values.

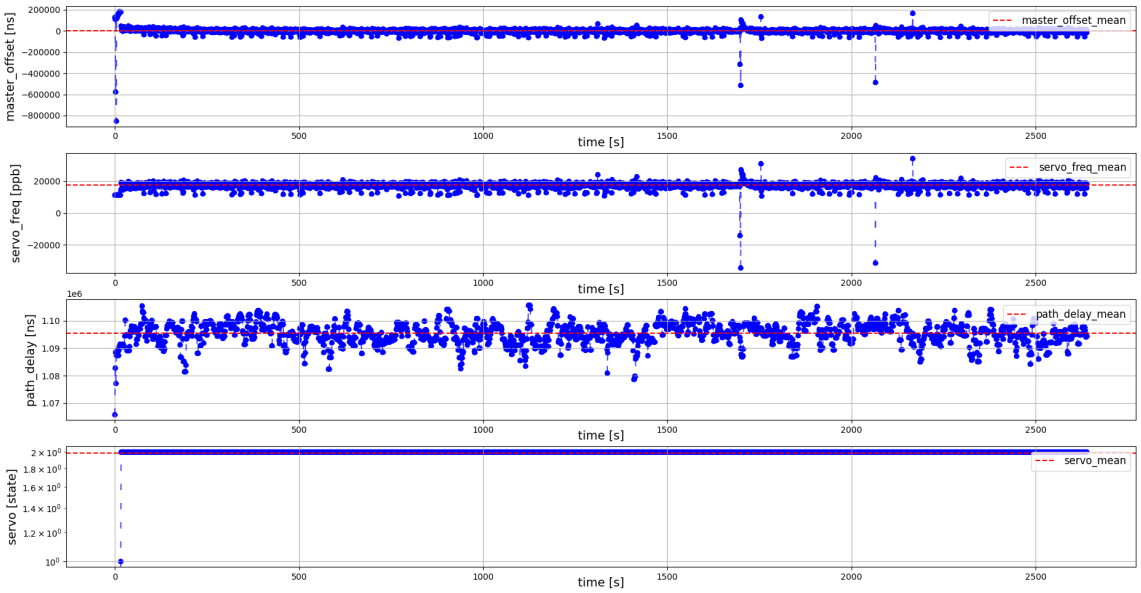


Fig. 4.3: No encryption; Multicast; L2 encapsulation, Software timestamping

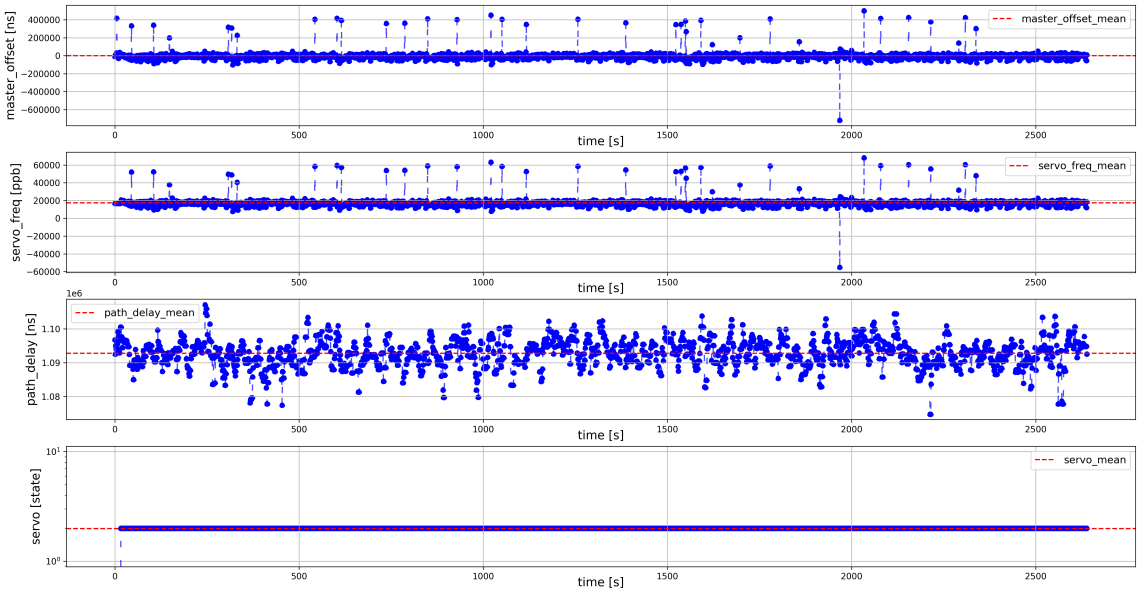


Fig. 4.4: No encryption; Unicast; L2 encapsulation, Software timestamping

Additionally, by plotting the times between the captures of consecutive PTP packets for the same unicast communication mode, which can be seen in 4.5, it can be observed that there are no noticeable inconsistencies in packet arrival times during the whole duration of the measurement. The figure demonstrates that arrival times between messages are never higher than one second and that there are no noticeable inconsistencies.

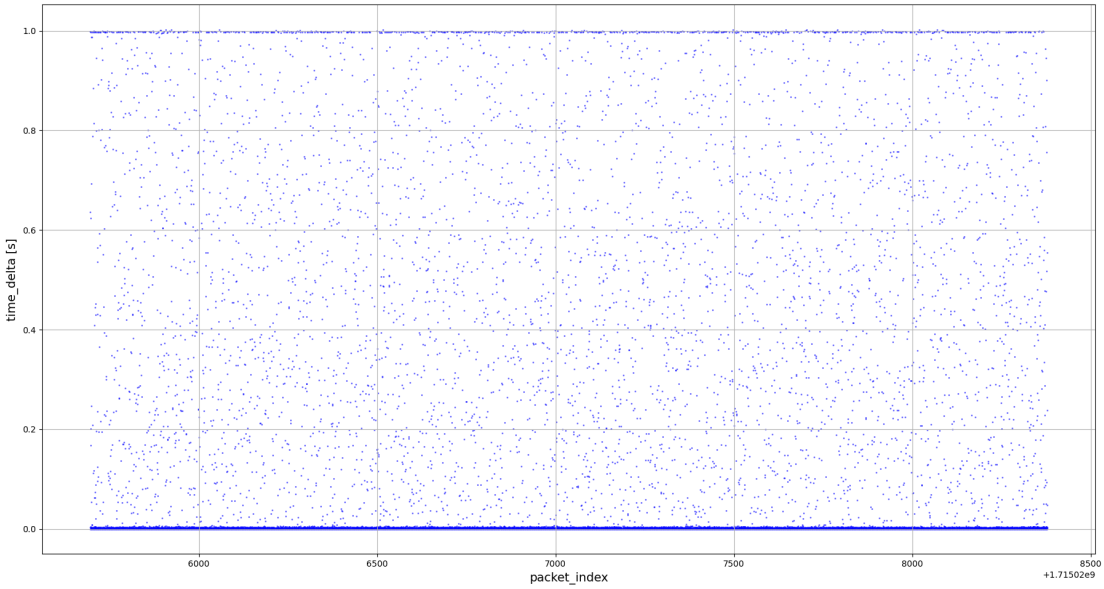


Fig. 4.5: No encryption; Unicast; L2 encapsulation, Software timestamping - packet time deltas

The first assumption as to why such irregularities occur was that there is some sort of interference coming from active processes on either Master of Slave clock devices, such as *chronyd* service for NTP synchronization. Nevertheless, after disabling *chronyd* and other potentially disrupting processes, the irregularities when using the software timestamping, still seem to be prevalent. To determine the root cause of irregularities occurring in offset and servo frequency values further research would have to be conducted analyzing more deeply the behavior of PTP stack in Linux and the behavior of software timestamping. The comparison of modes with software timestamping can be seen in 4.6.

It can be however assumed that additional processing done by the Linux network stack in combination with *ptp4l* causes inconsistencies when the software timestamp is delivered to the relevant process. It can be concluded that this behavior is also

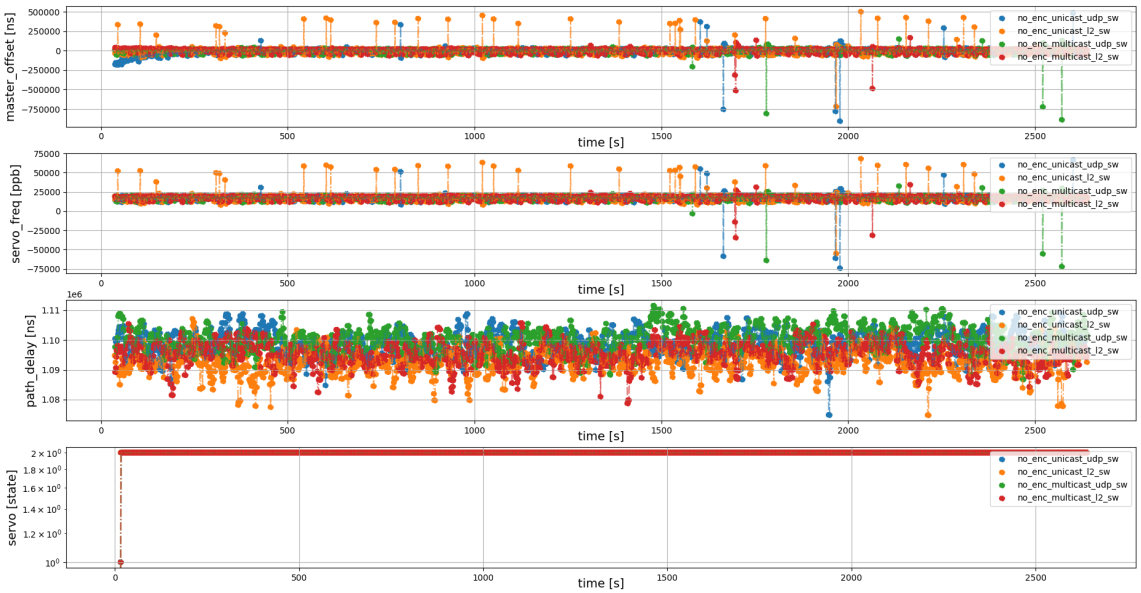


Fig. 4.6: Comparison, No encryption, Software timestamping, (removed initial outliers)

expected to be observable when communicating over all security protocols that use software timestamping.

4.2.1.3 With encryption

By analyzing the results of PTP operations under different security protocols it can be noticed that IPsec and Macsec had very little impact on the measured parameters when used with hardware timestamping. As can be seen in the comparison plot 4.7, where IPsec was plotted in two different operational modes, the behavior is very similar to the scenarios where no encryption is in use. Note that even though in theory IPsec tunnel mode is expected to create more overhead due to the addition of a new IP header, in practice the effect on PTP parameters is not very significant.

Similar results can be seen in figure 4.8 with Macsec encryption in use, where the results very much resemble those of PTP with no encryption.

It should be however noted that due to the fact that in the configuration process of the security protocols, as a part of the measurement script, both Master and Slave already communicated with each other by issuing *ping* command, thus exchanging all the necessary security association information. This means that the servo stabilization period is significantly shortened, as compared to the situation where PTP

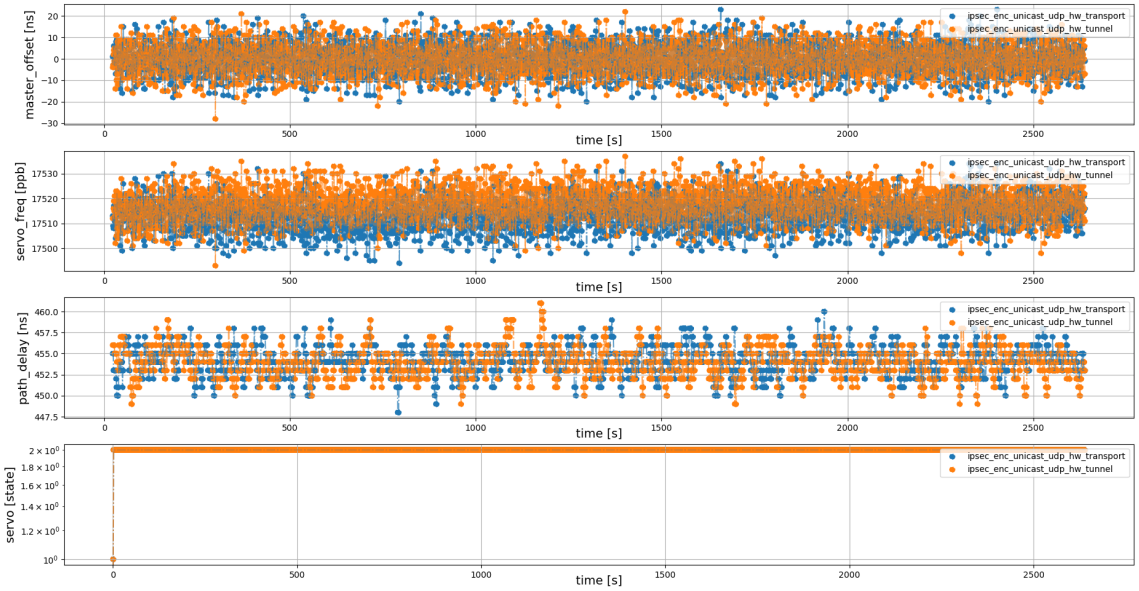


Fig. 4.7: Comparison, IPsec encryption, Hardware timestamping, (removed initial outliers)

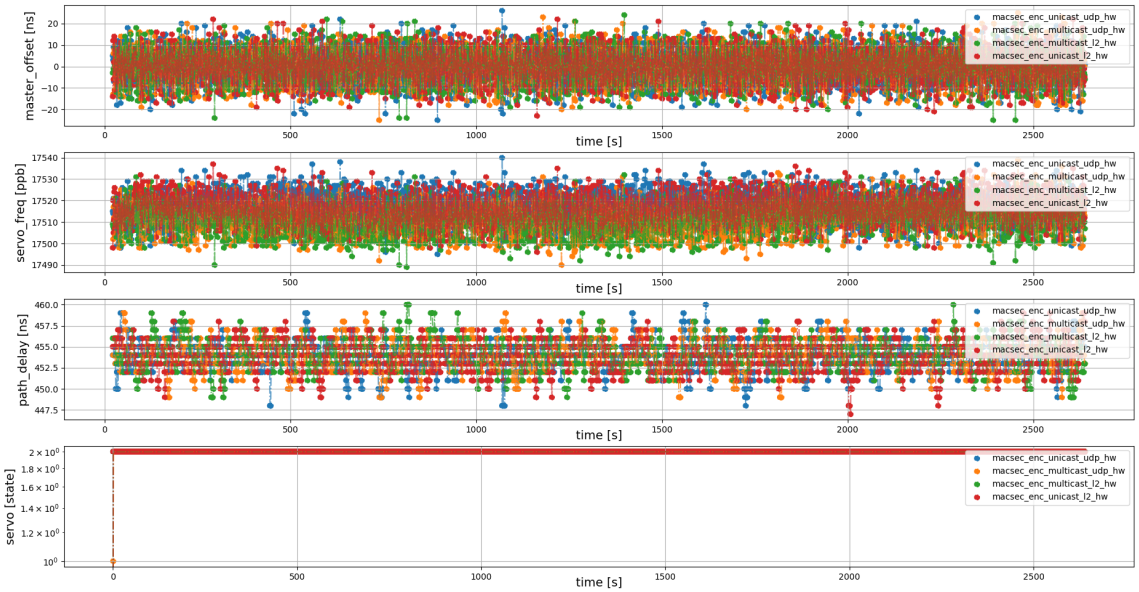


Fig. 4.8: Comparison, Macsec encryption, Hardware timestamping, (removed initial outliers)

messages were to start being exchanged right away after security protocol configuration, which would significantly delay the initial exchange of PTP messages.

In regard to software timestamping, it could be previously observed in scenarios with no encryption that various inconsistencies in measured parameters occur, which are even more intensified when communicating via unicast. Figure 4.9 shows such a scenario, where spikes are even more prominent due to the fact that IPsec operates solely in the unicast mode.

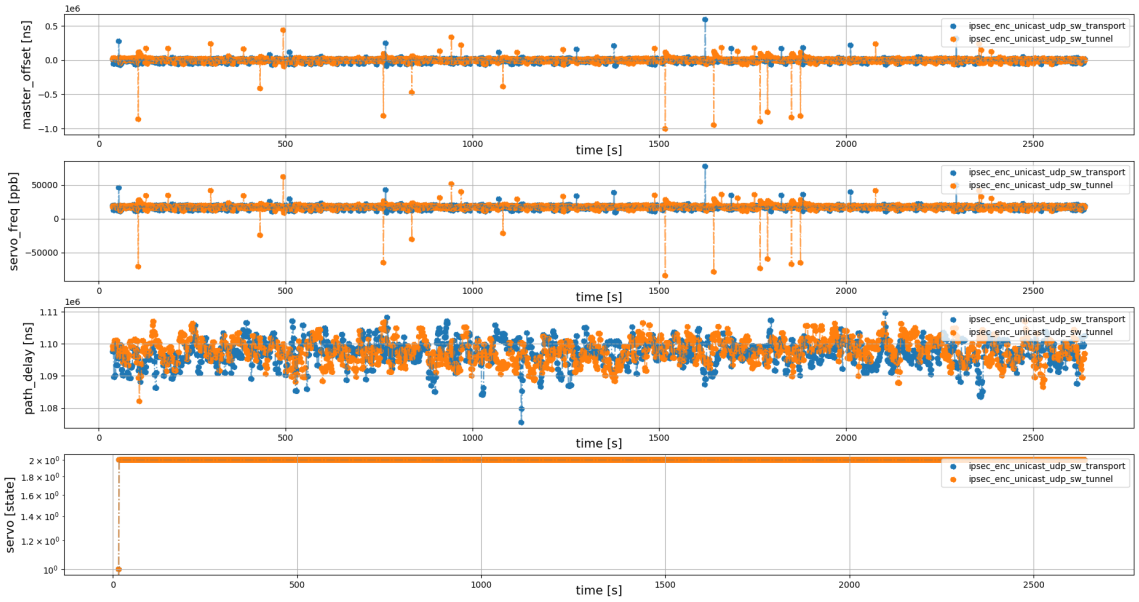


Fig. 4.9: Comparison, IPsec encryption, Software timestamping, (removed initial outliers)

By looking at figure 4.10 representing measurements with Macsec encryption, it can be observed that behavior remains very similar to that of a PTP with no encryption and software timestamping. Nevertheless, it does not seem that the Macsec security protocol causes any more significant instabilities than those already existing in a regular operational state, as all the deviations stay using the same range of values as in non-encrypted mode.

Finally, by observing the measurements with Wireguard encryption in 4.11, it can be noticed that the impact on the PTP parameters is very significant. The base fluctuations of the seemingly stabilized state overcome even the spikes observed in other security protocols when unicast is used. Also, the spikes that occur, in Wireguarad when *ptp4l* operates in unicast, represent significantly higher values

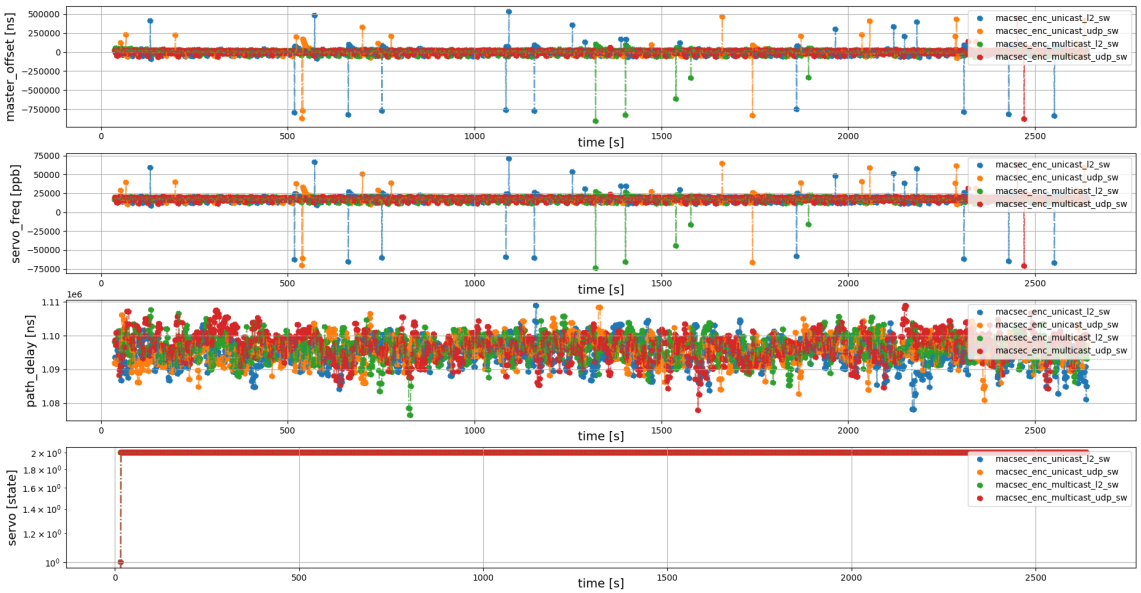


Fig. 4.10: Comparison, Macsec encryption, Software timestamping, (removed initial outliers)

than observed in any other security protocols. Nonetheless, the worse performance for Wireguard was expected as the version written in the Go language operates entirely in the user space, therefore severe performance limitations are in place.

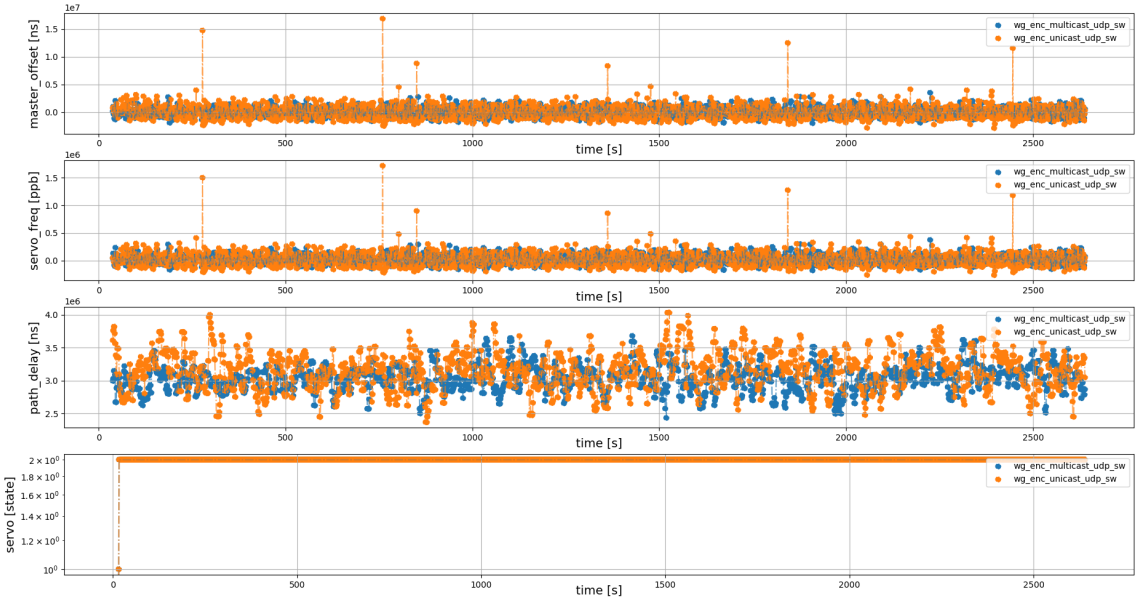


Fig. 4.11: Comparison, Wireguard encryption, Software timestamping, (removed initial outliers)

4.2.2 Numerical data analysis

4.2.2.1 Methods

This section shall look into the statistical properties of the data collected in real time. The statistics tables were generated by measurement script described in 4.1 from all the collected values.

Considering that differences in performance results of measurements with software and hardware timestamping are rather significant, they will be analyzed separately to provide better insight into the properties of encryption protocols and PTP operational modes. Note that for the statistical analysis, all sample values before the servo became locked were removed, as well as the additional twenty samples after the locking of the servo. This is done in order to eliminate initial outliners and ensure that all statistically analyzed parameters represent the behavior of PTP in a fully stabilized servo state, as was mentioned in the previous section.

The exact statistical parameters used for analysis are *mean*, *median*, *mean - median*, *variance*, *standard deviation*, *absolute mean deviation*, *absoluted median deviation*. Additionally, z-score and *Interquartile range* (IQR) statistical measures were used in order to calculate the probability of spikes in values. Both of these measures are used to determine the categorization of outliers in data and detect

them. Based on the number of outliers in the data, the total spike percentage can be calculated by using an empirical probability equation *P(A)* as seen in 4.1.

$$P(A) = \frac{\text{Number of spikes}}{\text{Total number of samples}} \tag{4.1}$$

However before it is proceeded with a description of the spike detection methods, the probability distribution and its parameters must be described. Considering the nature of the behavior of all previously visualized *ptp4l* parameters, the probability distribution can be considered to be normal, due to tolerances introduced by the application use case scenario requirements. This is determined by visually observing the behavior of parameters in 4.2.1, where it can be noticed that the sample data always have a tendency to be distributed in the proximity of the mean value, with occasional spikes. Unless there are some major inconsistencies in the system the *ptp4l* and related processes always try to keep the time offset and servo frequency values as consistent as possible. As for the path delay, point-to-point communication assures that there are no long-term inconsistencies that would cause the skewness of the distribution. This may however not be applicable in scenarios with switched networks, where path delays may show different tendencies.

From a purely statistical perspective, the probability distribution is likely to be considered more left or right-skewed depending on the measurement results, which can be determined by the difference between mean and median. In case the resulting value of the difference is negative, the probability distribution is skewed to the left, if the resulting value is positive, the probability distribution is skewed to the right. The description of exact values is provided in the following sections.

The parameters that are commonly used to describe probability distribution are the mean, denoted as μ, and standard deviation, denoted as σ. The μ can be described as seen in 4.2, where each data sample is denoted as x_i and n describes the total number of data samples [77]. The σ can be described as seen in 4.3 [77], where the same variables are used. If the square root were to be removed from the formula, the attribute would be called variance σ^2.

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{4.2}$$

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2} \tag{4.3}$$

The z-scores are calculated by taking the previously calculated mean value (μ) of all data samples, taking the previously calculated standard deviation (σ) of all data samples, and applying the formula 4.4 on a single data sample x [78]. The

z represents the z-score for a data sample x. Data samples which are then to be considered outliers are selected according to their z-score.

$$z = \frac{x - \mu}{\sigma} \tag{4.4}$$

The IQR is calculated by rearranging all data samples from smallest to largest and then finding the first quartile (*Q1*) and third quartile (*Q3*) and then subtracting as seen in formula 4.5 [78].

$$IQR = Q3 - Q1 \tag{4.5}$$

Lower and upper boundaries are then calculated as demonstrated in formulae 4.7, where the *multiplier* is selected according to the strictness of the measurement. The data samples that are to be considered outliers are either below the *Lower_boundary* or above the *Upper_boundary*.

$$Lower\_boundary = Q1 - multiplier * IQR \tag{4.6}$$

$$Upper\_boundary = Q3 - multiplier * IQR \tag{4.7}$$

These measures may yield slightly different results due to different statistical properties and different sensitivity to outlier detection. Considering the fact that spikes tend to be rather extreme, as could be observed in the measured data plots in 4.2.1, the parameters for these statistical measures were selected as seen in the table 4.2.2.1.

Method	Threshold
IQR Multiplier	3.5
Z-Score	3

Tab. 4.2: Outlier detection criteria

Only statistical measures for time offset, which is denoted as *master_offset*, are showcased in the following section, as it is the most representative parameter of PTP behavior. The rest of the results can be viewed in the appendix and electronic appendix.

4.2.2.2 Hardware timestamping

The statistical measures observed in table 4.3 show that all the operational modes provide very similar statistical parameters with only negligible differences. With the mean value being close to zero and the median being zero in most cases, it can be determined that synchronization with the Master clock device is very precise. The multicast mode in Ethernet encapsulation shows a slight skewness tendency which is however negligible considering the values being in nanoseconds. The absolute mean and absolute median deviation indicate slight deviation from its respective parameters, which however stays under six nanoseconds. The standard deviation and variance show relatively low values, which serves as further proof of stability. The IQR measure shows a zero percent probability of spikes, which can be considered a reliable factor. Even though the z-score measure shows a small probability of spikes, it must be taken into account that the z-score is much more sensitive to outliers, meaning that some values may be falsely detected as spikes.

master_offset	multicast_udp	multicast_l2	unicast_udp	unicast_l2
mean [ns]	-0.000764	-0.000382	0.001528	-0.000764
median [ns]	-1.000000	0.000000	0.000000	0.000000
mean - median [ns]	0.999236	-0.000382	0.001528	-0.000764
abs. mean dev. [ns]	5.993082	5.506890	5.715135	5.743349
abs. median dev. [ns]	5.000000	4.000000	5.000000	5.000000
standard dev. [ns]	7.306437	7.210917	7.109586	7.153827
variance [ns^2]	53.384027	51.997326	50.546216	51.177234
spike prob. (Z-score) [%]	0.114635	0.305577	0.229183	0.152788
spike prob. (IQR) [%]	0.000000	0.000000	0.000000	0.000000

Tab. 4.3: Master to Slave time offset statistics, No encryption, Hardware timestamping

The statistical parameters for both IPsec tunnel and IPsec transport also indicate stability comparable to the modes with no encryption, as can be seen in 4.4.

master_offset	unicast_udp_tunnel	unicast_udp_transport
mean [ns]	-0.000382	0.006114
median [ns]	0.000000	0.000000
mean - median [ns]	-0.000382	0.006114
abs. mean dev. [ns]	5.870481	5.823810
abs. median dev. [ns]	5.000000	5.000000
standard dev. [ns]	7.244225	7.207284
variance [ns^2]	52.478792	51.944938
spike prob. (Z-score) [%]	0.152847	0.076423
spike prob. (IQR) [%]	0.000000	0.000000

Tab. 4.4: Master to Slave time offset statistics, IPsec encryption, Hardware timestamping

Finally, by looking at Macsec statistics in 4.5, where all PTP operational modes are supported, it can be noticed that properties are nearly identical to the case with no encryption.

master_offset	multicast_udp	multicast_l2	unicast_udp	unicast_l2
mean [ns]	0.000000	-0.001146	0.008785	0.006875
median [ns]	-1.000000	0.000000	0.000000	0.000000
mean - median [ns]	1.000000	-0.001146	0.008785	0.006875
abs. mean dev. [ns]	5.658387	5.787256	5.892278	5.877373
abs. median dev. [ns]	4.000000	5.000000	5.000000	5.000000
standard dev. [ns]	7.218570	7.272419	7.340325	7.260878
variance [ns^2]	52.107757	52.888081	53.880366	52.720350
spike prob. (Z-score) [%]	0.152847	0.267380	0.076394	0.152788
spike prob. (IQR) [%]	0.000000	0.000000	0.000000	0.000000

Tab. 4.5: Master to Slave time offset statistics, Macsec encryption, Hardware timestamping

Curiously it can be noticed that in all cases where multicast communication happens over UDP, the median offset value is -1. It should be implied that this is likely a coincidence with no further implication for PTP behavior. It can be therefore concluded that security protocols, when used with hardware timestamping,

have little to no impact on the time offset between the Master clock and the Slave clock in point-to-point communication.

4.2.2.3 Software timestamping

The table 4.6 shows statistics for PTP operational modes with no encryption. Already by looking at mean and median, it can be noticed that values show significantly higher offset as compared to hardware timestamping. The difference between the median and indicates distribution skewness in all scenarios, indicating lessened preciseness of synchronization between the Master and Slave clock. The most noticeable case can however be seen in the scenario where unicast communication with UDP encapsulation is used. The absolute mean and median deviations also indicate significant variability in all cases, where again measures with unicast communication mode show higher variability around the mean and median. This can be further verified by looking at standard deviation and variability values, which are also higher, compared to multicast communication modes. This is caused by the spikes that occur significantly more in the unicast modes. It can be noticed that both z-score and IQR measures also have a higher probability of spikes.

master_offset	multicast_udp	multicast_l2	unicast_udp	unicast_l2
mean [ns]	-76.626826	887.488470	-5663.757494	-409.686900
median [ns]	1500.000000	2514.500000	-938.000000	-1689.000000
mean - median [ns]	-1576.626826	-1627.011530	-4725.757494	1279.313100
abs. mean dev. [ns]	12931.681166	12674.044938	19044.885933	20748.314841
abs. median dev. [ns]	9080.500000	9386.000000	10570.000000	12083.000000
standard dev. [ns]	32563.671723	22930.234403	42333.245522	48443.222736
variance [ns^2]	1.060393e+09	5.257956e+08	1.792104e+09	2.346746e+09
spike prob. (Z-score) [%]	0.307456	0.345888	1.268255	1.267768
spike prob. (IQR) [%]	0.499616	0.307456	2.843966	1.344602

Tab. 4.6: Master to Slave time offset statistics, No encryption, Software timestamping

The measurement results where IPsec encryption is used, which can be seen in 4.7, are mostly similar to scenarios with no applied encryption, UDP encapsulation, and unicast communication. Interestingly the the mean and median values seem lower when compared to scenarios with no encryption. However, when looking at the standard deviation and variance, these values seem to be considerably higher in IPsec tunnel mode, compared to scenarios with IPsec transport and no encryption.

This is likely caused by overhead due to the creation and addition of an additional IP header. However, it can be noticed that the spike probability is not increased.

master_offset	unicast_udp_tunnel	unicast_udp_transport
mean [ns]	685.627210	-242.541891
median [ns]	1454.500000	402.500000
mean - median [ns]	-768.872790	-645.041891
abs. mean dev. [ns]	18388.892218	14032.873247
abs. median dev. [ns]	10450.500000	9612.000000
standard dev. [ns]	57137.906324	25440.252987
variance [ns^2]	3.264740e+09	6.472065e+08
spike prob. (Z-score) [%]	0.807071	0.576480
spike prob. (IQR) [%]	1.844735	0.576480

Tab. 4.7: Master to Slave time offset statistics, IPsec encryption, Software timestamping

By looking at the measures for the Macsec encryption scenario in 4.8, it can be seen that behavior in all modes comes very close to those with no encryption, with some differences that can be however attributed to randomness caused by packet jitter.

master_offset	multicast_udp	multicast_l2	unicast_udp	unicast_l2
mean [ns]	24.118370	1223.571264	578.602766	52.556879
median [ns]	808.500000	2773.000000	559.000000	1624.500000
mean - median [ns]	-784.381630	-1549.428736	19.602766	-1571.943121
abs. mean dev. [ns]	11831.190453	13208.535717	14025.511338	18278.241024
abs. median dev. [ns]	9063.500000	9123.000000	8932.000000	10011.000000
standard dev. [ns]	24703.901199	33211.815791	39382.458237	55527.100472
variance [ns^2]	6.102827e+08	1.103025e+09	1.550978e+09	3.083259e+09
spike prob. (Z-score) [%]	0.153728	0.230503	0.806761	0.730208
spike prob. (IQR) [%]	0.153728	0.268920	1.383020	1.152959

Tab. 4.8: Master to Slave time offset statistics, Macsec encryption, Software timestamping

The statistics for PTP modes with Wireguard encryption, as seen in 4.9, show significantly higher values than any other operational modes. It can be noticed that the median is exponentially higher in both scenarios, as well as the skewness of

data. The standard deviation and variance also display quite a significant variation in values. It can be noticed that spike probabilities are lower than in other security protocols, however, it must be taken into account that calculated percentages are relative to existing values. Overall the Wireugard-go seems to provide the least stability in the case of PTP synchronization with PTP software timestamping.

master_offset	multicast_udp	unicast_udp
mean [ns]	-4350.662183	18181.068793
median [ns]	-77185.500000	-61637.500000
mean - median [ns]	72834.837817	79818.568793
abs. mean dev. [ns]	571390.851630	812345.643964
abs. median dev. [ns]	455109.000000	666066.500000
standard dev. [ns]	729749.326843	1.157275e+06
variance [ns^2]	5.325341e+11	1.339286e+12
spike prob. (Z-score) [%]	0.422752	0.461184
spike prob. (IQR) [%]	0.000000	0.230592

Tab. 4.9: Master to Slave time offset statistics, Wireguard encryption, Software timestamping

4.2.3 Selection of the most optimal security protocol

Based on the analysis of security protocols using visual and statistical data it can be concluded that the Macsec security protocol is the most optimal security protocol as it shows very little difference in the behavior of PTP. It also supports all encapsulation, communication, and timestamping modes, unlike the other two protocols. The biggest limitation of Macsec is however the fact that it communicates solely on the Data Link Layer, meaning that MAC addresses are used as communication endpoints and encrypted communication can happen only in LAN. This may pose a challenge to PTP as synchronization between different networks may be required. For this the IPsec protocol be used, however, it must be taken into account that only unicast communication is supported, meaning that utilization of BMCA is limited only to predefined Grandmaster clock devices. Based on these facts a new example network topology was designed, as can be seen in 4.12, based on the previous topologies described in 1.3. All devices located in LAN after the Boundary clock would be using a virtual Macsec network interface, where the Transparent clock would receive messages through a virtual interface closer to the Boundary clock, update the Correction field in the message, and send the message further through

the other virtual interface. The Boundary clock would communicate via the IPsec tunnel directly with the Grandmaster clock or through other non-blocking intermediate nodes. Assuming that hardware timestamping would be used, the impact on PTP precision would be very negligible.

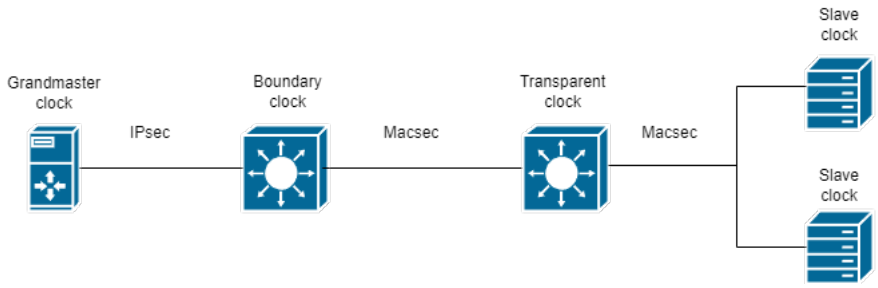


Fig. 4.12: Topology with multiple layers of synchronization and encryption

If software timestamping was to be applied, the devices in the network must consider the fact that stability is not always guaranteed, even if no security protocol is applied.

Based on the results, the least optimal protocol to be used with PTP is Wireguard. Although the easiest to configure and manage, Wireguard-go provided the worst results in all tested scenarios. As was mentioned before, this is likely due to the fact that only the user space implementation could be used. If used for encryption of PTP, it should be considered only if there are no other options available.

Conclusion

The Master's thesis focused on the explanation of PTP concepts and systems properties, the selection of security protocols, their implementation over PTP and the determination of the most optimal security protocol based on the measurement results obtained by using a specialized tool designed for PTP parameter measurement and analysis.

In the first part of the thesis, the research was done as to explain the complete functionality of PTP, including its possibilities and operations. The focus was put specifically on the synchronization process and network properties as well as the possible use case scenarios. The following chapter provided a selection of possible security protocols which could be used to implement a cyber security of PTP, which is by default non-existent. The security protocols were analyzed primarily from the network attribute perspective, including the considerations for PTP operations.

The following part of the thesis focused on further research of the practical implementation of previously described concepts in Linux. The creation of the custom Linux image for an embedded system with an ARM processor was required, including the addition of necessary layers and packages. The same chapter explained the timing properties in the Linux kernel as well as the explanation and usage of the user space PTP driver tools. This was followed by practical implementation for each security protocol, where all protocols were configured to be PTP-compatible. The Wireguard and IPsec security protocols were configured using tools provided by relevant drivers, where however some limitations were observed in regard to communication options and timestamping possibilities. The Macsec protocol had to be enabled by direct patching of the relevant kernel driver, in order to provide required core functionalities for *ptp4l* and make the synchronization process possible.

The final part of the thesis was concerned with the measurement and result analysis. This involved the creation of a multifunctional tool for real-time raw data capture and visualization, including the automated test-case scenario creation and final result comparison and statistical data analysis. With the help of this tool, a broad range of operational scenario tests could be performed and relevant data could be obtained. Upon the analysis of all the obtained data, it could be determined that neither IPsec nor Macsec security protocols have a noticeable impact on the PTP parameters if used with hardware timestamping. However, in the scenarios where the software timestamping is used, instabilities persisted across all operational modes. The user space implementation of Wireguard-go showed the worst results and can be deemed to be least suited for providing encryption for PTP.

It can be concluded that based on the provided results and possibilities of each security protocol, the Macsec, with the timestamping patch in the kernel driver, provides the most optimized security implementation for PTP, due to its possibility to work over all PTP operational modes as well as having almost no impact on the synchronization parameters.

Bibliography

[1] Precision Time Protocol Version 2 (PTPv2) Management Information Base *Internet Engineering Task Force (IETF)*, 2017, https://www.rfc-editor.org/rfc/rfc8173.html, *8173*, *2070-1721*,

[2] Crystal Oscillator, *Technopedia*, 2015, https://www.techopedia.com/definition/2245/crystal-oscillator. [Online; accessed 24-October-2023].

[3] History of NTP, *NTPsec*, 2023, https://docs.ntpsec.org/latest/history.html, [Online; accessed 24-October-2023].

[4] IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008), *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, 2020, 10.1109/IEEESTD.2020.9120376.

[5] What's the difference between types of clocks?, *Tekron*, 2023, https://tekron.com/news/release/differenttypesofclocks/#:~:text=What%20is%20a%20boundary%20clock,to%20pass%20down%20the%20network., [Online; accessed 24-October-2023].

[6] What are bits, bytes, and other units of measure for digital information?, *University Information Technology Services*, 2018, https://kb.iu.edu/d/ackw#:~:text=A%20bit%20is%20a%20binary,one%20bit%20at%20a%20time. [Online; accessed 9-January-2024].

[7] Introduction to TCP/IP (Part 2) - Five Layer Model and Applications, *Microchip* 2023, https://microchipdeveloper.com/xwiki/bin/view/applications/tcp-ip/five-layer-model-and-apps/#HTCP2FIPFive-LayerSoftwareModelOverview. [Online; accessed 9-January-2024].

[8] Synchronizace v distribuovaných řídicích systémech: Precision Time Protocol (PTP) podle IEEE 1588 (in Czech), *Časopis Automa - časopis pro automatizační techniku*, 2010, *Roč. 2010, č. 02, s. 3.*

[9] IEC/IEEE 61588-2021, *IEC/IEEE International Standard - Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, 2021, 10.1109/IEEESTD.2021.9456762.

[10] Adaptive Computing Documentation Portal, *AMD*, 2023, https://docs.xilinx.com/r/en-US/ug1602-ptp-user/How-PTP-Works, [Online; accessed 24-October-2023].

[11] Nokia help, *Nokia*, 2023, https://infocenter.nokia.com/public/7705SAR234R1A/index.jsp?topic=%2Fcom.nokia.basic-system-guide%2Fbest-master-clock-algorithm.html, [Online; accessed 24-October-2023].

[12] PTP: Timing accuracy and precision for the future of computing, *Engineering at Meta*, 2023, https://engineering.fb.com/2022/11/21/production-engineering/future-computing-ptp/, [Online; accessed 24-October-2023].

[13] Understanding 1588v2, *Huawei*, 2023, https://support.huawei.com/enterprise/br/doc/EDOC1100278756/835fea49/understanding-1588v2-g82751, [Online; accessed 24-October-2023].

[14] What is round-trip time? | RTT definition | cloudflare, *Cloudfare*, 2023, https://www.cloudflare.com/learning/cdn/glossary/round-trip-time-rtt/, [Online; accessed 24-Octobe-2023].

[15] End to End Versus Peer to Peer, *Tekron*, 2019, https://tekron.com/news/release/end-to-end-versus-peer-to-peer/, [Online; accessed 11-November-2023].

[16] Low Latency Ethernet 10G MAC Intel® FPGA IP User Guide, *Intel*, 2023, https://www.intel.com/content/www/us/en/docs/programmable/683426/18-1-18-1/about-ll-ethernet-10g-mac.html, [Online; accessed 14-December-2023].

[17] PTP over Ethernet, *Juniper*, 2022, https://www.juniper.net/documentation/us/en/software/junos/time-mgmt/topics/topic-map/ptp-over-ethernet.html, [Online; accessed 14-December-2023].

[18] What You Should Know about Ethernet Frame Format?, *MiniTool*, 2023, https://www.minitool.com/lib/ethernet-frame.html, [Online; accessed 14-December-2023].

[19] MAC, *Computer Hope*, 2022, https://www.computerhope.com/jargon/m/mac.htm. [Online; accessed 16-December-2023].

[20] Ptp, *Wireshark Wiki*, 2023, https://wiki.wireshark.org/Protocols/ptp.md. [Online; accessed 18-December-2023].

[21] Implemetning IEEE 1588v2 for use in the mobile backhaul, *Caltex Solutions Ltd.*, 2009, https://opencores.org/websvn/filedetails?repname=ha1588&path=%2Fha1588%2Ftrunk%2Fdoc%2Ftool%2Fptpv2_timing_analyzer%2FCalnex_-_IEEE_1588v2_PTP.pdf [Online; accessed 18-December-2023].

[22] BERDANI, D., TIPPENHAUER, N., MELIS A., ET AL., *Time sensitive networking security: issues of precision time protocol and its implementation*, 2023, 10.1186/s42400-023-00140-5, *vol. 6, no. 8*

[23] Introduction to Precision Time Protocol (PTP), *NetworkLessons*, 2023, https://networklessons.com/cisco/ccnp-encor-350-401/introduction-to-precision-time-protocol-ptp, [Online; accessed 28-December-2023].

[24] ALGHAMDI W. , SCHUKAT M., *Precision time protocol attack strategies and their resistance to existing security extensions*, 2021, 10.1186/s42400-021-00080-y. *vol. 4, no. 12*,

[25] IPsec vs. WireGuard, *tailscale.com*, 2024, https://tailscale.com/compare/ipsec, [Online; accessed 28-December-2023].

[26] WireGuard: Next Generation Kernel Network Tunnel, *wireguard.com*, 2020, https://wireguard.com/papers/wireguard.pdf, [Online; accessed 28-December-2023].

[27] WireGuard: The Next-Gen VPN Protocol, *KeySight*, 2022, https://www.keysight.com/blogs/en/tech/nwvs/2022/09/22/wireguard-the-next-gen-vpn-protocol. [Online; accessed 29-December-2023].

[28] IPsec (Internet Protocol Security), *NetworkLessons*, 2023, https://networklessons.com/cisco/ccie-routing-switching/ipsec-internet-protocol-security, [Online; accessed 8-January-2024].

[29] What Is IPsec?, *Huawei*, 2023, https://info.support.huawei.com/info-finder/encyclopedia/en/IPsec.html, [Online; accessed 7-January-2024].

[30] Internet Key Exchange, *Juniper*, 2023, https://www.juniper.net/documentation/us/en/software/junos/vpn-ipsec/topics/topic-map/security-ike-basics.html, [Online; accessed 8-January-2024].

[31] Encapsulating Security Payload, *IBM*, 2023, https://www.ibm.com/docs/en/i/7.4?topic=protocols-encapsulating-security-payload, [Online; accessed 8-January-2024].

[32] Media Access Control Security MACsec Overview, *Study CCNP*, 2024, https://study-ccnp.com/media-access-control-security-macsec-overview/, [Online; accessed 3-January-2024].

[33] MACsec frame format, *Ruckus Wireless*, 2024, https://docs.ruckuswireless.com/fastiron/08.0.60/fastiron-08060-securityguide/GUID-333630FE-363D-43F1-A4C9-0EDD0D0D53E2.html, [Online; accessed 7-January-2024].

[34] MACsec: a different solution to encrypt network traffic, *Red Hat Developer*, 2016, https://developers.redhat.com/blog/2016/10/14/macsec-a-different-solution-to-encrypt-network-traffic, [Online; accessed 7-January-2024].

[35] Ethernet over IP (EoIP), *NetworkLessons*, 2024, https://notes.networklessons.com/ethernet-over-ip-(eoip), [Online; accessed 8-January-2024].

[36] Embedded Linux for i.MX Applications Processors, *NXP*, 2024, https://www.nxp.com/design/design-center/software/embedded-software/i-mx-software/embedded-linux-for-i-mx-applications-processors:IMXLINUX, [Online; accessed 12-February-2024].

[37] Yocto Project Compatible Layers, *The yocto project*, 2024, https://www.yoctoproject.org/development/yocto-project-compatible-layers/, [Online; accessed 12-February-2024].

[38] Yocto Project Overview and Concepts Manual, *The Yocto project*, 2024, https://docs.yoctoproject.org/2.6.1/overview-manual/overview-manual.html, [Online; accessed 12-February-2024].

[39] Technical overview, *The Yocto project*, 2024, https://www.yoctoproject.org/development/technical-overview/, [Online; accessed 12-February-2024].

[40] Images, *The Yocto project*, 2024, https://docs.yoctoproject.org/ref-manual/images.html, [Online; accessed 12-February-2024].

[41] Building, *The Yocto project*, 2024, https://docs.yoctoproject.org/ref-manual/building.html, [Online; accessed 12-February-2024].

[42] Creating Partitioned Images Using Wic, *The Yocto project*, 2024, https://docs.yoctoproject.org/dev-manual/wic.html, [Online; accessed 12-February-2024].

[43] IPv4 Address Blocks Reserved for Documentation, *Internet Engineering Task Force (IETF)*, 2010, https://www.rfc-editor.org/rfc/rfc5737, *5737*, *2070-1721*

[44] chrony.conf(5) Manual Page, *Chrony project*, 2024, https://chrony-project.org/doc/3.4/chrony.conf.html, [Online; accessed 12-February-2024].

[45] Přesný čas - veřejné NTP servery (in Czech), *FinalTek.com*, 2024, https://shop.finaltek.com/index.php/knowledgebase/57/P%C5%99esny-%C4%8Das---ve%C5%99ejne-NTP-servery.html, [Online; accessed 12-February-2024].

[46] yoctoproject / poky, *github.com*, 2024, https://github.com/yoctoproject/poky/commit/2e07f1440f36d0efc304f1dbe8c1, [Online; accessed 25-April-2024].

[47] openembedded / meta-openembedded, *github.com*, 2024, https://github.com/openembedded/meta-openembedded/commit/e12d38e91efff3f, [Online; accessed 25-April-2024].

[48] openembedded / meta-openembedded, *github.com*, 2024, https://github.com/openembedded/meta-openembedded/commit/5be2e20157f30, [Online; accessed 12-February-2024].

[49] PTP hardware clock infrastructure for Linux, *The Linux Kernel*, 2024, https://docs.kernel.org/driver-api/ptp.html, [Online; accessed 25-April-2024].

[50] Timestamping, *The Linux Kernel*, 2024, https://docs.kernel.org/networking/timestamping.html, [Online; accessed 23-January-2024].

[51] Socket(7) — Linux manual page, *man7*, 2023, https://man7.org/linux/man-pages/man7/socket.7.html, [Online; accessed 23-January-2024].

[52] Linux Kernel Support for IEEE 1588 Hardware Timestamping, *Renesas*, 2021, https://www.renesas.com/us/en/document/whp/linux-kernel-support-ieee-1588-hardware-timestamping, [Online; accessed 23-January-2024].

[53] WEIDONG Y., *IEEE1588 Clock servo algorithm*, 2009, 10.1109/ICEMI.2009.5274861, *pp. 1-341-1-344*

[54] Synchronizing Time with Linux* PTP *Avnu Alliance*, 2023, https://tsn.readthedocs.io/timesync.html, [Online; accessed 23-January-2024].

[55] Leap second and UT1-UTC information, *NIST*, 2023, https://www.nist.gov/pml/time-and-frequency-division/time-realization/leap-seconds#:~:text=The%20current%20difference%20between%20UTC%20and%20TAI%20is%2037%20seconds.%20(&text=The%20table%20below%20lists%20all,or%20are%20scheduled%20to%20occur., [Online; accessed 23-January-2024].

[56] ethtool(8) — Linux manual page, *man7*, 2023, https://man7.org/linux/man-pages/man8/ethtool.8.html, [Online; accessed 29-January-2024].

[57] ptp4l(8) - Linux man page, *die.net*, 2024, https://linux.die.net/man/8/ptp4l, [Online; accessed 29-January-2024].

[58] Configuring PTP Using ptp4l, *Fedora Project Docs*, 2024, https://docs.fedoraproject.org/en-US/fedora/latest/system-administrators-guide/servers/Configuring_PTP_Using_ptp4l/, [Online; accessed 5-February-2024].

[59] pmc(8) - Linux man page, *die.net*, 2024, https://linux.die.net/man/8/pmc, [Online; accessed 6-February-2024].

[60] pmc(8): ptp management client *linuxptp project*, 2024, https://linuxptp.nwtime.org/documentation/pmc/, [online; accessed 6-february-2024].

[61] Software User Manual, *Ouster*, 2024, https://data.ouster.io/downloads/software-user-manual/software-user-manual-v2.1.0.pdf, [online; accessed 6-february-2024].

[62] phc2sys(8) - Linux man page, *die.net*, 2024, https://linux.die.net/man/8/phc2sys, [online; accessed 6-february-2024].

[63] ip-macsec(8) — Linux manual page, *Red Hat Blog*, 2024, https://www.redhat.com/en/blog/architecting-containers-part-1-why-understanding-user-space-vs-kernel-space- [online; accessed 20-february-2024].

[64] Universal TUN/TAP device driver, *kernel.org*, 2002, https://www.kernel.org/doc/Documentation/networking/tuntap.txt, [online; accessed 18-february-2024].

[65] Introduction to strongSwan, *strongSwan Documentation*, 2024, https://docs.strongswan.org/docs/5.9/howtos/introduction.html, [online; accessed 27-february-2024].

[66] swanctl.conf, *strongSwan Documentation*, 2024, https://docs.strongswan.org/docs/5.9/swanctl/swanctlConf.html, [online; accessed 27-february-2024].

[67] forecast Plugin, *strongSwan Documentation*, 2024, https://docs.strongswan.org/docs/5.9/plugins/forecast.html, [online; accessed 27-february-2024].

[68] Unicast PTP, *meinberg*, 2024, https://blog.meinbergglobal.com/2014/04/16/unicast-ptp/, [online; accessed 27-february-2024].

[69] Frequently Asked Questions (FAQ), *strongSwan Documentation*, 2024, https://docs.strongswan.org/docs/5.9/support/faq.html, [online; accessed 27-february-2024].

[70] ip-macsec(8) — Linux manual page, *man7*, 2023, https://man7.org/linux/man-pages/man7/socket.7.html, [Online; accessed 14-March-2024].

[71] urandom(4) - Linux man page *die.net*, 2024, https://linux.die.net/man/4/urandom, [Online; accessed 14-March-2024].

[72] What's new in MACsec, *Red Hat Blog*, 2017, https://developers.redhat.com/blog/2017/06/28/whats-new-in-macsec-setting-up-macsec-using-wpa_supplicant-and-optionally-networkmanager, [online; accessed 18-May-2024].

[73] ethtool kernel driver, *googlesource.com*, 2024, https://gfiber.googlesource.com/kernel/quantenna/+/master/net/core/ethtool.c, [Online; accessed 17-April-2024].

[74] vlan kernel driver, *bootlin.com*, 2024, https://elixir.bootlin.com/linux/v4.19.312/source/net/8021q/vlan_dev.c, [Online; accessed 17-April-2024].

[75] Paramiko, *paramiko.org*, 2024, https://www.paramiko.org/ [Online; accessed 18-April-2024].

[76] Paramiko Documentation, *paramiko.org*, 2024, https://docs.paramiko.org/en/latest/api/client.html [Online; accessed 22-April-2024].

[77] How to Calculate Standard Deviation (Guide) *Scribbr*, 2024, https://www.scribbr.com/statistics/standard-deviation/ [Online; accessed 18-May-2024].

[78] Outliers Detection Using IQR Z-score LOF and DBSCAN, *analyticsvidhya.com*, 2024, https://www.analyticsvidhya.com/blog/2022/10/outliers-detection-using-iqr-z-score-lof-and-dbscan/ [Online; accessed 9-May-2024].

Symbols and abbreviations

AH	Authentication Header
BMCA	Best Master Clock Algorithm
BSP	Board Support Package
CLI	Command-line interface
E2E	End-to-end
EoIP	Ethernet over IP
ESP	Encapsulating Security Payload
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
ICV	Integrity Check Value
IEEE	Institute of Electrical and Electronics Engineers
IKE	Internet Key Exchange
IP	Internet Protocol
IPsec	Internet Protocol security
IQR	Interquartile range
LAN	Local Area Network
MAC	Medium access control
MACsec	Medium access control security
NIC	Network interface controller
NTP	Network Time Protocol
OSI	Open Systems Interconnection
OUI	Organizational Unique Identifier
P2P	Peer-to-Peer

ppb	Parts Per Billion
PSK	Pre-shared key
PTP	Precision Time Protocol
QSDO	Qualified standards development organizations
RTT	Round-trip time
RFC	Request for Comments
SSH	Secure Shell Protocol
stdin	standard input
stdout	standard output
TAI	International Atomic Time
TCP/IP	Transmission Control Protocol/Internet Protocol
TLV	Type Length Values
UDP	User Datagram Protocol
UTC	Coordinated Universal Time
VLAN	Virtual local area network
VPN	Virtual private network

List of appendices

A Additional measrument results

A.1 Ptp4l calculated path delay, Hardware timestamping

path_delay	multicast_udp	multicast_l2	unicast_udp	unicast_l2
mean [ns]	454.113107	453.933537	453.960275	454.028648
median [ns]	454.000000	454.000000	454.000000	454.000000
mean - median [ns]	0.113107	-0.066463	-0.039725	0.028648
abs. mean dev. [ns]	1.319782	1.446615	1.328928	1.327300
abs. median dev. [ns]	1.000000	1.000000	1.000000	1.000000
standard dev. [ns]	1.691368	1.841757	1.716975	1.690179
variance [ns^2]	2.860726	3.392069	2.948002	2.856704
spike prob. (Z-score) [%]	0.191058	0.267380	0.114591	0.000000
spike prob. (IQR) [%]	0.000000	0.000000	0.000000	0.000000

Tab. A.1: Master to Slave path delay statistics, No encryption, Hardware timestamping

path_delay	unicast_udp_tunnel	unicast_udp_transport
mean [ns]	453.964463	454.139855
median [ns]	454.000000	454.000000
mean - median [ns]	-0.035537	0.139855
abs. mean dev. [ns]	1.394483	1.321878
abs. median dev. [ns]	1.000000	1.000000
standard dev. [ns]	1.789206	1.681204
variance [ns^2]	3.201259	2.826448
spike prob. (Z-score) [%]	0.229270	0.267482
spike prob. (IQR) [%]	0.000000	0.000000

Tab. A.2: Master to Slave path delay statistics, IPsec encryption, Hardware timestamping

path_delay	multicast_udp	multicast_l2	unicast_udp	unicast_l2
mean [ns]	454.089415	454.057678	453.964477	453.908709
median [ns]	454.000000	454.000000	454.000000	454.000000
mean - median [ns]	0.089415	0.057678	-0.035523	-0.091291
abs. mean dev. [ns]	1.350923	1.334134	1.294966	1.431871
abs. median dev. [ns]	1.000000	1.000000	1.000000	1.000000
standard dev. [ns]	1.719438	1.715686	1.712723	1.788383
variance [ns^2]	2.956468	2.943579	2.933421	3.198312
spike prob. (Z-score) [%]	0.000000	0.267380	0.420168	0.267380
spike prob. (IQR) [%]	0.000000	0.000000	0.000000	0.000000

Tab. A.3: Master to Slave path delay statistics, Macsec encryption, Hardware timestamping

A.2 Ptp4l calculated path delay, Software timestamping

path_delay	multicast_udp	multicast_l2	unicast_udp	unicast_l2
mean [ns]	1.100064e+06	1.095549e+06	1.098648e+06	1.092712e+06
median [ns]	1.100063e+06	1.095704e+06	1.098612e+06	1.092796e+06
mean - median [ns]	0.674865	-154.989623	36.456572	-83.591241
abs. mean dev. [ns]	3160.049712	3064.749497	3230.511898	3158.760717
abs. median dev. [ns]	2661.000000	2519.500000	2797.500000	2541.000000
standard dev. [ns]	3943.974344	3900.990038	4041.844647	4075.721799
variance [ns^2]	1.555493e+07	1.521772e+07	1.633651e+07	1.661151e+07
spike prob. (Z-score) [%]	0.076864	0.576480	0.269024	1.037265
spike prob. (IQR) [%]	0.000000	0.000000	0.115296	0.000000

Tab. A.4: Master to Slave path delay statistics, No encryption, Software timestamping

path_delay	unicast_udp_tunnel	unicast_udp_transport
mean [ns]	1.097696e+06	1.097170e+06
median [ns]	1.097765e+06	1.097478e+06
mean - median [ns]	-68.958493	-308.159877
abs. mean dev. [ns]	2983.542918	2975.457626
abs. median dev. [ns]	2595.500000	2353.000000
standard dev. [ns]	3689.165854	3816.215386
variance [ns^2]	1.360994e+07	1.456350e+07
spike prob. (Z-score) [%]	0.076864	0.922367
spike prob. (IQR) [%]	0.000000	0.038432

Tab. A.5: Master to Slave path delay statistics, IPsec encryption, Software timestamping

path_delay	multicast_udp	multicast_l2	unicast_udp	unicast_l2
mean [ns]	1.096709e+06	1.096480e+06	1.095627e+06	1.094645e+06
median [ns]	1.096952e+06	1.096579e+06	1.095687e+06	1.094828e+06
mean - median [ns]	-243.259416	-99.265847	-59.994237	-183.453497
abs. mean dev. [ns]	3065.950005	2746.566042	2823.630742	2933.832196
abs. median dev. [ns]	2477.500000	2243.000000	2520.000000	2423.000000
standard dev. [ns]	3941.093497	3516.332065	3516.498512	3721.557111
variance [ns^2]	1.553222e+07	1.236459e+07	1.236576e+07	1.384999e+07
spike prob. (Z-score) [%]	0.499616	0.691510	0.691510	0.499616
spike prob. (IQR) [%]	0.000000	0.192086	0.000000	0.000000

Tab. A.6: Master to Slave path delay statistics, Macsec encryption, Software timestamping

path_delay	multicast_udp	unicast_udp
mean [ns]	3.037686e+06	3.148619e+06
median [ns]	3.037105e+06	3.160712e+06
mean - median [ns]	580.997310	-12092.142583
abs. mean dev. [ns]	151928.237884	231503.076855
abs. median dev. [ns]	125753.000000	196881.500000
standard dev. [ns]	193528.400297	289928.906398
variance [ns^2]	3.745324e+10	8.405877e+10
spike prob. (Z-score) [%]	0.307456	0.230592
spike prob. (IQR) [%]	0.000000	0.000000

Tab. A.7: Master to Slave path delay statistics, Wireguard encryption, Software timestamping

B Software versions used in the project

Software	Version
Linux Kernel	4.19.35
Ethtool	4.19
Ptp4l	4.1
Iproute2	6.7.0
Wireguard-go	0.0.20230223
StrongSwan	5.9.13
Macsec	4.19
Tshark	3.0.3

Tab. B.1: Versions of used software

C Encryption algorithms used by security protocols

Security Protocol	Encryption Algorithm
WireGuard	ChaCha20
IPSec	AES-128-GCM
MACsec	AES-128-GCM

Tab. C.1: Encryption algorithms used by security protocols

D Content of the electronic attachment

The electronic attachment to this thesis contains a directory with a measurement script, a directory with meta layers, and a directory with measurement results and statistics. *Pcap* files are not included so as not to reveal proprietary device information. All *png* images were downsized so as not to exceed the electrical appendix submission criteria. *README.md* files were included as to provide additional clarification and sourcing.

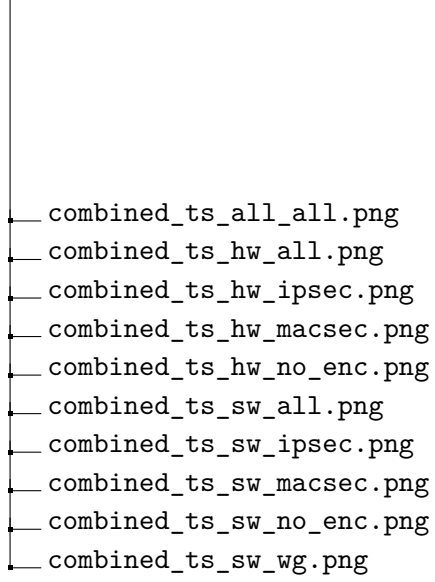
```
/ ..... root of the attached archive
├── README.md
├── meta-custom ..... custom Yocto meta layers used for Linux image creation
│   ├── README.md
│   └── meta-custom
│       ├── recipes-connectivity
│       │   ├── iproute2
│       │   │   ├── iproute2
│       │   │   │   └── 0001-libc-compat.h-add-musl-workaround.patch
│       │   │   └── iproute2_6.7.0.bb
│       │   └── linuxptp
│       │       ├── linuxptp
│       │       │   ├── 0001-include-string.h-for-strncpy.patch
│       │       │   ├── 0002-linuxptp-Use-CC-in-incdefs.sh.patch
│       │       │   └── systemd
│       │       │       ├── phc2sys@.service.in
│       │       │       └── ptp4l@.service.in
│       │       └── linuxptp_4.1.bb
│       ├── recipes-kernel
│       │   ├── linux
│       │   │   ├── linux-rt
│       │   │   │   └── my_kernel
│       │   │   │       ├── 0001_ethtool_macsec.patch
│       │   │   │       ├── 0002_debug_ethtool.patch
│       │   │   │       └── 0003_crypto_kernel_reqs.patch
│       │   │   └── linux-rt.bbappend
│       │   └── wireguard
│       │       └── wireguard-tools_%.bbappend
│       └── recipes-support
│           └── strongswan
│               └── strongswan_5.9.13.bb
└── setup_and_measure ..... python files
    ├── README.md
    ├── confdata.yml ..... measurment specification file
    ├── requirements.txt
    └── setup_and_measure ..... configuration and measurement script
```

```
    ├── __main__.py
    ├── class_utils.py
    ├── files_packages.py
    ├── logger.py
    ├── networking.py
    ├── ptp_config_files.py
    ├── ptp_reader.py
    ├── sec.py
    ├── stats_compare.py
    └── vardata.py
└── results. . . . . all results generated by measurement script
    ├── network-caps. . . . . directory for .pcap files
    ├── real-time-raw-plots. . . . . plots updated in real time
    │   ├── ipsec_enc_unicast_udp_hw_transport.png
    │   ├── ipsec_enc_unicast_udp_hw_tunnel.png
    │   ├── ipsec_enc_unicast_udp_sw_transport.png
    │   ├── ipsec_enc_unicast_udp_sw_tunnel.png
    │   ├── macsec_enc_multicast_l2_hw.png
    │   ├── macsec_enc_multicast_l2_sw.png
    │   ├── macsec_enc_multicast_udp_hw.png
    │   ├── macsec_enc_multicast_udp_sw.png
    │   ├── macsec_enc_unicast_l2_hw.png
    │   ├── macsec_enc_unicast_l2_sw.png
    │   ├── macsec_enc_unicast_udp_hw.png
    │   ├── macsec_enc_unicast_udp_sw.png
    │   ├── no_enc_multicast_l2_hw.png
    │   ├── no_enc_multicast_l2_sw.png
    │   ├── no_enc_multicast_udp_hw.png
    │   ├── no_enc_multicast_udp_sw.png
    │   ├── no_enc_unicast_l2_hw.png
    │   ├── no_enc_unicast_l2_sw.png
    │   ├── no_enc_unicast_udp_hw.png
    │   ├── no_enc_unicast_udp_sw.png
    │   ├── wg_enc_multicast_udp_sw.png
    │   └── wg_enc_unicast_udp_sw.png
    └── real-time-raw-values. . . . . data updated in real time
        ├── ipsec_enc_unicast_udp_hw_transport.csv
        ├── ipsec_enc_unicast_udp_hw_tunnel.csv
        ├── ipsec_enc_unicast_udp_sw_transport.csv
        ├── ipsec_enc_unicast_udp_sw_tunnel.csv
        ├── macsec_enc_multicast_l2_hw.csv
        ├── macsec_enc_multicast_l2_sw.csv
        ├── macsec_enc_multicast_udp_hw.csv
        ├── macsec_enc_multicast_udp_sw.csv
        ├── macsec_enc_unicast_l2_hw.csv
        └── macsec_enc_unicast_l2_sw.csv
```

```
├── macsec_enc_unicast_udp_hw.csv
├── macsec_enc_unicast_udp_sw.csv
├── no_enc_multicast_l2_hw.csv
├── no_enc_multicast_l2_sw.csv
├── no_enc_multicast_udp_hw.csv
├── no_enc_multicast_udp_sw.csv
├── no_enc_unicast_l2_hw.csv
├── no_enc_unicast_l2_sw.csv
├── no_enc_unicast_udp_hw.csv
├── no_enc_unicast_udp_sw.csv
├── wg_enc_multicast_udp_sw.csv
└── wg_enc_unicast_udp_sw.csv
└── stats_plots_comparisons.....statistics created after measurment
    ├── box.....box plots
    │   ├── ipsec_enc_unicast_udp_hw_transport_iqr_3.5.png
    │   ├── ipsec_enc_unicast_udp_hw_tunnel_iqr_3.5.png
    │   ├── ipsec_enc_unicast_udp_sw_transport_iqr_3.5.png
    │   ├── ipsec_enc_unicast_udp_sw_tunnel_iqr_3.5.png
    │   ├── macsec_enc_multicast_l2_hw_iqr_3.5.png
    │   ├── macsec_enc_multicast_l2_sw_iqr_3.5.png
    │   ├── macsec_enc_multicast_udp_hw_iqr_3.5.png
    │   ├── macsec_enc_multicast_udp_sw_iqr_3.5.png
    │   ├── macsec_enc_unicast_l2_hw_iqr_3.5.png
    │   ├── macsec_enc_unicast_l2_sw_iqr_3.5.png
    │   ├── macsec_enc_unicast_udp_hw_iqr_3.5.png
    │   ├── macsec_enc_unicast_udp_sw_iqr_3.5.png
    │   ├── no_enc_multicast_l2_hw_iqr_3.5.png
    │   ├── no_enc_multicast_l2_sw_iqr_3.5.png
    │   ├── no_enc_multicast_udp_hw_iqr_3.5.png
    │   ├── no_enc_multicast_udp_sw_iqr_3.5.png
    │   ├── no_enc_unicast_l2_hw_iqr_3.5.png
    │   ├── no_enc_unicast_l2_sw_iqr_3.5.png
    │   ├── no_enc_unicast_udp_hw_iqr_3.5.png
    │   ├── no_enc_unicast_udp_sw_iqr_3.5.png
    │   ├── wg_enc_multicast_udp_sw_iqr_3.5.png
    │   └── wg_enc_unicast_udp_sw_iqr_3.5.png
    ├── csvs. ..... numerical statistical values for each parameter
    │   ├── master_offset_statistics_hw.csv
    │   ├── master_offset_statistics_sw.csv
    │   ├── path_delay_statistics_hw.csv
    │   ├── path_delay_statistics_sw.csv
    │   ├── servo_freq_statistics_hw.csv
    │   ├── servo_freq_statistics_sw.csv
    │   ├── servo_statistics_hw.csv
    │   └── servo_statistics_sw.csv
    └── histograms. ..... histograms
```

- ipsec_enc_unicast_udp_hw_transport_hist_rice.png
- ipsec_enc_unicast_udp_hw_tunnel_hist_rice.png
- ipsec_enc_unicast_udp_sw_transport_hist_rice.png
- ipsec_enc_unicast_udp_sw_tunnel_hist_rice.png
- macsec_enc_multicast_l2_hw_hist_rice.png
- macsec_enc_multicast_l2_sw_hist_rice.png
- macsec_enc_multicast_udp_hw_hist_rice.png
- macsec_enc_multicast_udp_sw_hist_rice.png
- macsec_enc_unicast_l2_hw_hist_rice.png
- macsec_enc_unicast_l2_sw_hist_rice.png
- macsec_enc_unicast_udp_hw_hist_rice.png
- macsec_enc_unicast_udp_sw_hist_rice.png
- no_enc_multicast_l2_hw_hist_rice.png
- no_enc_multicast_l2_sw_hist_rice.png
- no_enc_multicast_udp_hw_hist_rice.png
- no_enc_multicast_udp_sw_hist_rice.png
- no_enc_unicast_l2_hw_hist_rice.png
- no_enc_unicast_l2_sw_hist_rice.png
- no_enc_unicast_udp_hw_hist_rice.png
- no_enc_unicast_udp_sw_hist_rice.png
- wg_enc_multicast_udp_sw_hist_rice.png
- wg_enc_unicast_udp_sw_hist_rice.png

- packet_time_deltas..... plots depicting times between consecutive packets/frames
  - ipsec_enc_unicast_udp_hw_transport.png
  - ipsec_enc_unicast_udp_hw_tunnel.png
  - ipsec_enc_unicast_udp_sw_transport.png
  - ipsec_enc_unicast_udp_sw_tunnel.png
  - macsec_enc_multicast_l2_hw.png
  - macsec_enc_multicast_l2_sw.png
  - macsec_enc_multicast_udp_hw.png
  - macsec_enc_multicast_udp_sw.png
  - macsec_enc_unicast_l2_hw.png
  - macsec_enc_unicast_l2_sw.png
  - macsec_enc_unicast_udp_hw.png
  - macsec_enc_unicast_udp_sw.png
  - no_enc_multicast_l2_hw.png
  - no_enc_multicast_l2_sw.png
  - no_enc_multicast_udp_hw.png
  - no_enc_multicast_udp_sw.png
  - no_enc_unicast_l2_hw.png
  - no_enc_unicast_l2_sw.png
  - no_enc_unicast_udp_hw.png
  - no_enc_unicast_udp_sw.png
  - wg_enc_multicast_udp_sw.png
  - wg_enc_unicast_udp_sw.png


combined_ts_all_all.png
combined_ts_hw_all.png
combined_ts_hw_ipsec.png
combined_ts_hw_macsec.png
combined_ts_hw_no_enc.png
combined_ts_sw_all.png
combined_ts_sw_ipsec.png
combined_ts_sw_macsec.png
combined_ts_sw_no_enc.png
combined_ts_sw_wg.png